A Micro-Coded Controller for a Medium Data Rate Satellite Payload Simulator

by

Kimberly Ann Smith

Submitted to the

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February, 1994

Signature of Author____
Department of Electrical Engineering and Computer Science
January 14, 1994

Certified by____
Professor James K. Roberge
Thesis Supervisor

Certified by____
Russell R. Rhodes
Company Supervisor (MIT Lincoln Laboratory)

Accepted by____
Frederic R. Morgenthaler
Chairman, Committee on Graduate Students

A Micro-Coded Controller for a Medium Data Rate Satellite Payload Simulator

by

Kimberly Ann Smith

Submitted to the Department of Electrical Engineering and Computer Science on
January 14, 1994 in partial fulfillment of the requirements for the
Degree of Master of Science in Electrical Engineering

ABSTRACT

Lincoln Laboratory is currently developing a payload simulator designed to operate according to Milstar standards. It is being developed as a testbed for the U.S. Army and will be used for the Army and other branches of the military to test their terminals prior to a Milstar satellite launch. This thesis project will entail all aspects of the design and development of the onboard processor from digital logic design to software design for data manipulation. The final thesis will be an essential part of a working satellite communication system.

Thesis Supervisor:	Russell R. Rhodes
Title:	Assistant Group Leader, Group 64, MIT Lincoln Laboratory
Thesis Supervisor:	Professor James K. Roberge
Title:	Professor of Electrical Engineering

Acknowledgment & Dedication

I would first like to thank Russ Rhodes for his guidance, patience, and wisdom over the past several years. His mentorship has served me well at both Lincoln Laboratory and MIT. I would also like to thank all of the members of group 64 for making my time spent with you so enjoyable and rewarding. Without John Pineau and Jack Kangas to provide listening ears and supportive advice, this would have been a much longer journey. I would like to thank Ken Clarke for kindly sharing his office space and time during my time at Lincoln. Your help and patience have been greatly appreciated. I would also like to thank Dave Cipolle for his assistance in the debugging the MOP and fighting with the Xilinx reps.

I would like to dedicate this thesis to my grandmother and grandfather Cripe for their support, guidance, and understanding over the last five years. Also to James Goldinger for his friendship and love that are such an important and vital part of my life.

Contents

1 Introduction

2 Performance Requirements

List of Figures

Chapter 1

Introduction

1.1 Motivation

The Department of Defense is developing a highly robust satellite communication system (Milstar) which will operate at EHF (44 GHz Uplinks/20 GHz Downlinks) and incorporate spread spectrum (frequency hopping) signals and on-board signal processing at low data rates (LDR; 75 bps to 2400 bps) and medium data rates (MDR; 4.8 Kbps to 1500 Kbps). The Milstar satellites also incorporate demand-assigned multiple access (DAMA) techniques. Lincoln laboratory is building a payload simulator designed to operate according to Milstar standards. It is being developed as a testbed for the U.S. Army and will be used for the Army and other branches of the military to test their terminals prior to a Milstar satellite launch. The time line for this project calls for a completed payload simulator by the summer of 1994.

The subject of this thesis is the development of a particular division of the payload simulator. The proposed MDR onboard processor (MOP) is designed to operate as an interface between the enhanced microprocessor adaptive routing controller (EMARC) and the MDR onboard buffer (MOB). The EMARC is termed the satellite's 'mind', and controls all requests for service, timing of satellite functions, recovery from execution problems, and can be used to alter the programming to the MOP during operation. The MOB contains all of the data storage between the uplink and downlink portions of the satellite and handles all of the concerns related to the data paths. The MOP serves as the interface between the two divisions, receives the transmission security (TRANSEC) streams and consequently generates the current uplink and downlink permuted hop numbers, sets up the antenna assignments for the MOB, performs error correction encoding and decoding, generates the timing information for terminals acquiring access, buffers and manipulates data flow from the individual terminals and the EMARC, notifies the EMARC of any system errors, collects telemetry data, and performs many other individual tasks of data manipulation. The position of the processor in the MDR system is shown in figure 1.1.

The current system structure will be able to handle four streams of data (i.e. four demodulators) as a subset of the full Milstar 32-channel system. A complete system would require eight sets of the demodulators and eight data frame buffers. The single MDR processor and EMARC are designed to be easily expandable to the full system size.

The simulator testbed contains a low-data-rate payload as well as the medium-data-rate payload that interacts with the MOP. The MDR data may be multiplexed with the LDR data prior to being transmitted on the downlink channel. The multiplexing of the two data streams will be controlled completely by the LDR payload simulator. Terminals wishing to communicate on the MDR system are either pre-assigned acquisition slots on the MDR system or begin with an LDR link and then request an MDR acquisition slot. The acquisition slot then allows the user to become time synchronized for MDR and then request service access.

MDR SIGNAL PROCESSOR BLOCK DIAGRAM

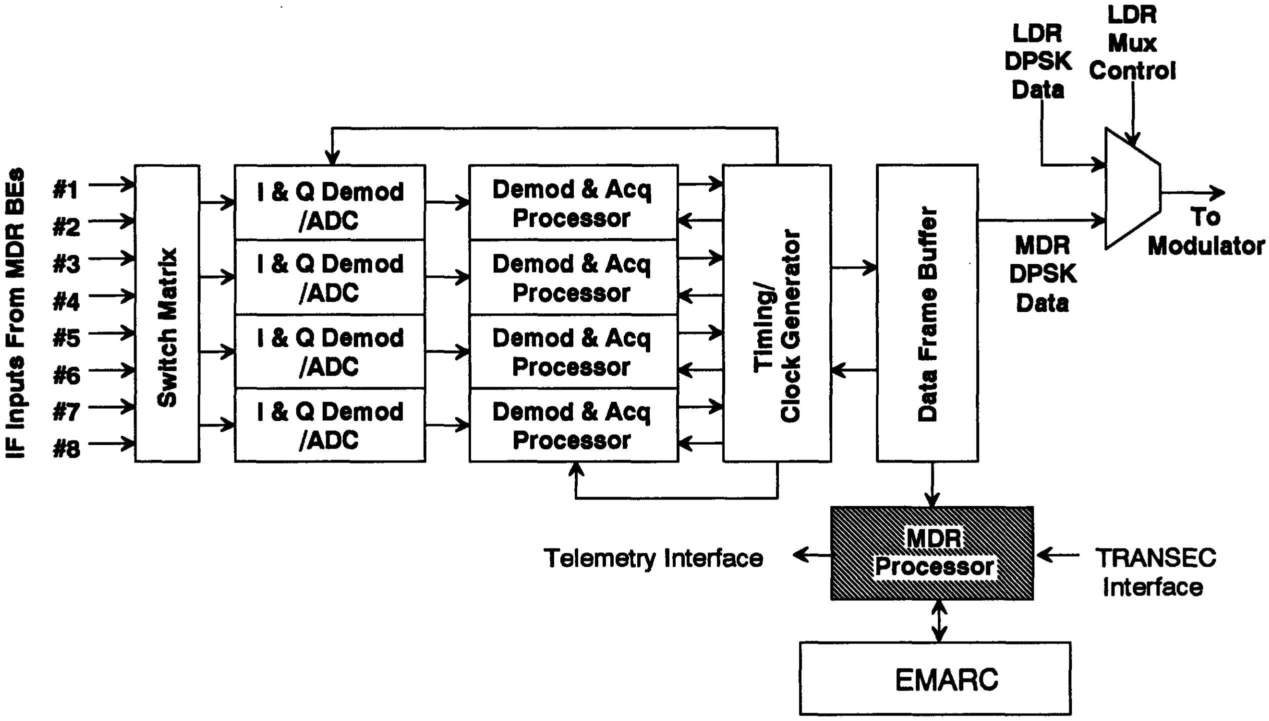


Figure 1.1: Overall MDR System Diagram

1.2 Processor Structure Overview

The processor consists of a micro-coded controller in connection with specialized sets of hardware to perform its tasks. All processor operations are performed along an eight-bit data bus that connects the system. A second eight-bit bus is reserved exclusively for loading the control words and is connected to the main bus through tri-state buffers to affect memory storage reads and writes.

The controller runs on a five-cycle instruction with four cycles used for retrieving the control word and one cycle for execution. The 32-bit control word is used to decode the bus source, strobe destination, jump control, ALU control, and memory data fields. The instruction word is pipelined once before implementation to allow maximum set-up time for all fields. All data is strobed into its destination at the end of the fifth cycle of the instruction time.

Most of the hardware design for this project will be implemented in programmable gate array (PGA) Xilinx chips, with the major exceptions of external memory and an external Viterbi decoder. The hardware designs are divided onto two Xilinx series 4010 chips with 5ns configurable logic block (CLB) delay. The job descriptions for the two 4010's can be briefly defined as a Interface/Specialized Hardware chip (Xilinx 1) and a Control/ALU chip (Xilinx 2) as shown in figure 1.2 below. The system clock and the hard reset signals will both be received onto Xilinx 2 and then sent, along with other timing information, to Xilinx 1. All interfaces are serial and operate according to several different protocols to be described later. The serial inputs all come directly into Xilinx 1 along with their required control lines. These are then converted into eight-bit parallel data and held in a first-in first-out buffer (FIFO). The outputs are held in a FIFO in Xilinx 1 until requested and then converted to serial data streams before transmission.

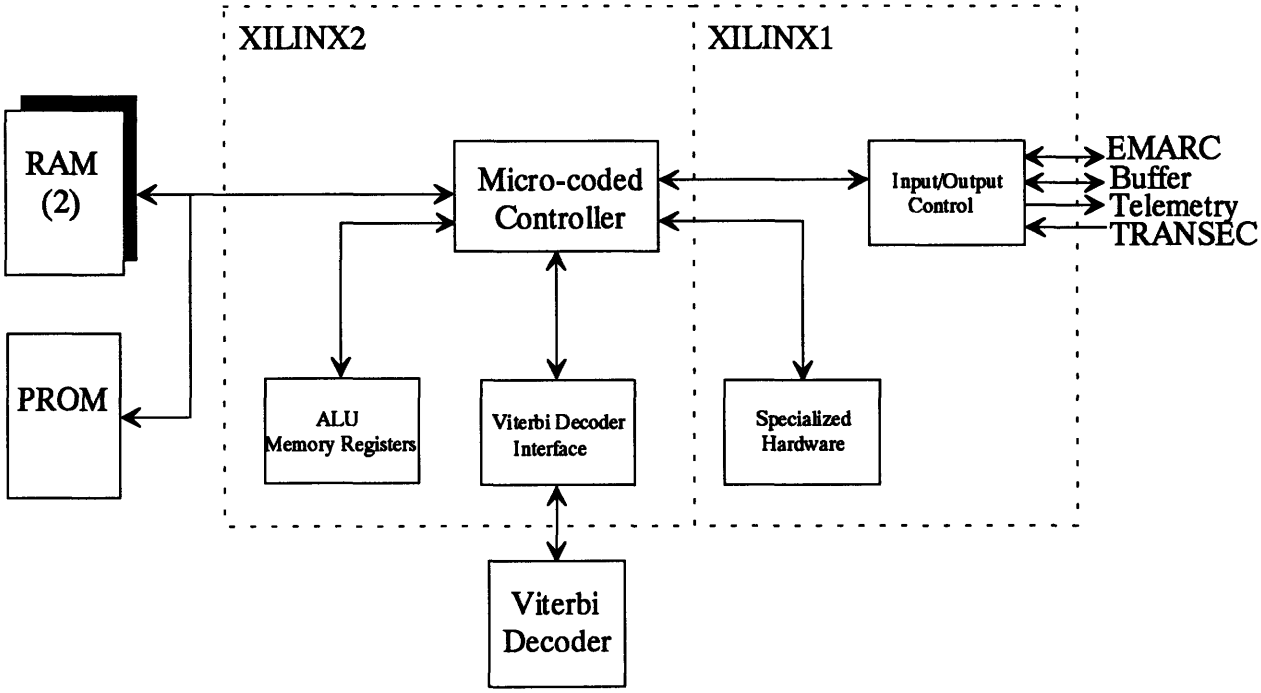


Figure 1.2: Division of processor duties among the Xilinx chips

The software for the micro-coded controller was designed using the METASTEP Language System, version 2.2. A general list of commands was created to serve the needs of this project. The commands include local (1-byte) and global (2-byte) jumps, immediate and local data loads, and multiple ALU and control statements. The commands were then compiled into 32-bit control statements and loaded into the onboard PROM.

The software for this system operates repeatedly over an **epoch** of time. Acquisition processing, ATOW (acquisition tracking and order wire), and MC3 (MDR Control 3) message generation and placement all require a full **epoch** before repetition. Other processor operations such as MC2 (MDR Control 2) manipulations and acquisition processing repeat every other **data frame** (of which there are a specified number within an epoch). Still other operations, such as servicing TRANSEC and updating time counts, must be performed once every **hop** (a time division within each frame).

To accommodate these different time schedules, the processor keeps a count of the current hop and frame count (mod epoch). The processor firmware operates without interrupts with three exceptions. One interrupt causes the processor to switch between processing long-term operations and servicing more immediate duties. This interrupt causes a forced jump to the HOP_MAIN routine with complete storage of system values. The other two interrupts occur when the processor misses a hop call and is automatically placed in a soft reset routine, or when a switch on the board is used to place the processor in a hard reset routine. The soft reset routine forces the firmware operations to the beginning of the operating program, while resetting all flags and counters. The hard reset routine reloads the on-chip RAM program from the PROM and then begins operations.

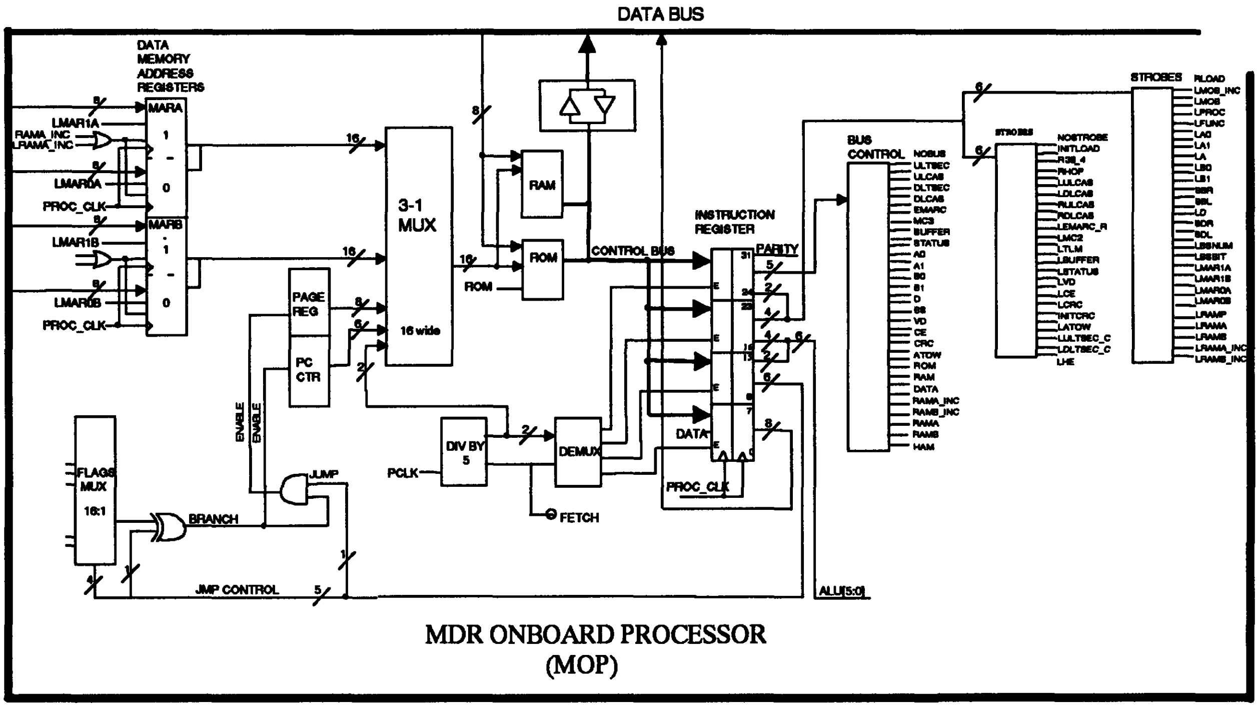


Figure 1.3: Processor control hardware

The diagram above shows an overview of the processor control hardware. The four-byte instruction register can be seen connected to the RAM and the PROM by the eight-bit control bus. The control bus is isolated from the main data bus by a set of bi-directional tri-states so that the instruction register can load a control word while the processor bus is transmitting other data. The control register has one cycle of delay so that as each operation is performed, the next instruction can be loaded. This results in lost operation time in the case of jump and branch instructions, but eases the timing constraints on the system by allowing a full five clock cycles to perform almost all operations. The only exceptions are RAM operations, where the read or write must be accomplished completely within the fifth cycle of the instruction. For all data transfers within the system hardware divisions, the bus source is released as soon as the current control word is activated at the beginning of an instruction cycle. The data is not strobed into the destination until the end of the fifth cycle (the execute cycle) to allow maximum time for transfer and set-up.

