SPACECRAFT MINICOMPUTER FOR CONTROL OF A COMMUNICATIONS PAYLOAD*

Russell R. Rhodes and Marilyn D. Semprucci
Massachusetts Institute of Technology, Lincoln Laboratory
Lexington, Massachusetts 02173

ABSTRACT

The paper describes the design, use and testing of a spacecraft minicomputer. The minicomputer was designed as a controller for the communications payload of a