There are three sources for the RAM and PROM addressing in the MOP. The first is the program counter consisting of the page register, the program counter, and the byte count. The byte count is the two least significant bits of the address, and is generated by a divide-by-five counter to address the four bytes of a control word and then hold for the fifth cycle. Shown below is a simulation of the divide by five counter. Each of the first four cycles is used to enable data into the instruction register, and the execute cycle is used to enable the strobes generated by the previous instruction.

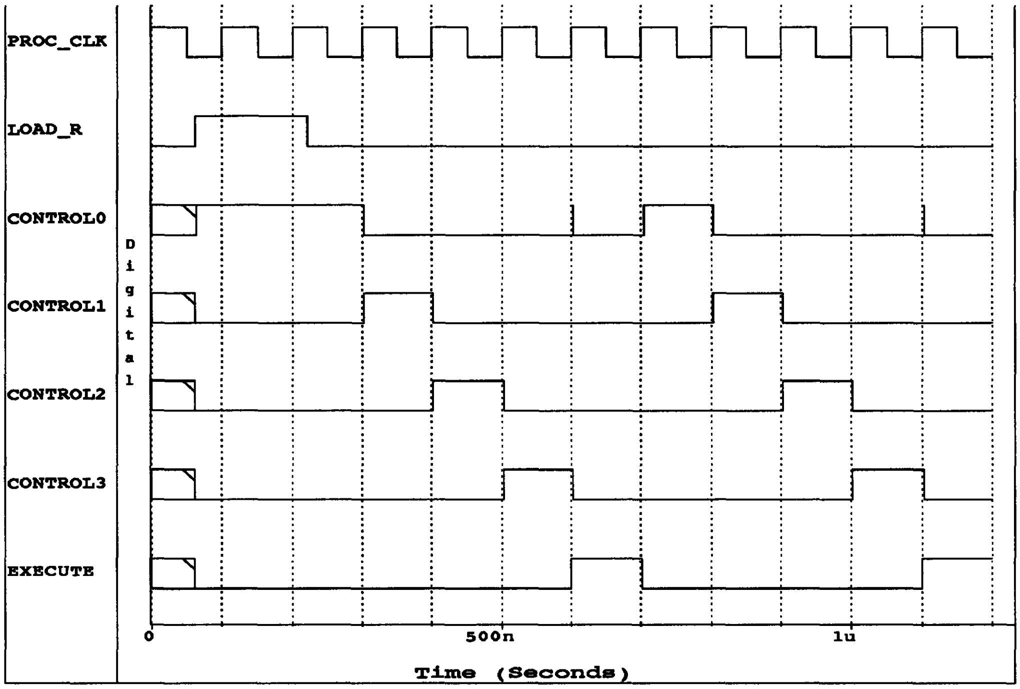


Figure 1.4: Instruction cycle control signals

The other two sources for RAM addressing are used to access the storage space in the RAM. They are used during the execute cycle only. By having two sets of RAM storage addressing it is possible to perform functions such as interleaving and deinterleaving much more efficiently. The three RAM addresses are multiplexed together and selected by the bus and strobe controls.

During a storage access to the RAM, the address must be changed during the execute cycle from the program address to the storage location address contained in memory address register A or B (MARxA or MARxB). The address is switched to storage as soon as the fifth cycle signal (execute) comes high. The control bus is connected to the data bus at the same time with the data direction being determined by the strobe and bus control values. If data needs to be read from the RAM (designated by RAMA, RAMB, RAMA_INC, or RAMB_INC as the bus sources) then the control bus is enabled onto the data bus. Conversely, if data is being written into the RAM (LRAMA, LRAMB, LRAMA_INC, or LRAMB_INC as the strobe control) then the data bus is enabled onto the control bus. At the end of the execute cycle, the data is then strobed either into the RAM or from the RAM and into one of the Xilinx registers. At this time, the RAM addressing is then changed back to the program control for the beginning of the next instruction cycle. The setup time of the CY7C256 RAM is 20ns, which is easily achieved within the processor clock period. The RAM also requires no hold time after a data write cycle. The danger involved in writing data to the RAM with no hold time is that the data bus could become unstable before the write enable signal has been removed. This timing concern has been analyzed and avoided.

1.2.1 Processor Interfaces

The processor interfaces with the EMARC, the MOB, the TRANSEC streams, the telemetry system, and the timing generator board of the satellite.

There are four EMARC interfaces with the onboard processor. Two of these interfaces provide system control over the processor and buffer memory. Through the control and response interfaces, the EMARC has complete control over the memory and operations of the processor. These interfaces may also be used to read and alter the buffer memory.

The other two EMARC interfaces are used for assigning system access. The first interface, from the processor to the EMARC, is used to transmit MC2 messages (requests for system access) from ground terminals. The second interface, from the EMARC to the processor, is used for returning MC3 messages (system access assignments) through the buffer memory to the ground terminals.

An extensive set of interfaces also exists for communication between the processor and the frame buffer cards. The entire interface is controlled by the processor. It can be used to read and write data into the buffer memory and control sections, set uplink and downlink buffer and hop numbers, set frame and epoch sync flags, and update beam numbers for MC3, ATOW, and synchronization messages. By reading from the buffer memory, the processor can retrieve acquisition data, MC2 messages, and DC offset data. A write to the control section of the buffer is used for setting up the uplink and downlink control words that are received from the EMARC as well as placing the prepared ATOW and MC3 messages for transmission.

The uplink and downlink TRANSEC data streams are one-way communication systems. The data received by the processor from these interfaces is used to generate the uplink and downlink permuted hop numbers for the data frame buffer. The processing performed by the MOP upon the two TRANSEC streams must be performed once every hop, without exception. The processor is designed to automatically jump to a reset state if a hop passes without service.

The telemetry interface will be used for testing purposes and periodic satellite health checks. Any data may be transmitted on the telemetry interface, although common examples will probably include demodulator DC offset values and accumulated acquisition results. The processor will retrieve data and prepare it for transmission on the telemetry interface at EMARC command.

The timing generator board interfaces consists of two one-way serial data lines. Both of these are clock signals required for MOP operation. The first is the processor clock which is the timing used for all firmware instructions. The second signal is a hop clock that is used to synchronize the MOP processes which must be performed on a per-hop basis.

The processor interface hardware is shown in figure 1.5.

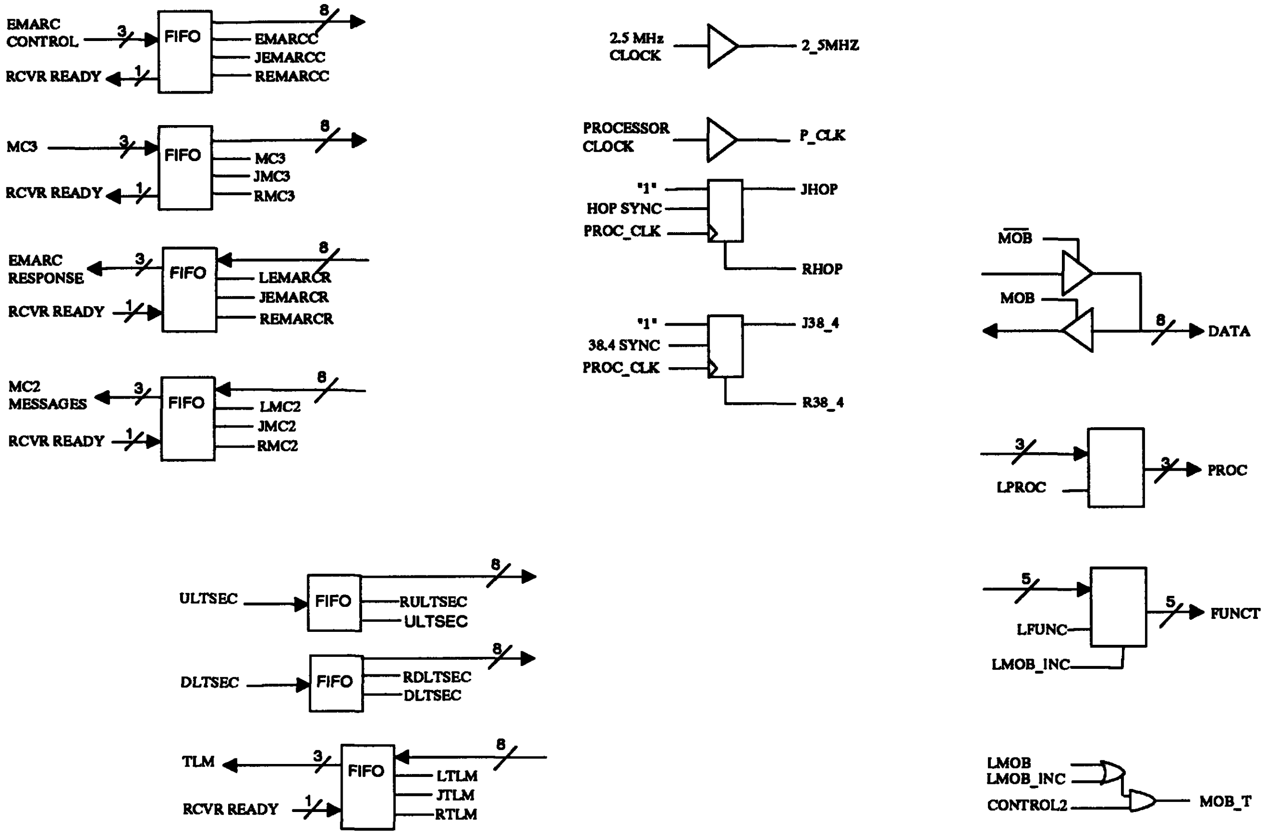


Figure 1.5: Input/Output Block Diagram

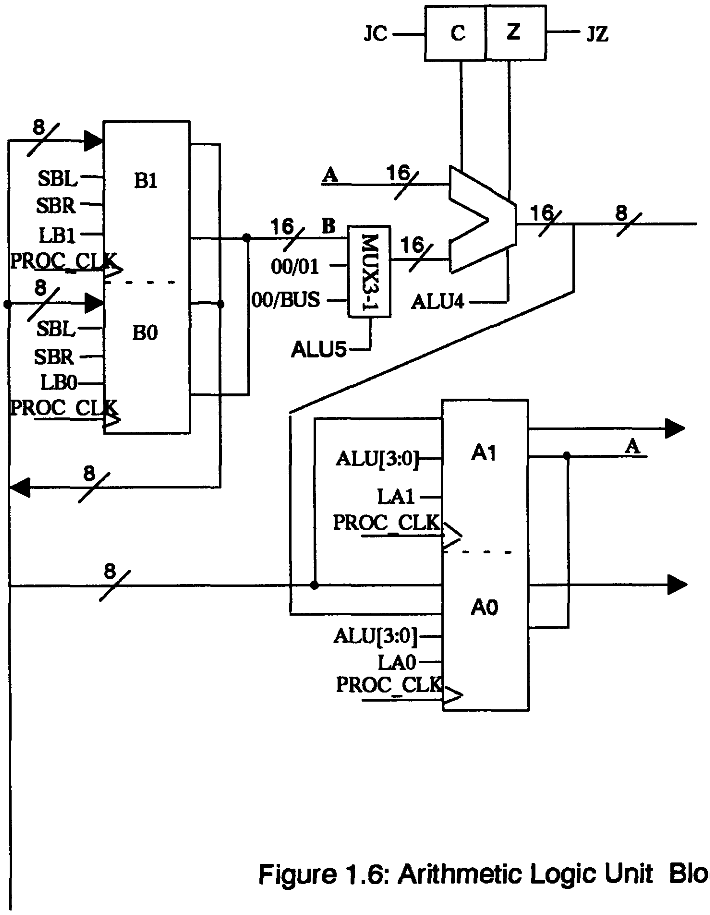
1.2.2 ALU/Data Storage Registers

The processor uses a 16-bit adder/subtractor in conjunction with six data registers to perform various tasks. There are five registers that may be used for temporary data storage. These are the A1, A0, B1, B0, and D registers. The A and B registers used as pairs provide the inputs to the 16-bit ALU. The A registers can also be added or subtracted with a h'00' in the MSByte and either the bus data or h'01' in the LSByte. The set of A registers also generate zero and carry flags. The structure of the ALU and the supporting storage registers is detailed in figure 1.6.

The A0 register alone can be used to compute some eight-bit logic functions. The functions of 'and', 'or', and 'xor' can be performed with the data from A0 and either the eight-bit bus or the B0 register. These operations are performed within one instruction cycle and the results are strobed back into the A0 register. These functions are indispensable to many of the normal processor operations.

The B registers are paired shift registers as well as individual storage units. This allows multiplication or division of the 16-bit contents by powers of two. This multiplication process is used to streamline the downlink time-permuting algorithm.

The MC3 and MC2 messages require block interleaving and deinterleaving, respectively, which is done with the use of the D storage register in conjunction with some specialized bit shifting hardware. The interleaving performed in this system is done on a bit-by-bit basis and requires a high percentage of the available processor cycles.



ALU FUNCTIONS

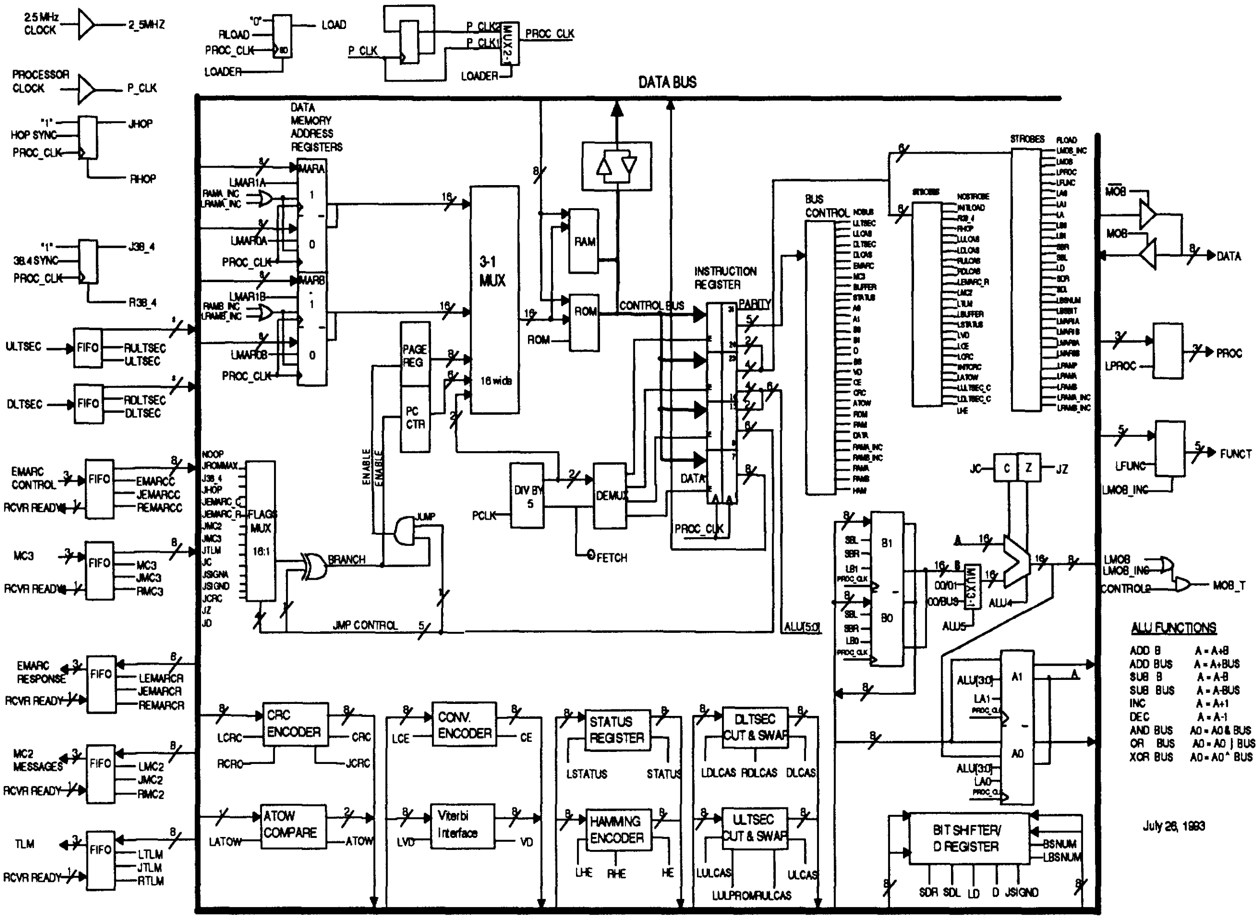
ADD B	A = A+B
ADD BUS	A = A+BUS
SUB B	A = A-B
SUB BUS	A = A-BUS
INC	A = A+1
DEC	A = A-1
AND BUS	A0 = A0 & BUS
OR BUS	A0 = A0 | BUS
XOR BUS	A0 = A0 ^ BUS

Figure 1.6: Arithmetic Logic Unit Block Diagram

1.2.3 Specialized Hardware Units

There are eight hardware segments programmed in Xilinx 1 for performing other duties. These hardware blocks are used for encoding (cyclic redundancy, Hamming, and convolutional codes), decoding, TRANSEC functions, and 2-bit ATOW message generations. The specifics behind these requirements will be discussed in detail in the chapter on hardware design.

Figure 1.7 shows the overall block diagram of the micro-coded controller. All of the segments just discussed can be easily seen, as well as the overall bus structure of the design. Because of the large number of hardware blocks that interact with the bus, overloading and long set-up times for the bus is an obvious concern. Through the addition of the pipeline in the control word, we have allowed approximately five times the original period for data movement along the bus to relax this time constraint.



MDR ONBOARD PROCESSOR
(MOP)

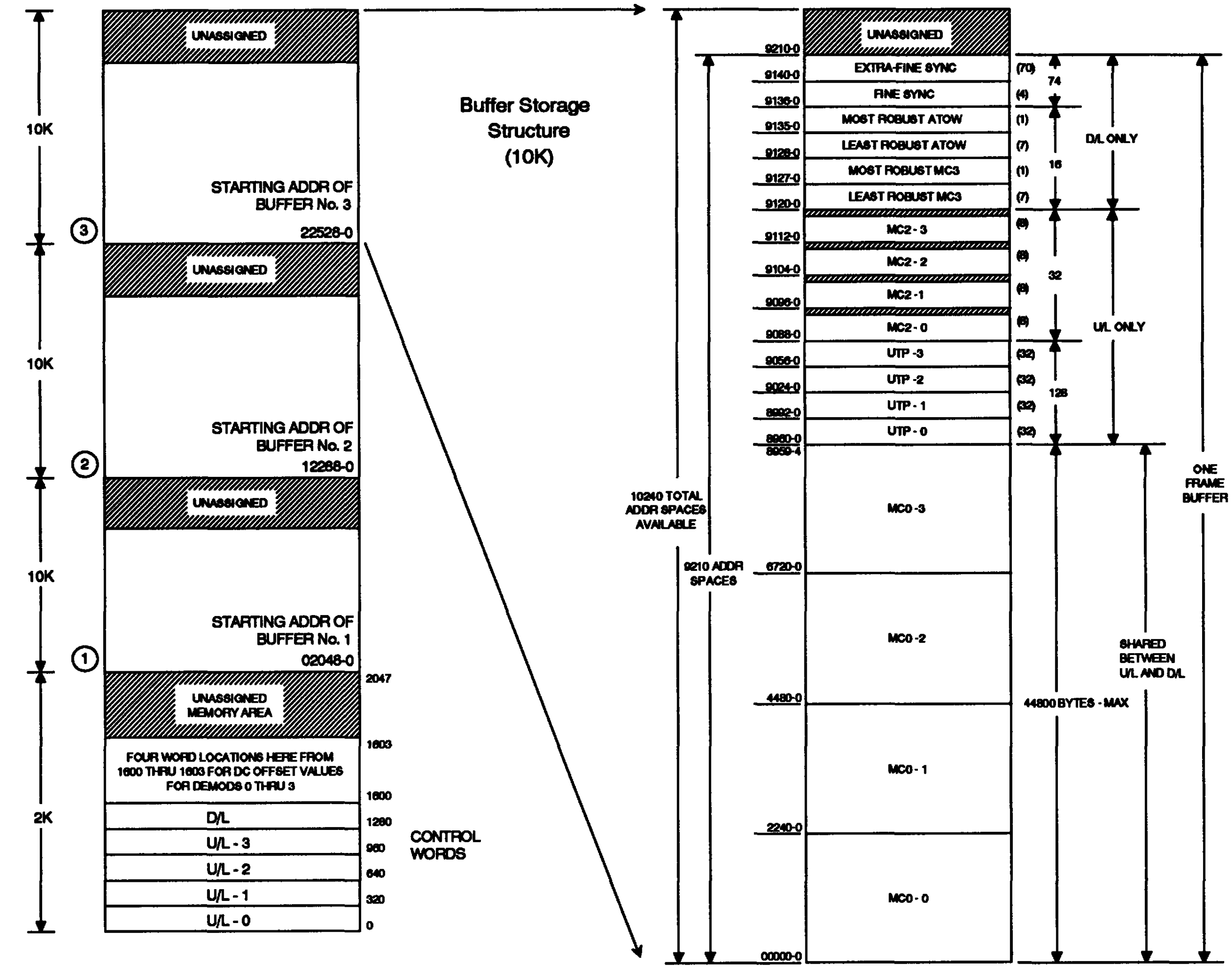
Figure 1.7: Total processor block diagram

1.3 Buffer Memory Structure

The MDR onboard buffer stores all of the data from the uplink demodulators and handles downlink transmission. The buffer memory consists of three identical segments, each of which can contain an entire frame's worth of communication. Three segments of memory are required because of a six hop lag between the uplink and downlink on the satellite. Therefore, the buffer completes filling one frame from the uplink before the current downlink frame buffer is completely transmitted. Due to different permutation of the hops on both links, there is no guarantee that the last six hops in an uplink frame would not be placed over any of the last six hops in the previous downlink frame.

The buffer memory has a parallel structure in which a 16-bit address locates a five-byte word. Whenever addressing the buffer memory it is necessary to specify not only the word address, but a three-bit byte address as well.

There are memory locations at the bottom of the three buffer frames for storage of uplink and downlink control words. For each hop, the buffer retrieves the control words corresponding to the two permuted hop numbers (one each for uplink and downlink). The control words are used to determine the beginning memory locations, data rates, and beam numbers for both the uplink and downlink. The locations of the MC2 and uplink time probes (UTP) for an uplink data frame are completely defined and may be seen in figure 1.8. Equivalently, the positions for both fine and extra fine downlink sync sequences as well as MC3 and ATOW messages may be seen for a downlink data frame.



MDR MEMORY MANAGEMENT MAP - FRAME BUFFER

COMBINED U/L AND D/L MEMORY PARTITIONING

Figure 1.8: Buffer Memory Structure

Chapter 2

System Functions

2.1 Acquisition Overview

To review the system operations for the satellite and specifically how these effect the duties of the processor, we should begin with the ground terminal acquiring time synchronization with the satellite. As the system being designed contains both a low and a medium data rate payload a ground terminal wishing to communicate at MDR has two options for acquiring access. The ground terminal may either first acquire timing on the LDR system and use this to gain MDR access, or it may use a pre-assigned MDR acquisition slot and directly acquire timing on the MDR system.

As can be seen in the following diagram, the first step in gaining time synchronization is through the downlink SYNC sequences from the satellite. On the MDR system there are both fine and extra fine downlink sequences. The ground terminal listens at an approximate time for these sequences and adjusts its timing accordingly; first using the fine, and then the extra-fine sequences. After acquiring downlink synchronization, the terminal is able to use its assigned acquisition slot to send uplink time probes to achieve more accurate timing. Once again, there are both fine and extra-fine versions of these sequences. Only users that plan to transmit at the highest available data rates are required to go through extra-fine uplink time synchronization.

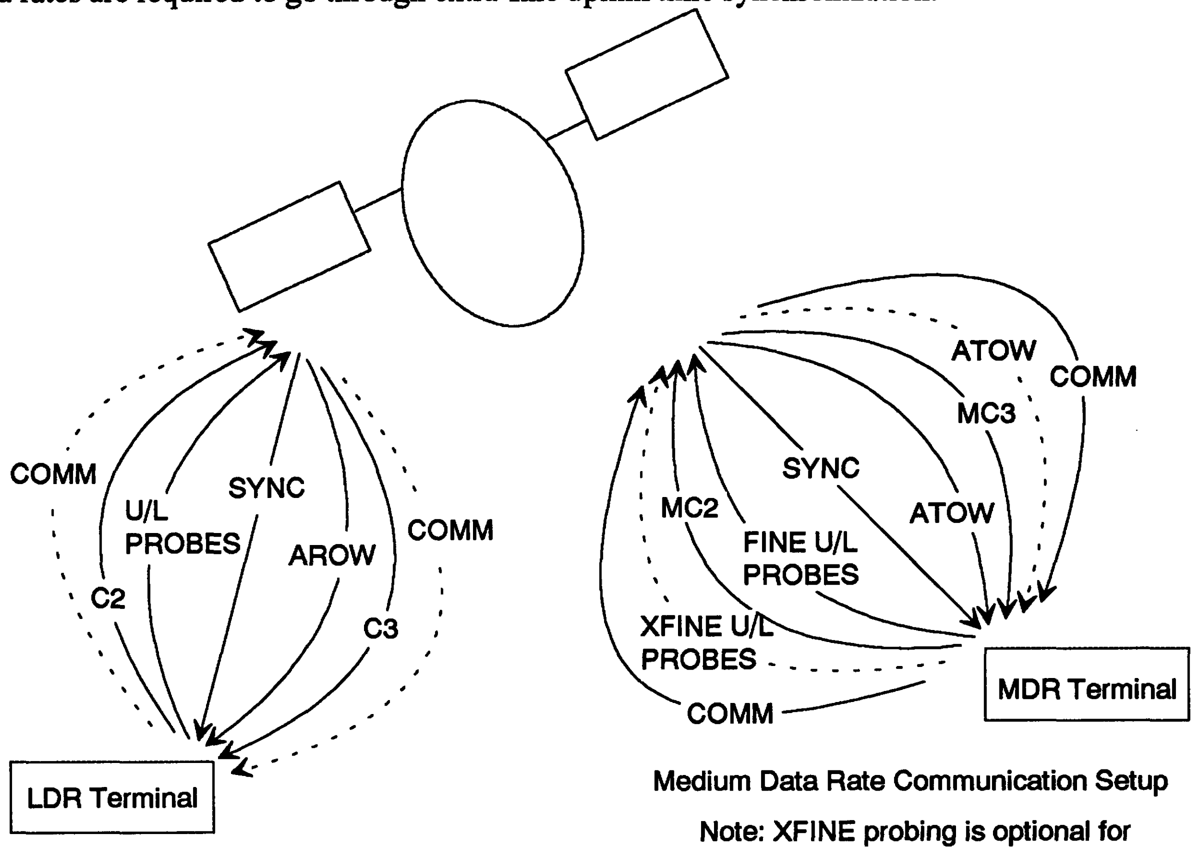


Figure 2.1: Satellite Signaling Procedures

To acquire fine uplink time synchronization, the user transmits a predetermined 18-bit sequence. The binary sequence chosen for this purpose was found by exhaustive search for maximum autocorrelation when precisely aligned and rapidly decreasing correlation values when compared with time shifted versions.

The terminal transmits the fine acquisition sequence during its assigned acquisition slots until synchronization is achieved. Each terminal during acquisition transmits eight times during one frame for two consecutive frames and then receives a two-bit response from the satellite indicating detection and the early/late result. The terminal will then shift its timing in the indicated direction and transmit again in the next set of assigned acquisition frames. The response from the satellite never indicates an 'on-time' result, therefore the terminal keeps trying to acquire until the response from the satellite changes. For example, if the terminal has been receiving multiple 'early' responses, it will assume it is very close to on time as soon as it receives a 'late' response.

After achieving fine synchronization, the terminal is able to transmit MC2 messages requesting communication access on the system. While awaiting an MC3 response granting system hops, the terminal may use its acquisition slots to achieve extra-fine synchronization.

To acquire extra-fine uplink time synchronization, the user transmits a predetermined 168-bit sequence. The binary sequence chosen for this purpose consists of a mac sequence of 79 bits in length transmitted on both the I and Q data streams. The sequences are shifted by 24 bits with respect to each other, and the ends of the message (168 - 2*79 = 10 bits) were chosen by exhaustive search to maximize autocorrelation and minimize off time peaks.

When the transmitted sequence is received at the satellite demodulators, it is compared with four shifted versions (two early, two late) of the fine sequence, and two versions (one early, one late) of the extra-fine sequence. Correlations for both fine and extra-fine acquisition are made as the demodulator has no way of knowing which level of synchronization is being attempted.

Each fine correlation value is 16 bits in magnitude and is packed into four consecutive words plus one byte in the buffer memory. The extra-fine correlation results have a magnitude of only 12-bits. Both 12-bit values are sent to the buffer memory where they are placed in the remaining four bytes of the last word used by the fine correlation results. These correlation values are held passed to the buffer memory for temporary storage. At defined times, this data will be removed by the processor for analysis.

The correlation values for sixteen transmissions from one terminal are therefore transmitted to the buffer memory during two consecutive data frames. This raw data is taken from the buffer memory by the processor after each set of eight correlations has been received. The MOP then processes this data and generates a two-bit response to return in the form of an ATOW message to the terminals so that they may accordingly adjust their timing. After many iterations of this process for fine synchronization, the terminal may repeat the same process for extra-fine synchronization. This is only required for users that wish to transmit at the highest available data rates.

2.2 Processor Acquisition Duties

Every data frame, the buffer memory receives eight hops of correlation data from each of the demodulator boards (currently four). All of this data is grouped together in a section of the buffer memory for storage. Unfortunately, because of pipeline delay, the last hop of acquisition for a frame arrives at the buffer during the first acquisition hop of the next frame. This results from demodulator delays and means that the processor can not begin to accumulate the data from an acquisition frame until almost the end of the following frame. Acquisition frame pairs always begin on the even frame and end on the odd frame, so the last data of an acquisition pair is received towards the end of the following even frame. The processor divides the work load over two frames by taking the first half of the data during the end of the second frame and beginning processing. The second half of the data is then taken during the third frame and processing the data into a final two-bit code is completed.

Processing the correlation values is done in three steps. The first step is to accumulate all 16 hops of the 6 sets (four fine, two extra-fine) of correlation data. Minimum correlation thresholds (potentially different for each channel) are then subtracted from all 6 results. Any positive results from the threshold subtraction are then stored again and comparisons are done between them to generate the final two-bit response. The two-bit binary output codes are defined as:

00: No correlations were above threshold
01: The terminal timing is early and needs to be delayed
(Either for the fine or extra-fine sequences and the early correlation is further above threshold than the late correlation)
10: The terminal timing is late and needs to be forwarded
11: Both fine and extra fine correlations were above threshold

These two-bit codes are generated for all users and then a number of these responses are packed into one message for transmission. These response messages are CRC encoded, convolutionally encoded, and then interleaved prior to placement in the buffer memory. The time schedule for preparing particular messages for transfer is based upon the particular probe block and time slot that a user was assigned for acquisition. The downlink messages can also be transmitted in either least robust or most robust mode (as can downlink MC3s). Both of these topics will be covered in detail in later sections.

2.3 MC2 Messages

Once a terminal has acquired satellite timing, it is able to request service hops from the EMARC through the use of MC2 messages. These MC2 messages may correspond to users transmitting in either the least or most robust mode. This does not alter the transmission rate of the MC2 message, but does effect the time slot which the MC2 message occupies. There are four MC2 message time slots per frame (one per channel for four channels in the current system) and these are stored in dedicated positions in the buffer memory until retrieved by the MOP.

MC2 messages are only received at the buffer during even frames. However, because of the time permutation of the uplink streams, the MC2 messages can arrive at the MOB at any point during an even frame. Therefore, the processor always operates on MC2 messages during the following odd frame. Since the buffer uses a rotating system of three frame memory spaces, there is no concern of the MC2 messages being overwritten before they are retrieved. The MOP retrieves one MC2 message at a time, processes it completely, and then stores the remaining message with the generated header bytes in a queue to await EMARC service.

The format of the MC2 messages is shown in the following diagram. When the message is received it requires deinterleaving, Viterbi decoding, and authentication prior to storage. The deinterleaving performed by the processor is accomplished over a series of hops by using the bit shifter and D register. After deinterleaving the message, the MOP provides the interface for a Qualcomm 1601 Viterbi decoder. All of the data has been hard decision demodulated prior to this point and due to the structure of the data, there is no concern of the data getting out of sync. The bit error rate (BER) tester that the 1601 provides is also not useful for this implementation as the minimum number of symbols that must be decoded prior to receiving a BER value is too large for the purposes considered here. (Multiple MC2 messages would need to be decoded to receive an accumulated BER, and it is likely that many MC2 messages sent through the decoder will be simply noise.) This would yield a high BER value that would be meaningless for most MC2 messages. After the Viterbi decoding, the message is authenticated by a CRC tail check so if a message was not completely error free after the Viterbi decoder, it will not be sent on to the EMARC. The remaining passing messages are then stored in the EMARC queue. The header information for each message is calculated by the MOP from memory tables set up by EMARC control.

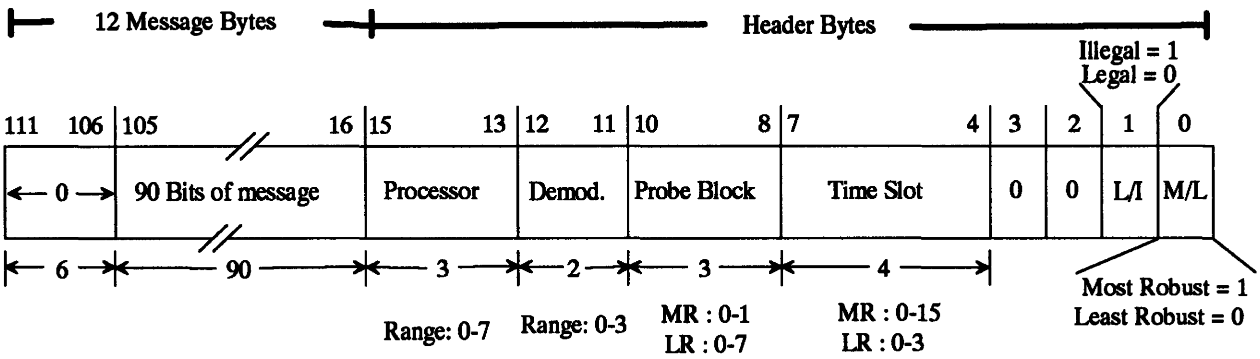


Figure 2.2: MC2 Message Format

During each hop, the MOP steps through a priority list of operations before performing long-term operations. The items on the priority list in order are:

1. Servicing an EMARC control message
2. Service the MC3 interface
3. Service the MC2 interface
4. Complete any MOP memory reads
5. Complete any MOB memory reads

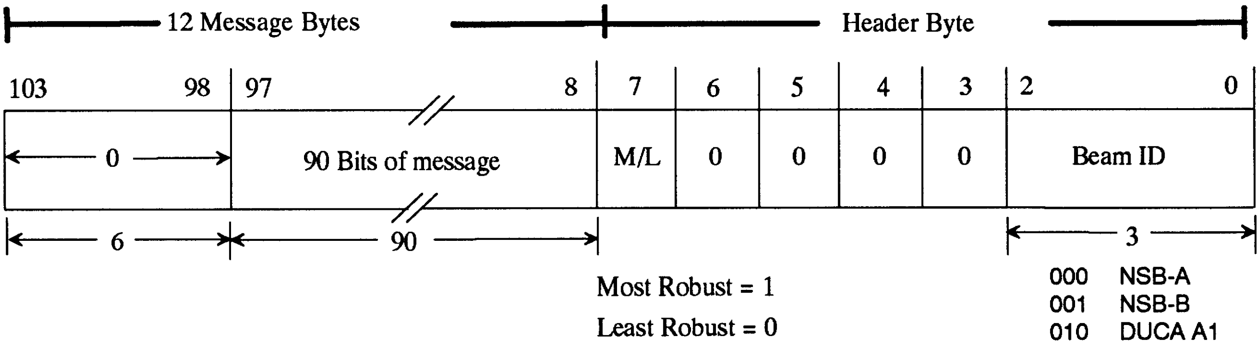
Only one of these operations will be performed during one hop. Potentially, none of these operations will be required and the MOP will proceed directly to the long-term operations.

2.4 MC3 Messages

After a terminal has acquired satellite timing and issued an MC2 request, it waits for an MC3 response. The MC3 messages are sent by the EMARC to ground terminals to acknowledge logon and inform users of their hop and channel assignments. The MC3 responses will be sent from the EMARC to the processor including the antenna designation and robustness level of the message. The processor then prepares the message and places it in the buffer memory for downlink transmission.

The processor operations for an MC3 message are almost identical to those performed to an MC2 message although they are performed in reverse order. The initial reception of an MC3 message is part of the priority list shown above. On average, the processor will not receive more than two MC3 messages during one data frame. After taking the MC3 message from the EMARC interface, the data manipulations performed on it are scheduled with the other long-term operations.

The MC3 messages received from the EMARC contain one byte of header information along with 90 bits of unencoded data. The processor stores the header byte and then adds a CRC tail to the data. Decoder flush bits are also added to the message at this point for the next step of convolutional encoding. The next step is to then interleave the message and place it in storage until the correct time to place it in the buffer memory. The format of a received MC3 message is shown below.



000	NSB-A
001	NSB-B
010	DUCA A1
011	DUCA A2
100	DUCA A3
101	DUCA B1
110	DUCA B2
111	DUCA B3

Figure 2.3: MC3 Message Format

2.5 Robustness levels

The concepts of most and least robust messages have been mentioned with respect to all of the control messages on the satellite. The MC3 and ATOW messages can be transmitted in one of these two modes. In the least robust mode, all of a message (of whichever format) is transmitted during one hop. In the most robust mode, 16 hops are required for the full message transmission.

With respect to the MC2 uplink message, the mode does not effect the transmission rate of the message. The header information added onto the MC2 message is taken from a preset table based strictly upon the frame number and the channel (demodulator) on which the message was received.

When considering the downlink messages (MC3 and ATOW), the robustness level does affect the operation of the MOP. In this mode the processor must not only remember the beginning frame of the transmission, but must break the message up into two-hop segments and send them to the buffer memory during each of the next eight data frames. The scheduling for message transmission and a more detailed definition of most and least robust designations will be given in chapter 5.

2.6 Coding Background

The Milstar satellites are designed to be able to operate with great certainty even in the presence of jammers or other interference. To this end, many steps were taken to protect the satellite communications. These include frequency hopping and time permutation of both uplink and downlink. Error correction is also used for all communications.

The actual data communication between users is encoded and interleaved on an end-to-end basis, but the satellite itself is passive with regard to their coding schemes. Data communication is brought into the satellite in a time permuted fashion and hard decision modulated to the bit. The data is then retransmitted according to a different permutation scheme but no error correction is done to the data on the satellite.

The most important aspects of the satellite communication system are the control messages. These are protected by nested encoding schemes, permutation with the data, and interleaving. The control and acquisition messages to and from the satellite are encoded twice and interleaved to aid in error correction. The data interleaving spreads bursts of errors to aid in decoding. The Viterbi decoder performs best when given a data stream with approximately random error patterns. If a jammer is trying to disturb satellite communication, most likely he will destroy a stream of consecutive data bits. This type of errors are difficult if not impossible for the Viterbi decoder to correct. If the data is interleaved prior to transmission and then deinterleaved at the satellite before decoding, the burst of transmission errors will be spread across the entire message. This will cause the errors to appear random.

This process is typically accomplished by block interleavers for control messages. In this case, the data stream is read into a block of memory in one direction and removed at right angles. This process is then reversed on the satellite to regenerate the original message before decoding.

The first decoding then performed is a Viterbi decoding algorithm for removing a convolutional code. Convolutional codes are a type of continuous code that are generally easy to implement but difficult to decode. After the Q1601 Viterbi decoder has removed the convolutional code, a CRC tail is checked for authentication. This is accomplished by reencoding the data portion of the message and comparing the generated tail with what was received. If the CRC tail does not check, the message is ignored. This is done to prevent random noise on an unused channel from being sent to the EMARC as a valid message. Most of the MC2 time slots on the uplink will be unused and this step prevents the EMARC from having to analyze many invalid messages.

The processor software and buffer control memory are also encoded to prevent single-event upsets from causing erroneous operations. The processor software has one bit of parity appended to each of its instruction words. Even parity is confirmed before each command is performed. If the parity is wrong (odd), the instruction is abandoned and an error message is sent to the EMARC. If the error is in part of a jump instruction, the entire command is aborted. The MOP continues to operate, possibly wrongly, but the chances of causing a fatal error by ignoring one instruction are less than the chances of the code becoming lost if the processor jumps to an invalid memory location.

The buffer control memory has an eight bit pseudo-Hamming code added to each of its 32-bit control words. The control words are setup by the EMARC via the processor and determine the operation for each uplink and downlink hop. The control words provide the starting data locations, data modes, chip rates, and beam number (downlink only) for each hop.

The Hamming tail appended to each control word is used completely for detection of errors, no correction is attempted. Seven bits of the tail are chosen for maximum distance and the final bit is an overall (even) parity check. Prior to executing a hop control, the buffer checks the one-bit parity and aborts the hop operation if an error is found. The buffer then notifies the processor of the error and does not transmit or store any data during that particular hop. The processor relays this information to the EMARC so that the control memory can be replaced.

Periodically, the MOP picks consecutive control words out of the buffer memory and performs a more extensive check for errors. If one is found the processor clears the control word and notifies the EMARC. This process will interrupt one communication link until the EMARC fixes the problem, but all other communication links will continue to operate normally. The error-detecting ability for the control words is much greater than the one-bit check performed by the buffer. The eight bits of encoding on a 32-bit control word provides greater detection than the three bit detection of a ideal Hamming code.

Ideal Hamming codes are generated with data in the form of :

$$n=2^m-1 \qquad k = 2^m-1-m$$

where the first term is the final encoded length and the second is the number of unencoded data bits. Codes that fit these parameters are perfect three error detecting codes. In the case described above, the code is a (40,32) code. The closest ideal Hamming codes to this are the (31,26) and the (63,57) codes. Therefore, a 32-bit sequence requires between five and six check bits to provide complete three bit detection.

For our byte-oriented system, we are provided with eight. By careful selection of parity check matrix columns, we are able to detect all five bit error combinations and many higher combinations as well. Figure 2.4 below shows the algorithm used for the generation of each parity bit in the check byte.

Parity Bit # Message Bit #	0	1	2	3	4	5	6	7
0	1	1		1	1			1
1	1	1		1		1		1
2	1			1	1			1
3	1	1		1				1
4	1	1	1	1				1
5		1	1					1
6	1		1		1			1
7	1		1			1		1
8	1		1				1	1
9	1	1	1		1			1
10		1		1	1			1
11		1		1		1		1
12		1		1			1	1
13				1	1			1
14		1		1				1
15	1	1			1			1
16	1	1		1				1
17	1	1					1	1
18	1	1						1
19	1	1	1					1
20	1			1				1
21	1		1					1
22	1					1		1
23	1						1	1
24	1							1
25		1						1
26			1					1
27				1				1
28					1			1
29						1		1
30							1	1
31								1

Figure 2.4: Hamming Encoder Algorithm

Chapter 3

Interface Protocols

The MOP interfaces with the EMARC, the MOB, the TRANSEC streams, the telemetry system, and the timing generator board of the satellite. Each of these interfaces operates according to a unique one- or two-way protocol.

3.1 EMARC Interfaces

All four interfaces connecting the EMARC to the MOP use the same protocol for operation. The handshaking for each interface includes two control lines, a data clock, and a serial data line. One of the control lines originates with the data receiver to indicate the ability to accept data. The other control line is a data enable from the transmitter. The timing requirements for the interface are shown in diagram 3.1.

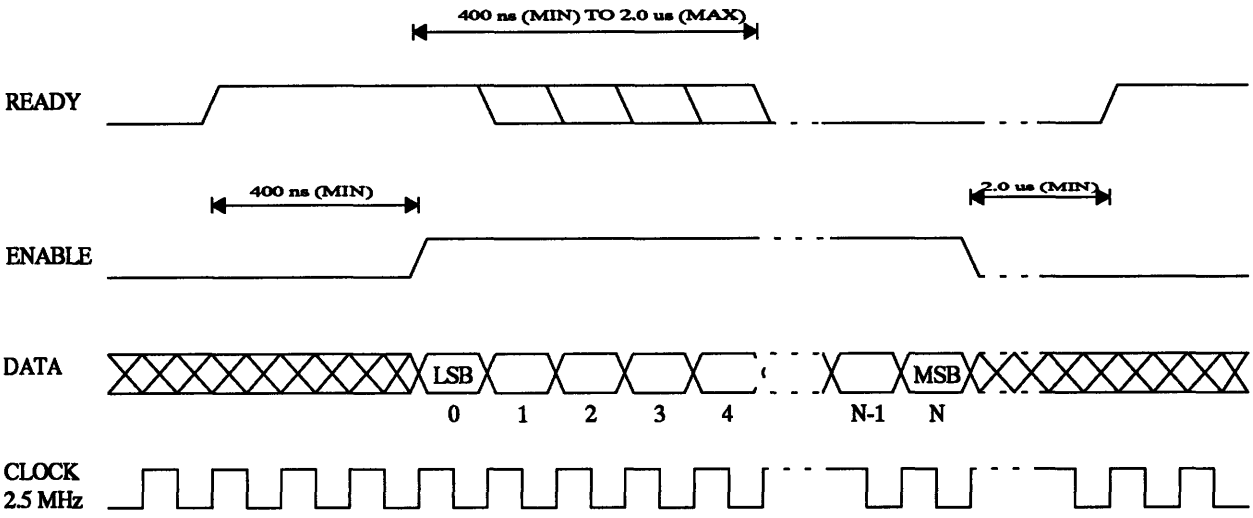


Figure 3.1: Serial Interface Timing

Two interfaces are used for transmission of data from the EMARC to the MOP. The 'ready' line from the MOP to the EMARC consists of an empty flag from the corresponding first-in first-out buffer (FIFO) indicating that the MOP has serviced the previous EMARC command on that interface. The EMARC is then able to use this interface at any time to transmit another message. Converting the data to a parallel format and storing it into a FIFO occurs without direct MOP control. No processing time or instructions are devoted to any of the input interfaces until the MOP is ready to operate on the received data.

On an input interface, the processor only supplies a ready flag to the transmitter. The ready flag has a region of required hold time that is met by the processor by removing the ready flag after one byte of data has been received. The ready flag is replaced by reading all of the data out of the storage FIFO and resetting the input and output addresses. The processor receives the serial data stream on the falling edge of the incoming 2.5 MHz clock, puts the data through a serial-to-parallel converter, and stores it in a FIFO until needed.

The first interface from the EMARC to the MOP is a control interface. Through this interface, the EMARC can control or alter both the processor and the frame buffer. These messages are a constant 72 bits in length, and can have any of 20 distinct formats. By sending a command, the EMARC can control the memory contents of the frame buffer and both the memory and operations of the MOP. This interface is serviced by the MOP during each hop when possible, but with a priority below that of acquisition and MC2 processing.

The second EMARC to MOP interface, an incoming message is termed an MC3 message. This is data to be transmitted from the EMARC to a particular system user. The MC3 message format includes one header byte along with 90 bits of unencoded information. The header byte is generated by the processor and contains the beam ID number and the mode of the transmission: least or most robust.

There are two interfaces for transmission of data from the MOP to the EMARC. The processor places data into one of these two buffers and it is held there until the 'ready' line from the EMARC is active. The data enable line from the MOP is equivalent to a full flag on the storage buffer indicating that an entire message is prepared. Once the 'ready' line is noted, the stored data is converted to a serial format and transmitted along with a synchronized clock. Once the data is stored into the FIFO, the rest of the operation does not require processor instruction cycles to be performed.

For an output interface, the processor supplies the serial data stream, data enable, and the 2.5 MHz clock to the receiver. The data is clocked out of the eight bit shift register on the rising edge of the 2.5 MHz clock that is selected by the jumper connector on the board. The enable line remains high until the beginning of the cycle after the last bit of data has been transmitted. A simulation of the EMARC response interface is included at the end of the attached hardware packet.

The first interface from the processor to the EMARC is the control response interface. Through this interface the MOP can respond to any control requests the EMARC has made and can inform the EMARC of any system errors that have occurred.

The format for these messages varies with the response being transmitted, but all messages are 72 bits in length. (See appendix A for the formats for EMARC control messages and responses.)

The second MOP to EMARC interface is used exclusively for MC2 messages, a reciprocal of the MC3 messages discussed earlier. These messages are transmitted from a ground terminal to the EMARC. These messages usually contain requests for service. The MC2 message format when transmitted to the EMARC includes two header bytes in addition to 90 bits of decoded information received from the ground terminal. The header bytes are generated by the MOP and contain the receiving processor and demodulator numbers, the uplink probe block number and time slot numbers, one bit to indicate if this message was received during a legal message time, and one bit to describe the mode of transmission.

The handshaking interface between the EMARC and the MOP guarantees that both reading and writing will not be done at the same time. It is not necessary that the interface have this ability, and the FIFO has been streamlined by preventing this option. The processor does not raise the ready line on an input interface until after all of the previous message has been removed from the FIFO. Also, the processor does not recognize that there is any data in the FIFO until an entire message has been received (FIFO full flag). On the output side, the processor will not raise an enable until the entire message is stored in the FIFO and will not try to store another message until the FIFO has been emptied by the EMARC.

3.2 Buffer Interfaces

The MOB interface consists of one bi-directional bus interface and three one-way control interfaces directed by the processor. The first control interface (processor select) is used to select one of the eight possible frame buffer boards for communication. The second control interface (function select) uses 5-bits of data to describe the type of information being sent to the buffer. The third control line is used by the processor to determine the data direction on the interface bus (MOB). Lastly, there is a one-line signal from the MOP that instructs the buffer to store the data currently on the interface bus (MOB_T). A hardware diagram of the interface and a simulation of the operations are included at the end of the hardware packet.

Both the processor select and function select interfaces are registers on the processor Xilinx 1 that supply consistent data to the buffer memory. The registers can be parallel loaded by the processor data bus at the end of the fifth (execute) cycle similar to all processor registers. The function select register also has the ability to be incremented automatically at the end of a function operation. The commands defined on this interface (Appendix B) have been ordered in such a way that the progress from one to the next is likely. The interface can also be used and not incremented if needed (LMOB_INC vs. LMOB).

Having written a precise address to the buffer, the processor may write or read one byte of data during an instruction cycle. During a write or read cycle, the bus is enabled by the processor for an entire processor instruction. If writing, the control signal (MOB_T) is set high for one processor clock in the middle of the insurrection. The data on the bus is enabled from the MOP data bus and is held stable for approximately two cycles on either side of this signal for the buffer to capture the data. If the cycle is a read instruction, the buffer has previously retrieved the entire word (five bytes) from memory and the particular chosen byte is enabled onto the processor data bus at the beginning of the instruction and strobed into its destination at the end of the execute cycle.

To manipulate the buffer memory directly, the function commands include informing the buffer that the data on the interconnecting bus is the MSB, LSB, or byte location (0-4) of an address. The byte instruction allows the processor to remove or write only one byte of the five byte wide buffer memory.

3.3 Transmission Security Interfaces

There are two TRANSEC inputs to the processor. These interfaces provide the processor with the data to calculate the uplink and downlink permuted hop numbers. These interfaces are completely unidirectional, with the TRANSEC streams occurring at a particular given time during each hop regardless of the status of the last message. If the processor has not retrieved the previous data from the FIFO, it will be overwritten will the new data. For this to happen, the processor must miss a hop call and should be in a reset routine.

The TRANSEC interface contains a serial data stream and a corresponding data clock. The data is received and clocked into serial-to-parallel registers at the falling edges of the clock and then stored in FIFO's until the processor removes it during the next hop operations. Only a small portion of the entire TRANSEC stream is used by the processor for calculations, the rest of the data is discarded.

3.4 Telemetry Interface

The telemetry interface is less defined than all of the other interfaces of the processor. It can be used by command from the EMARC to transmit any amount of data for testing or monitoring purposes. It is likely that in a satellite system this interface would be used periodically to dump the entire MOB control memory and the processor program.

The hardware of the interface is identical to that of the EMARC serial interfaces. The control lines, data lines, and clock rates all operate according to the same protocol.

3.5 Timing Generator Board Interface

The timing generator board interface is the most simplistic of all the processor interfaces. The timing generator transmits the processor clock and the hop sync to the

MOP. The signals are differentially received and brought onto the Xilinx chips. The processor clock is run directly to Xilinx 2 to generate the divide-by-5 control signals. The hop sync is run to Xilinx 1 to generate a system flag for performing hop operations.

The formats of the two signals are shown below:

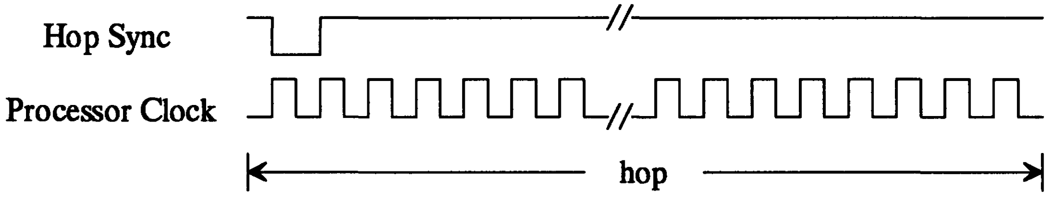


Figure 3.2 Timing Generator Interface Signals

The hop sync signal is low for a specified period of time near the beginning of each hop only. This signal is used to set a hop flag and reset an operation counter. The MOP uses this counter status as a flag to return from long-term operations to the mandatory hop functions. The counter is set to trigger the flag in time for the processor to notice it, store the present long-term operations, and prepare for the next hop. The hop flag is cleared at the end of each set of mandatory hop operations. The hardware is designed to force the program into a soft reset mode if the processor misses a hop call (i.e. a new hop sync arrives before the previous one has been cleared). After a certain number of soft resets have occurred within an epoch, the processor will resort to a hard reset routine to attempt to fix the operation error. As discussed previously, the hard reset routine will reload the RAM software from the PROM prior to restarting operations whereas the soft reset will simply clear all flags and begin operations again. These interrupts provide one way of recovering the MOP if a software error occurs that places the processor out of regular operations.

3.6 Alternate 2.5 MHz Interface

The 2.5 MHz alternate connector provides a clock that can be used for the EMARC output interfaces instead of the clocks received from the EMARC input interfaces. All data from the MC3 and control interfaces will be clocked into the serial to parallel converter with clocks from the same interface (Note: these clocks run continuously, independent of the data line status). To limit the clock distribution skew through the system, the output interfaces (MC2, EMARC response and Telemetry) will all be run from a central clock provided on the 2.5 MHz alternate connector. The processor can run without the alternate clock by using the input from the EMARC control interface if needed. There is a jumper on the board to select between the two signals.

CHAPTER 4

Hardware Design

The processor hardware design has been implemented on one-half of a regular PC board (7.5" x 8"). The design includes two Xilinx 4010PG191-5 chips, two IDT71B256 32Kx8 SRAMs, one CY7C286 64Kx8 PROMs, one Qualcomm 1601 Viterbi Decoder, and the necessary interface and loading hardware. Many of the interface signals are differentially received prior to the Xilinx interface. The tasks of the two Xilinx chips are divided roughly into interface hardware (Xilinx 1) and system control hardware (Xilinx 2).

4.1 Timing

Most of the processor timing is controlled from Xilinx 2. The processor clock from the timing generator board enters Xilinx 2 directly, and is transmitted from there to Xilinx 1. During a hard reset, the processor clock is divided by two to allow for a longer PROM access time. This clock manipulation is performed on Xilinx 2 and is irrelevant to the processing on Xilinx 1 as no operations during the loader involve that chip. The processor clock is distributed on both chips through a primary global buffer to minimize any routing delay in this critical signal.

The hardware in Xilinx 1 also contains a divide-by-five counter to generate the timing control signals (CONTROL0 - CONTROL3). The counter is synchronized to the counter on Xilinx 2 every instruction by using a common execute signal to reset. The hop sync signal and the 2.5 MHz clock both are received directly into Xilinx 1 and are also routed through primary global buffers but are never transmitted to Xilinx 2.

4.2 Xilinx 1

The first 4010 Xilinx contains all of the interface control hardware. The ULTSEC, DLTSEC, MC3 messages, and EMARC control messages are received serially into Xilinx 1 for storage. The storage hardware for these interfaces is identical except for the 'handshaking' ability added for the MC3 and EMARC interfaces. The FIFOs are designed to store 16 bytes of data and provide empty and full flags. The current address of the FIFO RAM can also be accessed and is used to implement the handshaking protocols. As stated previously, buffering incoming messages and storing them for transmission is done without processor control.

TRANSEC

The ULTSEC and DLTSEC hardware are also contained in Xilinx 1. The algorithms used for calculating the permuted hop numbers using the transec streams require processor control as well as specialized hardware. The transec interfaces buffer the data streams and the processor copies the required bytes of data into the hardware blocks. Part of the ULTSEC algorithm requires a table that is copied by processor control into hold registers periodically to be used in the hardware operations. The DLTSEC algorithm includes some detailed calculations performed independently by the processor and used in conjunction with hardware outputs to generate the complete hop address. (Note: The algorithms for both uplink and downlink calculations are classified and the schematics are not included.)

Status Register

Another specialized hardware block contained in Xilinx 1 is the status register. It consists of eight storage flip-flops that may be loaded by command from the bits on the data bus. The data contained in the status register is:

Bit0: Error in the buffer memory
Bit1: Error in the processor memory
Bit2: DLTSEC is currently turned off
Bit3: ULTSEC is currently turned off

If an error is found in the buffer memory either by the buffer or by the processor, a h'01'; is loaded into the status register. This places a '1' in the register bit indicating buffer error, but no other bits will be affected. The only way to clear the contents of Bits 0 or 1 is by a read of the status register. The DLTSEC and ULTSEC bits indicate whether the transec data is being used. The EMARC has the ability to 'turn off' the transec for testing purposes. In this case, the processor simply clears the contents of the transec FIFO prior to operating on it. The transec status bits are not cleared by a status read but are set or cleared by a combination of the data bus bits during a load of the status register.

Hamming Encoder

The Hamming encoder hardware in Xilinx 1 is used to add an eight-bit tail onto the buffer memory control words for error detection purposes. Because of the buffer memory structure, eight bits were available without adding to the control storage requirements. The last bit is used for an overall even parity check; this bit is checked each time the buffer uses a control word. The other seven bits were chosen for maximum distance between possible code words - giving approximately 6 bits of detection. The processor checks the full eight bits of Hamming tail at a rate of several per data frame. If the processor finds an error in a control word, the control word is immediately cleared and the buffer error flag is set.

The hardware consists of four registers to be loaded during four consecutive firmware writes (loadl LHAM RAMA_INC). The XOR tree results are then released onto the bus through a read operation (loadl LRAMA HAM). To check a control word in the buffer memory, the first four bytes of the word are used to regenerate the check byte which is compared with the existing check using the A0 register.

Figure 4.1: Algorithm for eight-bit hamming tail

CRC Tail Generator

The CRC encoder hardware in this system generates a systematic code with four parity check bytes appended to the end of the message. The encoder consists of a 32-bit shift register with a particular set of data taps that are XORed after each data bit shift. The result of the data taps is XORed with the incoming bit of data prior to being shifted into the 32-bit register. After all of the data has been entered in this way, zeros are placed in the input stream, and the next 32 results of the XOR tree are inverted and added to the message as the CRC tail. (Note: There is a two step delay through the hardware, and six flush bits are added at the end of the end of the tail for the convolutional encoder.) Decoding is performed by regenerating the parity check and comparing it with the tail from the received word. At this level of decoding, any non-authenticating messages are ignored.

Convolutional Encoder

The convolutional encoder hardware has been designed to operate on bytes of data rather than on a continuous data stream. The encoding is done in two steps, holding data from one byte to operate in combination with the next. This allows the byte- oriented hardware to simulate code generation on a continuous stream. This encoding technique is not systematic, a set of 'n' output bytes is read from the hardware for every 'k' input bytes. Note: The specifications (rate, constraint length, taps) of the convolutional encoder are classified and the schematics are not included.

ATOW Generator

The hardware for ATOW generation simplifies the processing of accumulated acquisition values. The accumulation results have thresholds subtracted and are then restored in memory (if the result is negative, a zero is stored). The firmware commands then make comparisons between the early and late versions for both sets of data. After this step is complete, the memory contains a h'01' in the location corresponding to the greater magnitude above threshold (for both fine and extra fine). These four values are then loaded into the ATOW hardware for the final two-bit code generation.

Bit Shift/D Register

This specialized hardware, used in conjunction with the MARxB addressing registers, tremendously reduces the number of firmware instructions required in the interleaving/deinterleaving processing. The D register alone can be used as a simple storage register, or may be shifted left or right (shifting in a zero always) with a flag set if the left-shifted bit was a '1'. If used with the contents of the bit shift register, a particular bit can be chosen from the data in the D register, placed in any position in an otherwise zero data byte, and accessed from the bus.

The control data for the selection and placement of the data bit is loaded into the bit shift register (loadl LBSNUM h'44'). The first four bits in this register control the bit to be chosen from D, and the last four control the placement position in a zero byte. The first bit in each field determines if the positions shall be incremented automatically after each read operation (loadl LA0 BITS). In general, many of the steps in the interleaving and deinterleaving operations use succeeding bits from different bytes of data, and the output containing one bit from D can be ORed into the data contained in the A0 register to generate an entire (de)interleaved byte. Having two sets of address registers for the RAM (MARxA and MARxB) allows reading and writing from two different places in memory without needing to continuously reset the addressing. By designing the hardware in this manner, the length of time required to perform interleaving operations were cut by approximately a factor of four.

Interrupts

There is one system interrupt in the processor that does not cause a restart of system timing. It is generated on Xilinx 1 and forces the program operation to the mandatory hop processing location (HOP_MAIN). When this occurs, all of the current operating parameters are stored including the program address. After the hop servicing is complete, the firmware is returned to the last program address plus one to continue operations (jret).

4.3 Xilinx 2

The second Xilinx chip contains all of the system control hardware as well as the Viterbi decoder interface hardware and the ALU unit. The RAM and PROM address selector, data outputs/inputs, and control lines originate from this chip. All of the firmware registers and decoding hardware as well as the parity check hardware for firmware commands are also in Xilinx 2.

Addressing and Interrupt Processing

The addressing for the PROM and RAM has three sources. As previously discussed, there are two sets of registers (MARxA and MARxB) that are automatically incrementable and can be used to access memory storage locations. The third set of address registers is the program counter. The top fourteen bits of the program counter may be loaded directly (as in branch, jump, and interrupt instructions) and are incrementable during normal operation. The bottom two bits come from the divide-by-five counter. All jump and branch statements are synchronous and do not effect these bits. During a hard reset, however, these are set to zero and resynchronized to the loader routine. The priority level for interrupts is:

1. Hard reset
2. Soft reset
3. Hop call

If a system if executing an interrupt routine and an interrupt of higher priority occurs, the system will immediately jump to the higher interrupt program. While executing a higher level interrupt, all interrupts of lower priority are ignored and the forcing signals are cleared before returning to regular operation.

There are also three memory locations that the program can be forced to in an interrupt. At the beginning of a hard reset, the program is sent to the loader routine. The loader routine begins at the first location in memory (h'0000') and the hardware is set up to simply reset (asynchronously) all sixteen bits of the counter for a hard reset. All operations that were being performed when a hard reset occurs are abandoned. After the loader is finished, the processor begins operating as if completely restarted.

At the beginning of a soft reset, the program is sent to the prehop routine. This routine is located at position h'0020' in the software. The top byte of the program register is cleared and the bottom byte (except for the bottom two bits) is loaded with h'20' synchronously. Since this is a synchronous operation (occurs at the end of an instruction), the divide-by-five counter is not affected.

When a hop call interrupt occurs, the program is forced to the main hop routine. This is the part of the firmware that must be performed once every hop and is located at memory h'0030'. This operation is also synchronous, and requires storage of the system conditions. The program counter location is automatically stored prior to jumping to the hop call routine, and can be incremented for a return to the next instruction. At the beginning of the hop call routine, all storage registers and MARxA values are stacked and reloaded at the end of the routine. (Note: MARxB values are not affected by operation of the hop call program.)

To perform a 2-byte jump instruction requires two firmware instructions. The first instruction places the MSByte in the B1 register. The second instruction places the

LSByte on the data bus. At the end of the execute cycle of the second instruction, both new address bytes are strobed into the top 14 bits of the program address register. (The bottom two bits remain in step.) The address set-up for the MSByte comes from the dedicated lines from B1 to the ALU unit, therefore the bus is not required. Regular jump instructions are inhibited by a parity error in a firmware instruction. If a parity error occurs in either half of the instruction, the jump command will be aborted and the EMARC will be notified. The processor will continue operating on the next instruction in the program until/unless a soft reset is commanded.

Decodes

The 32-bit control word is decoded by fields as soon as it is clocked into the second stage of the pipeline structure. All fields are decoded immediately, and the strobes are gated with the execute signal at their destinations.

Arithmetic Logic Unit

The arithmetic logic unit (ALU) consists of a 16-bit adder/subtractor in addition to the four A and B storage registers. The first set of inputs to the ALU come from the A registers directly, with no tri-stating or bus control needed. The second set of inputs can be selected to perform various arithmetic tasks. The two-byte output from the ALU is placed at an input to the A0 register (LSByte) and available for bus access (MSByte). During a full 16-bit operation, the ALU bus control will be selected and both the A0 and A1 registers will be strobed to accept the resulting data.

There are two jump flags generated by an operation involving the arithmetic logic unit. A 'jump on zero' flag is set when the results of an ALU add/subtract is zero. A 'jump on carry' flag is set when an ALU operation results in an overflow. Both flags are unchanged until next operation using the ALU.

All selectors for both addition/subtraction and the 'b' inputs to the ALU are direct from the instruction word. If no arithmetic function is selected, the ALU will perform a subtraction of h'0000' from the A register values. The result will wait at the bus source and A0 inputs but have no affect during a non-arithmetic operation.

The A0 register uses combinational logic and D flip-flops to perform the eight-bit logic operations. The functions may be performed on the data from the A0 register in combination with data from other registers or the instruction data field.

Viterbi Decoder Interface

The Q1601 Viterbi decoder includes a processor interface that can be used to set system performance parameters. The interface includes a five-bit address bus, eight-bit data bus, and control signal to alter the decoder setup or to monitor the system status. The interface hardware allows the MOP to monitor the decoder and simplifies the task of transmitting bytes of data to be decoded.

When operating in the parallel mode with hard decision values, the Q1601 expects two bits of data during each input clock cycle. The interface hardware breaks bytes of encoded data into two-bit pairs to be output to the decoder and enables the input (to the Q1601) clock for each data pair. This allows a byte of data to be sent to the decoder during one processor cycle. After the four pairs of data are sent, the hardware disables the decoder clock until the next set of data is available.

After an entire message has been transmitted to the decoder, the processor transmits flush data (pairs of zeros) until the decoder has generated the complete output sequence. When operating in parallel mode, the delay through the decoder is 102.5 cycles of the input data clock. The decoded data is generated one bit at a time and sent back to the Xilinx interface. The interface hardware packs the serial data into bytes which are then stored in a FIFO until needed.

CHAPTER 5

Firmware Design

5.1 Processor Instruction Word Format

The processor firmware design is based upon a 32-bit control word that is compiled using the Metastep V2.0 system. The fields of the control word are defined as shown in figure 5.1. The parallel loading of the control word begins with bits 31-26, and the last byte loaded is the data field. Because of the pipeline structure of the processor, all fields have the same amount of setup time before use.

31	30 26	25 20	19 14	13 8	7 0
P	Bus Control	Strobes	ALU Control	Jump Control	Data

Figure 5.1: Firmware instruction fields

The definitions of the individual fields are as follows:

1. Parity (1):One bit even parity added after the control word is constructed
2. Bus Control (5): Controls the bus source during an instruction cycle, includes the data field of the control word
3. Strobes (6): Generates the operation strobe to occur at the end of the execute cycle; includes load destinations, resets, and register shifts. (Note: this field is gated with the execute signal at its destination to limit final propagation

delay.)

4. ALU Control (6): Controls all ALU and logic functions.
   ALU0-3 - Decodes to select specific logic or arithmetic function
   ALU4 - Selects add or subtract option
   ALU5 - Indicates INC or DEC operation
5. Jump Control (6): Generates the selection from potentially 16 jump options, selects the active sense of the jump command (jump on true or false), and selects either a local (branch) or a long jump (jump).
6. Data (8): Allows the software to perform data operations on immediate data (included in the software instruction)

The macros that have been written for compiling software jump and branch instructions are included in the Microword Definition File (Appendix C).

5.2 Schedule of Long-Term Duties

As discussed previously, the processor duties can be divided into hop processing tasks and long-term operations. The schedule shown in figure 5.2 gives an estimate of the amount of time required to perform all operations. Most of the processor schedule is identical for any odd or even frame. The preparation and transmission of ATOW messages, however, runs on a schedule that causes 0, 1, or 2 messages to be prepared during any even frame. (The schedule count is made for the worst case.) The MC3 message requirement may also vary depending on the system load.

HOP TIMING/SCHEDULE

	All frames	Odd	Even
Using hop overhead values	33%		
EMARC servicing (Averaged and expected every other hop)	5%		
MC2 Messages (4 per odd frame)		6%	
MC3 Messages (2 per frame)	4%		
Acquisition			
Accumulation (odd and even frames)	1%		
Manipulation (even frames only)			1%
Atow Generation (Worst case: 2/even frame)			4%
		49%	48%

5.2: Timing estimates for processor duties

A copy of the current processor operating program is included in Appendix D. This program will be updated as full system integration proceeds.

CHAPTER 6

Testing and Conclusions

Once the initial design of the processor was complete, a board layout was sent to fabrication. Two test connectors were integrated into the design to aid in debugging and to ease any changes that need to be made. One of the connectors was placed between the two Xilinx chips and routed to sixteen unused pins on each. The other was connected to unused pins on Xilinx 2 and several Viterbi Decoder status signals for test purposes. The test connector placed between the two chips can be used to implement any major changes by routing many signals from one chip to the other.

For initial debugging, the processor was tested alone. External processor clock and power signals were applied to the chip. Testing was done using a Hewlitt Packard 16500 logic analyzer to view the program operations.

The only major change implemented during testing was to add an interrupt driven hop call instead of the previously designed polling system. The major advantage of this change was in time saving. By operating with a polling system, the long-term schedules would lose two instruction cycles periodically (approximately once every 20 cycles) in testing for the hop flag. Changing this to an interrupt function added a level of complexity by changing the number of possible routes through the program. Prior to launching a system such as this all possible program routes should be thoroughly tested.

This interrupt makes that a nearly impossible task considering the size of the long-term program. By limiting the system to only one non-terminal interrupt, however, we can be certain of keeping the system in a known state.

The acquisition processing changed several times during the design and testing of the processor. The final result to be used in the payload simulator was described previously in chapter 2. A second method has been proposed, however, for processing the fine acquisition sequences. This alternative would process twice as many comparisons for delayed versions of the fine sequence and would generate a variable threshold that would reduce the possible affect of a jammer interfering with one acquisition hop. The processors duties for this new system would be expanded somewhat, but would still fall within the time restrictions of the system. No hardware changes will be required to implement this change, all alterations can be performed easily within the firmware structure.

Although the medium data rate onboard processor has been tested individually, much work remains to be done. The processor must be integrated into the complete payload simulator structure. The interface protocols have been completely defined, but the processor hardware and software can be easily adapted in the event that any changes need to be made.

List of Acronyms and Abbreviations

ATOW	Acquisition Tracking Order Wire
BER	Bit Error Rate
BPS	Bits Per Second
CLB	Configurable Logic Block
CRC	Cyclic Redundancy Code
DAMA	Demand Assigned Multiple Access
DLTSEC	DownLink TRANSEC
EHF	Extremely High Frequency
EMARC	Enhanced Micro-Coded Adaptive Routing Controller
FIFO	First-In First-Out
LDR	Low Data Rate
LSB	Least Significant Bit
LSByte	Least Significant Byte
MC2	MDR Control message 2
MC3	MDR Control message 3
MDR	Medium Data Rate
Milstar	New-generation space-communications system to meet the projected minimum essential wartime operational requirements associated with military communications
MOB	MDR Onboard Buffer
MOP	MDR Onboard Processor
MSB	Most Significant Bit
MSByte	Most Significant Byte
PGA	Programmable Gate Array
PROM	Programmable Read-Only Memory
SRAM	Static Random Access Memory
TRANSEC	Transmission Security
ULTSEC	UpLink TRANSEC
UTP	Uplink Time Probes

BIBLIOGRAPHY

Berlekamp, Elwyn R. Algebraic Coding Theory. Laguna Hills, CA: Aegean Park Press. 1984.

Clark, George C. and J. Cain. Error-Correction Coding for Digital Communications. New York: Plenum Press. 1981.

FLTSATCOM EHF Package to Milstar Terminal Interface Control Drawing (LL-1005). Milstar Joint Program Office. 8 October 1985. Secret.

Hirschler-Marchand, P.R. Binary Sequences of Arbitrary Length with Near-Ideal Correlation. Lexington, MA: Lincoln Laboratory. 13 June 1989.

Milstar Payload/Terminal Segment Interface Control Drawing SI-1135. Milstar Joint Program Office. 30 October 1992. Secret.

APPENDIX A

EMARC to MOP Control words (Continued on next page)

			MOP Write	MOB Memory Write	MOB U/L Con. Write	MOB D/L Con. Write	TLM	MOP Read	MOB Read	Pointer Read	U/L ON	U/L OFF	D/L ON	D/L OFF	Hard Reset	Soft Reset	Jump	MR ATOW TABLE	LR ATOW TABLE	STATUS CHECK
Type 0-4	0 4	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
Processor 5-7	5 7		unused (0)	0-7	0-7	0-7		unused (0)	0-7								unused (0)	Address offset	Address offset	
8-15	8 15		# of bytes 1-5	unused (0)	unused (0)	unused (0)		# 32-bit Words	# 40-bit Words											
Address 16-31	16 31		Address	Address	Address	Address		Address	Address								Address	unused (0)	unused (0)	
32–71	32 63	unused (0)	Data	Data	Data	Data	TBD	unused (0)	unused (0)	unused (0)	unused (0)	unused (0)	unused (0)	unused (0)	unused (0)	unused (0)	unused (0)	WORD	WORD	unused (0)
	64 71				unused (0)	unused (0)														

EMARC to MOP Control words (cont.)

LR MC2 TABLE	MR MC2 TABLE
13	14
Address offset	Address offset
unused (0)	unused (0)
WORD	WORD
unused (0)	unused (0)

MOP to EMARC Control words

	Bits	MOP Read	MOB Read	Pointer Read	STATUS CHECK
Type 0-1	0 1	0	1	2	3
Status 2-7	2 7	Status	Status		Status
8-15	8 15	# 32-bit words still to read	# 40-bit words still to read		Proc. #
Address 16-31	16 31	Address of current word	Address of current word	TBD	unused (0)
Data 32–71	32 63	Data	Data		
	64 71	unused (0)			

APPENDIX B

BUFFER FUNCTION DEFINITIONS (PAGE 1 OF 2)

LOCAL FUNCTIONS

(APPLY TO ADDRESSED PROCESSOR ONLY)

FUNCTION	READ (MOB=1)	WRITE (MOB=0)
0	STATUS	READ REQUEST
1	BYTE0	BYTE ADDRESS
2	BYTE1	WORD ADDRESS (LSB)
3	BYTE2	WORD ADDRESS (MSB)
4	BYTE3	DATA
5	BYTE4	SPARE

BUFFER FUNCTION DEFINITIONS (PAGE 2 OF 2)

GLOBAL FUNCTIONS

(APPLY TO ALL PROCESSORS)

FUNCTION	READ (MOB=1)	WRITE (MOB=0)	
6	UNUSED	D/L BUFFER	(XXXXXXBB)
7	UNUSED	D/L PERM (LSB)	
8	UNUSED	D/L PERM (MSB)	(XXXXXXXD)
9	UNUSED	BYTE ADDRESS	
A	UNUSED	WORD ADDRESS (LSB)	
B	UNUSED	WORD ADDRESS (MSB)	
C	UNUSED	DATA	
D	UNUSED	U/L BUFFER	
E	UNUSED	U/L PERM (LSB)	
F	UNUSED	U/L PERM (MSB)	
10	UNUSED	UNUSED MC2 (1)	(XXXXXXXU)
11	UNUSED	FRAMESYNC/38.4 SYNC	(XXXFXXXE)
12	UNUSED	LR MC3 BEAM	(XXXXXBBB)
13	UNUSED	MR MC3 BEAM	
14	UNUSED	LR ATOW BEAM	
15	UNUSED	MR ATOW BEAM	
16	UNUSED	FINE SYNC BEAM	(XEEEXOOO)
17	UNUSED	XFINE SYNC BEAM	

APPENDIX C

```
/*************************************************************************
* Microword Definition File for:  MDR Onboard Processor (MOP)
*        --------------------------------------------------------------
* format:  | Parity | Bus Control | Strobes | ALU | Jump Control | Data  |
*        ---------+---------+------------+-------------+--------+--------
*  bits: |  31   |30 - 26|25  -  20|19   -  14|13 - 8|7 - 0|
*        ---------+---------+------------+-------------+--------+--------
*  size: |   1  |   5   |     6     |      6     |  6  |  8  |
*        --------------------------------------------------------------
* Revision History:
*         11/16/92  EGR -  First Draft.
*         01/21/93  EGR -  Updated Strobes and Flags fields
*         02/05/93  EGR -  added macros, hamming and constraints
*         07/12/93  KAS - Changes to macros, strobe, bus, and flag fields
*         07/14/93  KAS - Add ori, xori, ... macros
*         07/15/93  KAS - Add new branch macro for CPAGE:
*         07/19/93  KAS - Added Viterbi processor controls
*----------------------------------------------------------------------*/
/* Specify the heading and footing for the listing output */
/*-----------------------------------------------------*/

heading 'MDR Onboard Processor (MOP)';
footing '(c) Copyright MIT Lincoln Laboratory 1993';

/* General Usage Equates */
/*---------------------*/
TRUE  equ 1;
FALSE equ ! TRUE;
BRAMASK  equ h'00';
NBRAMASK equ h'10';
JMPMASK  equ h'20';
NJMPMASK equ h'30';
```

```
/* Macro usage variables */
integer mtemp;

/* The 'instruction' statement specifies the version and length of the microword        */
/*-------------------------------------------*/

mop: instruction version ('1.0'), length (32);

/* Parity Field Definition */
/*------------------------*/
_parity: bits (31),                  /* single bit instr. parity   */
         length(1),                   /* field length in bits */
         parity;                     /* assembler will generate it */

/* Bus Control Field Definition */
/*------------------------------*/
bus:   bits (30..26),
        length(5),                    /* field length in bits */
        values( h'00' : NOOP,
              h'01' : ULTSEC,     /* uplink tsec fifo */
              h'02' : ULCAS,           /* uplink cut and swap */
              h'03' : DLTSEC,     /* downlink tsec fifo */
              h'04' : DLCAS,           /* downlink cut and swap */
              h'05' : RAMA,            /* ram addressed from MAR A reg. */
              h'06' : RAMB,            /* ram addressed from MAR B reg. */
              h'07' : STATUS,     /* system status register */
              h'08' : A0,        /* alu a0 register */
              h'09' : A1,        /* alu a1 register */
              h'0A' : B0,        /* alu b0 register */
              h'0B' : B1,        /* alu b1 register */

              h'0D' : RAMA_INC,   /* autoinc MAR A reg */
              h'0E' : RAMB_INC,   /* autoinc MAR B reg */
              h'0F' : CE,        /* convolutional encoder */
              h'10' : CRC,        /* crc generator */
              h'11' : ATOW,            /* atow compare */
        h'12' : ROM,        /* read from rom */
        h'13' : DATA,            /* micro-instruction data field */

        h'15' : EMARCC,     /* emarc cmd fifo */
        h'16' : MC3,       /* mc3 fifo */
        h'17' : D,        /* d register */
        h'18' : VD,        /* viterbi decoder */
        h'19' : MOB,        /* MDR Onboard Buffer memory */
        h'1A' : HE,        /*  hamming encoder */
        h'1B' : BS,        /* bit shifter */
        h'1C' : ULPROM,     /* Uplink Transec PROM data */
        h'1D' : RDA1,      /* data from MAR1A*/
        h'1E' : RDA0 /* data from MAR0A*/
    ),
    valid(h'00'..h'0B',
       h'0D'..h'13',
       h'15'..h'1E'),          /* define valid field values */
    check(busCheck),             /* attach constraint to field */
    default(NOOP);

/* Strobe Field Definition */
/*-----------------------------*/
/* Strobes are generally only used for loading data to/from */
/* registers, reseting flip-flops, and for register control */
/* functions such as SBR (shift B right).              */
/*-------------------------------------------------------*/
strobe: bits (25..20),
    length(6),              /* field length in bits */
    values( h'00' : NOOP,
        h'01' : LULCAS,    /*load*/
        h'02' : LDLCAS,
        h'03' : LCRC,
        h'04' : LEMARCR,
        h'05' : LHE,
        h'06' : LVD,
        h'07' : LMC2,
h'08' : LFUNC,
h'09' : LSTATUS,
h'0A' : LA0,
h'0B' : LA1,
h'0C' : LB0,
h'0D' : LB1,
h'0E' : LD,
h'0F' : LBSNUM,
h'10' : LCE,
h'11' : LATOW,
h'12' : LMOB,
h'13' : LMOB_INC,
h'14' : LTLM,
h'15' : LPROC,
h'16' : LMAR0A,
h'17' : LMAR1A,
h'18' : LMAR0B,
h'19' : LMAR1B,
h'1A' : LULPROM,
h'1B' : LVITADD,
h'1C' : LVITDATA,

h'1E' : LRAMA,
h'1F' : LRAMB,

h'21' : RULCAS,
h'22' : RDLCAS,
h'23' : RCRC,
h'24' : REMARCR,
h'25' : RHE,
h'26' : RVD,
h'27' : RLOAD,
h'28' : R38_4,
h'29' : RHOP,
h'2A' : RDLTSEC,
h'2B' : RULTSEC,
                h'2C' : REMARCC,
                h'2D' : RMC3,
                h'2E' : RMC2,
                h'2F' : RTLM,
                h'30' : RVITDE,
                h'31' : RLOAD_H,

                h'33' : CPA,
                h'34' : RSR,
                h'35' : LDRA,
                h'36' : SRINC,
                h'37' : SBR,
                h'38' : SBL,
                h'39' : SDR,
                h'3A' : SDL,

                h'3C' : SR_EMARC,
                h'3D' : HR_EMARC,
                h'3E'  : LRAMA_INC,
                h'3F'  : LRAMB_INC
            ),
            valid(h'00'..h'1C',
                h'1E'..h'1F',
                h'21'..h'31',
                h'33'..h'3A',
                h'3C'..h'3F'),          /* define valid field values */
            check(strobeCheck),          /* attach constraint to field */
            default(NOOP);

/* ALU Control Field Definition  */
/*-------------------------------*/
alu:   bits (19..14),
            length(6),              /* field length in bits */
            values( h'00' : NOOP,
                h'02' : OR_BUS,
                h'04' : XOR_BUS,
            h'06' : AND_BUS,

            h'08' : SUB_B,
            h'09' : SUB_BUS,

            h'18' : ADD_B,
            h'19' : ADD_BUS,

            h'21' : DEC,
            h'31' : INC
        ),
        valid(h'00',
            h'02',
            h'04',
            h'06',
            h'08'..h'09',
            h'18'..h'19',
            h'21',
            h'31'),          /* define valid field values */
        check(aluCheck),             /* attach constraint to field */
        default(NOOP);

/* Jump Control Field Definition */
/*-------------------------------*/
jump:   bits (13..8),
        length(6),                   /* field length in bits */
        values( h'00' : NOOP,
              h'01' : ROMMAXF,        /* rom max flag set */
              h'02' : 38_4F,          /* 38.4 flag set */
              h'03' : HOPF,           /* hop flag set */
              h'04' : EMARCCF,        /* emarc command flag set */
              h'05' : EMARCRF,        /* emarc response flag set */
              h'06' : MC2F,           /* mc2 flag set */
              h'07' : MC3F,           /* mc3 flag set */
              h'08' : TLMF,           /* tlm flag set */
              h'09' : CF,             /* carry flag set */
```

```
   h'0A' :  SIGNAF,          /* sign A flag set */
   h'0B' :  SIGNDF,           /* crc flag set */
   h'0C' :  ZF,            /* zero flag set */
   h'0D' :  DF,            /* D flag set */

/* h'0F :  NOTUSEDF,          NOT USED */

   h'10' :  BUNCND,          /* unconditional branch */
   h'11' :  NROMMAXF,        /* rom max flag not set */
   h'12' :  N38_4F,         /* 38.4 flag not set */
   h'13' :  NHOPF,          /* hop flag not set */
   h'14' :  NEMARCCF,       /* emarc_command flag not set */
   h'15' :  NEMARCRF,
   h'16' :  NMC2F,
   h'17' :  NMC3F,
   h'18' :  NTLMF,
   h'19' :  NCF,
   h'1A' :  NSIGNAF,
   h'1B' :  NSIGNDF,
   h'1C' :  NZF,
   h'1D' :  NDF,

/* h'1F :  NNOTUSEDF,         NOT USED */

/* h'20' :  INVALIDF,         INVALID INSTRUCTION */
   h'21' :  JROMMAX,         /* jump flag set */
   h'22' :  J38_4,
   h'23' :  JHOP,
   h'24' :  JEMARCC,
   h'25' :  JEMARCR,
   h'26' :  JMC2,
   h'27' :  JMC3,
   h'28' :  JTLM,
   h'29' :  JC,
   h'2A' :  JSIGNA,
   h'2B' :  JSIGND,
          h'2C' :  JZ,
          h'2D' :  JD,

        /* h'2F' :  J2F,                NOT USED */

          h'30' :  JUNCND,           /* jump always */
          h'31' :  JNROMMAX,          /* jump flag not set */
          h'32' :  JN38_4,
          h'33' :  JNHOP,
          h'34' :  JNEMARCC,
          h'35' :  JNEMARCR,
          h'36' :  JNMC2,
          h'37' :  JNMC3,
          h'38' :  JNTLM,
          h'39' :  JNC,
          h'3A' :  JNSIGNA,
          h'3B' :  JNSIGND,
          h'3C' :  JNZ,
          h'3D' :  JND
       /*  h'3F' :  JN3F            not used therefore invalid */
      ),
      valid(h'00'..h'0D',            /* define valid field values */
            h'10'..h'1D',
            h'21'..h'2D',
            h'30'..h'3D'),
      check(jumpCheck),                /* attach constraint to field */
      default(NOOP);

/* Data Field Definition */
/*-----------------------*/
data:   bits (7..0),
        length(8),                  /* field length in bits */
        values(h'00':NOOP),
        valid(h'00'..h'FF'),          /* define valid field values */
        check(dataCheck),               /* attach constraint to field */
        mask,                       /* use only lower order bits */
```

```
          default( NOOP );

   endInstruction;
   eject;

   /*---------------------------------------------
   * MicroWord Instruction Field Constraint Macros
   *---------------------------------------------*/

macro strobeCheck;  /* error check strobe field */
begin
endm;

macro busCheck;  /* error check bus field */
begin
endm;

macro jumpCheck;  /* error check jump field */
begin
endm;

macro aluCheck;  /* error check alu field */
begin
endm;

macro dataCheck; /* error check data field */
begin
endm;

  /*---------------------------------------------------------------
   * Instruction: noop   (No Operation)
   *
   * Description: This is the "do-nothing" but eat 5 processor cycles macro.
   *          It is intended that with even parity, the NOOP instruction
   *          should result in a 32 bit opcode of 0x0000. (all zeros)
   *
```

```
* Usage:     noop;
*-----------------------------------------------------------------*/
macro noop;
begin
 bus = NOOP,
 strobe = NOOP,
 alu = NOOP,
 jump   = NOOP,
 data   = NOOP,
endm;


/*-----------------------------------------------------------------
* Instruction: bra  (branch)
*
* Description: branch within local page. Page rollover not checked.
*
* Usage:  bra address <condition>
*-----------------------------------------------------------------*/
macro bra &addr &cnd;
begin
        #if ( ! ( narg in [ 1..2 ] ) )
                error 'Wrong number of arguments for bra';
        #endIf,
        #if ( narg == 1 )       /* unconditional branch */
                jump = BUNCND | BRAMASK,
        #elseIf ( &cnd in [ ROMMAXF..DF ] )  /* branch if flag set */
                jump = &cnd | BRAMASK,
        #elseIf ( &cnd in [ NROMMAXF..NDF ] ) /* branch if flag not set */
                jump = &cnd | NBRAMASK,
        #else
                error 'Condition is not valid for bra';
        #endIf,

        bus = DATA,
        strobe = NOOP,
        alu    = NOOP,
        data   = &addr,          /* put address on bus */
endm;

/*-----------------------------------------------------------------
 * Instruction: bra_a0  (branch)
 *
 * Description: branch used for CPAGE addressing. Page rollover not checked.
 *
 * Usage:  bra_a0 <condition>
 *-----------------------------------------------------------------*/
macro bra_a0 &cnd;
begin
        #if ( ! ( narg in [ 0..1 ] ) )
                error 'Wrong number of arguments for bra';
        #endIf,
        #if ( narg == 0 )        /* unconditional branch */
                jump = BUNCND | BRAMASK,
        #elseIf ( &cnd in [ ROMMAXF..DF ] )  /* branch if flag set */
                jump = &cnd | BRAMASK,
        #elseIf ( &cnd in [ NROMMAXF..NDF ] ) /* branch if flag not set */
                jump = &cnd | NBRAMASK,
        #else
                error 'Condition is not valid for bra';
        #endIf,

        bus = A0,
        strobe = NOOP,
        alu    = NOOP,
        data   = NOOP,
endm;

/*-----------------------------------------------------------------
 * Instruction: jmp  (jump)
 *
 * Description: jump anywhere
 *
```

```
* Usage:  jmp address <condition>
*----------------------------------------------------------------*/
macro jmp &addr &cnd;
begin

        bus = DATA,
        strobe = LB1,
        alu    = NOOP,
        data   = &addr >> 6;    /* put address on bus */

        #if ( narg == 1 )         /*unconditional jump*/
                jump = JUNCND,
        #elseIf ( &cnd in [ ROMMAXF..DF ] )  /* jump if flag set */
                jump =  &cnd | JMPMASK,
        #elseIf ( &cnd in [ NROMMAXF..NDF ] )  /* jump if flag not set */
                jump = &cnd | NJMPMASK,
        #else
                error 'Condition is not valid for jmp';
        #endIf,

        bus = DATA,
        strobe = NOOP,
        alu    = NOOP,
        data   = &addr,          /* put address on bus */
endm;

/*----------------------------------------------------------------
* Instruction: jret  (return to the subroutine calling address (plus one from srinc)
*
* Description: jump anywhere
*
* Usage:  jret <condition>
*----------------------------------------------------------------*/
macro jret &cnd;
begin
        #if ( ! ( narg in [ 0..1 ] ) )
```

```
            error 'Wrong number of arguments for jmp';
        #endIf,
        #if ( narg == 0 )       /* unconditional jump */
            jump = BUNCND,
        #elseIf ( &cnd in [ ROMMAXF..DF ] )  /* jump if flag set */
            jump =  &cnd | BRAMASK,
        #elseIf ( &cnd in [ NROMMAXF..NDF ] )  /* jump if flag not set */
            jump = &cnd | NBRAMASK,
        #else
            error 'Condition is not valid for jmp';
        #endIf,

        bus = DATA,
        strobe = RSR,
        alu    = NOOP,
        data  = NOOP;    /* put address on bus */
endm;

/*--------------------------------------------------------------
 * Instruction: jsr  (jump to subroutine - stores current address into registers)
 *
 * Description: jump anywhere
 *
 * Usage:  jsr address <condition>
 *--------------------------------------------------------------*/
macro jsr &addr &cnd;
begin

        bus = DATA,
        strobe = LB1,
        alu    = NOOP,
        data   = &addr >> 6;    /* put address on bus */

        #if ( narg == 1 )       /*unconditional jump*/
            jump = JUNCND,
        #elseIf ( &cnd in [ ROMMAXF..DF ] )  /* jump if flag set */
```

```
            jump =  &cnd | JMPMASK,
        #elseIf ( &cnd in [ NROMMAXF..NDF ] )  /* jump if flag not set */
            jump = &cnd | NJMPMASK,
        #else
            error 'Condition is not valid for jmp';
        #endIf,

        bus = DATA,
        strobe = LDRA,
        alu    = NOOP,
        data   = &addr,         /* put address on bus */
endm;

/*----------------------------------------------------------------
 * Instruction: loadl (load local)
 *
 * Description: sets the destination (strobe) and source (bus release)
 *          flags to allow data transfers between registers on the
 *          local data bus.  The 'load' command is reserved for the
 *          linker.
 *
 * Usage:  loadl destination source
 *-----------------------------------------------------------------*/
macro loadl &dest &src;
begin
        #if ( narg != 2 )
            error 'Wrong number of arguments for loadl';
        #endIf,
        #if ( ! ( &dest in [ LULCAS..LVITDATA] )  &&
            ! ( &dest in [ LRAMA..LRAMB] ) &&
            ! ( &dest in [ LRAMA_INC..LRAMB_INC]))
              error 'Invalid loadl dest';
        #endIf,
        #if ( ! ( &src in [ ULTSEC..B1 ] ) &&
              ! ( &src in [RAMA_INC..DATA ] ) &&
              ! ( &src in [EMARCC..RDA0 ] ))
            error 'Invalid loadl src';
        #endIf,

        bus = &src,     /* strobe source to release data onto bus */
        strobe = &dest,    /* strobe destination to load data from bus */
        alu   = NOOP,
        jump  = NOOP,
        data  = NOOP,
endm;

/*-----------------------------------------------------------------
 * Instruction: loadi (load immedate)
 *
 * Description: loads data from microword data field into the stobed
 *           destination.
 *
 * Usage:  loadi destination value <MSB>
 *-----------------------------------------------------------------*/
macro loadi &dest &val &ctl;
begin
        set mtemp = &val;
        #if ( '&ctl' != '<nil>' )
            #if ( '&ctl' == 'MSB' )
                set mtemp = &val >> 8;
            #elseif ( '&ctl' != 'LSB' )
                error 'loadi control must be either MSB or LSB';
            #endif,
        #endif,

        bus   = DATA,     /* strobe data register to release data */
        strobe = &dest,    /* strobe destination to load data from bus */
        alu   = NOOP,
        jump  = NOOP,
        data  = mtemp,     /* assign data */
endm;
```

```
/*----------------------------------------------------------------
 * Instruction: reset
 *
 * Description: reset specified flag by hitting the appropriate strobe.
 *
 * Usage:  reset p1
 *----------------------------------------------------------------*/
 macro reset &p1;
 begin
        #if ( ! ( &p1 in [RULCAS..RLOAD_H ] ))
                error 'Invalid reset option';
        #endIf,

        bus    = NOOP,
        strobe = &p1,       /* strobe P1 to clear it */
        alu    = NOOP,
        jump   = NOOP,
        data   = NOOP,
 endm;

/*----------------------------------------------------------------
 * Instruction: sbr (shift B right)
 *
 * Description:
 *
 * Usage: sbr;
 *----------------------------------------------------------------*/
 macro sbr;
 begin
        bus    = NOOP,
        strobe = SBR,
        alu    = NOOP,
        jump   = NOOP,
        data   = NOOP,
 endm;
```

```
/*---------------------------------------------------------------
 * Instruction: sbl (shift B left)
 *
 * Description:
 *
 * Usage: sbl;
 *---------------------------------------------------------------*/
 macro sbl;
 begin
         bus    = NOOP,
         strobe = SBL,
         alu    = NOOP,
         jump   = NOOP,
         data   = NOOP,
 endm;

/*---------------------------------------------------------------
 * Instruction: sdr (shift D right)
 *
 * Description:
 *
 * Usage: sdr;
 *---------------------------------------------------------------*/
 macro sdr;
 begin
         bus    = NOOP,
         strobe = SDR,
         alu    = NOOP,
         jump   = NOOP,
         data   = NOOP,
 endm;

/*---------------------------------------------------------------
 * Instruction: sdl (shift D left)
 *
 * Description:
```

```
*
* Usage: sdl;
*-----------------------------------------------------------------*/
macro sdl;
begin
        bus    = NOOP,
        strobe = SDL,
        alu    = NOOP,
        jump   = NOOP,
        data   = NOOP,
endm;

/*----------------------------------------------------------------
* Instruction: cpa (copy MARxA into shadow registers)
*
* Description:
*
* Usage: cpa;
*-----------------------------------------------------------------*/
macro cpa;
begin
        bus    = NOOP,
        strobe = CPA,
        alu    = NOOP,
        jump   = NOOP,
        data   = NOOP,
endm;

/*----------------------------------------------------------------
* Instruction: srinc (Increment the subroutine return address)
*
* Description:
*
* Usage: srinc;
*-----------------------------------------------------------------*/
macro srinc;
```

```
begin
        bus    = NOOP,
        strobe = SRINC,
        alu    = NOOP,
        jump   = NOOP,
        data   = NOOP,
endm;

/*-----------------------------------------------------------------
 * Instruction: add_bus
 *
 * Description:
 *
 * Usage:   add_bus source
 *-----------------------------------------------------------------*/
macro add_bus &p1;
begin
        bus    = DATA,
        strobe = NOOP,
        alu    = ADD_BUS,
        jump   = NOOP,
        data   = &p1,
endm;

/*-----------------------------------------------------------------
 * Instruction: addi_bus
 *
 * Description:
 *
 * Usage:   addi_bus value
 *-----------------------------------------------------------------*/
macro addi_bus &val;
begin
        bus    = DATA,
        strobe = NOOP,
        alu    = ADD_BUS,
        jump   = NOOP,
        data   = &val,
endm;

/*-----------------------------------------------------------------
 * Instruction: add_b
 *
 * Description:
 *
 * Usage:   add_b
 *-----------------------------------------------------------------*/
macro add_b;
begin
        bus    = NOOP,
        strobe = NOOP,
        alu    = ADD_B,
        jump   = NOOP,
        data   = NOOP,
endm;

/*-----------------------------------------------------------------
 * Instruction: sub_b
 *
 * Description:
 *
 * Usage:   sub_b
 *-----------------------------------------------------------------*/
macro sub_b;
begin
        bus    = NOOP,
        strobe = NOOP,
        alu    = SUB_B,
        jump   = NOOP,
        data   = NOOP,
endm;
```

```
/*---------------------------------------------------------------
 * Instruction: sub_bus
 *
 * Description:
 *
 * Usage:   sub_bus source
 *---------------------------------------------------------------*/
macro sub_bus &p1;
begin
        bus    = DATA,
        strobe = NOOP,
        alu    = SUB_BUS,
        jump   = NOOP,
        data   = &p1,
endm;

/*---------------------------------------------------------------
 * Instruction: subi_bus
 *
 * Description:
 *
 * Usage:   subi_bus value
 *---------------------------------------------------------------*/
macro subi_bus &val;
begin
        bus    = DATA,
        strobe = NOOP,
        alu    = SUB_BUS,
        jump   = NOOP,
        data   = &val,
endm;

/*---------------------------------------------------------------
 * Instruction: inc
 *
 * Description:
```

```
*
* Usage:   inc
*---------------------------------------------------------------*/
macro inc;
begin
        bus    = NOOP,
        strobe = NOOP,
        alu    = INC,
        jump   = NOOP,
        data   = NOOP,
endm;

/*---------------------------------------------------------------
* Instruction: dec
*
* Description:
*
* Usage:   dec
*---------------------------------------------------------------*/
macro dec;
begin
        bus    = NOOP,
        strobe = NOOP,
        alu    = DEC,
        jump   = NOOP,
        data   = NOOP,
endm;

/*---------------------------------------------------------------
* Instruction: and_bus
*
* Description:
*
* Usage:   and_bus  source
*---------------------------------------------------------------*/
macro and_bus &p1;
```

```
begin
        bus    = DATA,
        strobe = NOOP,
        alu    = AND_BUS,
        jump   = NOOP,
        data = &p1,
endm;

/*---------------------------------------------------------------
* Instruction: andi_bus
*
* Description:
*
* Usage:   andi_bus value
*---------------------------------------------------------------*/
macro andi_bus &val;
begin
        bus    = DATA,
        strobe = NOOP,
        alu    = AND_BUS,
        jump   = NOOP,
        data = &val,
endm;

/*---------------------------------------------------------------
* Instruction: or_bus
*
* Description:
*
* Usage:   or_bus  source
*---------------------------------------------------------------*/
macro or_bus &p1;
begin
        bus    = DATA,
        strobe = NOOP,
        alu    = OR_BUS,
        jump  = NOOP,
        data  = &p1,
endm;

/*----------------------------------------------------------------
 * Instruction: ori_bus
 *
 * Description:
 *
 * Usage:  ori_bus  value
 *----------------------------------------------------------------*/
macro ori_bus &val;
begin
        bus   = DATA,
        strobe = NOOP,
        alu   = OR_BUS,
        jump  = NOOP,
        data  = &val,
endm;

/*----------------------------------------------------------------
 * Instruction: xor_bus
 *
 * Description:
 *
 * Usage:  xor_bus  source
 *----------------------------------------------------------------*/
macro xor_bus &p1;
begin
        bus   = DATA,
        strobe = NOOP,
        alu   = XOR_BUS,
        jump  = NOOP,
        data  = &p1,
endm;
```

```
/*----------------------------------------------------------------
 * Instruction: xori_bus
 *
 * Description:
 *
 * Usage:   xori_bus value
 *-----------------------------------------------------------------*/
 macro xori_bus &val;
 begin
         bus    = DATA,
         strobe = NOOP,
         alu    = XOR_BUS,
         jump   = NOOP,
         data   = &val,
 endm;

 /* Tell definition assembler we are done. */
 /*----------------------------------------*/
endDefinition;
= NOOP,
 endm;
```

APPENDIX D

```
/***************************MOP OPERATING PROGRAM******************************/

/*****************************************************************************
*
*       This program is the shell from which all processor operations are performed. It operates over a
*       two-frame cycle with various tests for operations that are performed only during specific frames.
*
*       Note that from the interrupt hardware setup, the hop_main routine MUST reside at location
*       h'0000'.
*
*****************************************************************************/
segment mainSeg;

entry start, HOP_MAIN, INC_HOP, CHECK_DL, NEXT0, NEXT, DL_OP,
PRIORITY, SCHEDULE, DLPF40, BIT0, BIT1, BIT2, BIT3, BIT4, BIT5, BIT6, BIT7,
NSET40, FRAME, USED_MC2, EVEN, ODD, ULBUFF, STORE_UL, FRM_SYNC,
SYNC_BMS, FINE, XFINE, DLFRAME, STORE_DL, ULP_OP, LOC2, LOC1,
STR_ODD, LDULP, EPIC, MOP_READ, MOB_READ, CONT_R, MC3_READ, LOADER, DLONC,
ULONC, PREHOP;

LHOP equ h'800a';
DL_PF40 equ h'8015';
MULT equ h'8016';
RMOP equ h'8055';
RMOB equ h'8058';
RMOB1 equ h'8059';
LFRAME equ h'8011';
LULBUFF equ h'8013';
LDLBUFF equ h'8014';
MC2TSEC equ h'80c0';
ULPROM0 equ h'80b4';
ULPROM1 equ h'80b5';
ULPROM2 equ h'80b7';
PROM_REG equ h'dd00';
FSYNC equ h'8040';
ULTSECC equ h'8001';
DLTSECC equ h'8002';
PSIZE equ h'8000';      /*Actually, FFFF-Program size*/
TEST1 equ h'80f0';
STACK equ h'7f00';

start:

org     h'0000';
include 'LOAD'                      /*Contains the code used to load the RAM from the PROM*/
                                    /*Also Contains branch to the code used for soft reset routine*/

loadi   LSTATUS        h'21';       /*Reenables interrupts*/
WAIT:
jmp     WAIT;                       /*Puts the program into a loop waiting until a hop call occurs*/
```

```
org      h'0030';
include 'HOP_MAIN'

/*Contains main code pertaining to operations that must be performed every hop in addition to functions
*        that must be performed on precise hops.
*
*        DLPF40                  Every 8th hop (on the 7s)
*        INC_HOP                 Every hop
*        FRAME                   Beginning of each uplink frame
*        DLFRAME                 Beginning of each downlink frame
*        EPOCH                   Beginning of each epoch
*        ULTSEC                  Every hop
*        ULPROM                  Even hops
*        DLTSEC                  Every hop
*
********************************************************************************/

/*Now  the long-term operations must be included*********************************************
*
*        Schedule of duties to be performed over multiple hop cycles
*
*        Include: MC2 operations              4 / odd frame
*                 MC3 operations              ~ 2 / frame
*                 Acquisition/ATOW
*                         Accumulation        Every frame - Late in the frame (Last 35 hops)
*                         Manipulation        Even frames
*                         Message gen.        Scheduled
*                 Test MOB Control msgs       1 / frame
*
*                         Wait for hop call / frame restart
*
********************************************************************************/

include 'MOBTEST'                             /*Tests the MOB control memory one message per
                                              frame*/
include 'MC3WR_P'

/*Subroutines that are called from programs above*/

include 'PREHOP1'                             /*Called from LOAD*/

include 'HOP_8th'                             /*Called from HOP_MAIN*/
include 'MC2WR_P'
include 'FRAME_P'
include 'DLFRM_P'
include 'EPOCH_P'
include 'ULPROM_P'
include 'EMARCC_P'
include 'MC3RD_P'

end;
```

```
/*Program: LOAD************************************************************************
*
*        Performs the loading of RAM from PROM
*
***************************************************************************************/

        loadi    LSTATUS       h'20';            /*Disables interrupts during the loader routine*/
        reset  RLOAD;
        loadi  LMAR1A  h'00';
        loadi  LMAR0A  h'00';
        loadi  LA1    h'00';
        loadi  LA0    h'00';

         reset  RULTSEC;
        reset  RDLTSEC;

        loadi  LA1    PSIZE  MSB;
        loadi  LA0    PSIZE  LSB;

/*___________________________________________________________________________________*/

LOADER:

        loadl  LRAMA_INC     ROM;
        inc;

        jmp    LOADER  NCF;
         noop;

         reset  R38_4;
         reset  RLOAD_H;

         jmp    PREHOP;
         noop;

/*___________________________________________________________________________________*/

org h'20';                                       /*Soft reset interrupt location*/

PREHOP:                      /*Location 32*/

        jmp    PREHOP1;
        noop;

/*___________________________________________________________________________________*/

/*Program: PREHOP1*******************************************************************/

PREHOP1:

        reset  RLOAD;
        loadi  LVITADD h'02';        /*Parallel, QPSK*/
        loadi  LVITDATA       h'04';

        loadi  LVITADD h'03';        /*No descrambler, no differential decoder*/
        loadi  LVITDATA       h'01';        /*direct data, sign-magnitude*/

        loadi  LVITADD h'04';        /*Set reset bit, clear at beginning of frame*/
        loadi  LVITDATA       h'04';

        loadi  LVITADD h'15';        /*Requires 1-time initialization to zero*/
        loadi  LVITDATA       h'00';

        loadi  LVITADD h'16';        /*Requires 1-time initialization to zero*/
        loadi  LVITDATA       h'00';

        loadi  LMAR1A LHOP   MSB;
        loadi  LMAR0A LHOP   LSB;
        loadi  LRAMA_INC      h'00';
        loadi  LRAMA  h'00';

        loadi  LMAR0A  LFRAME LSB;
        loadi  LRAMA  h'00';

/*done*/
```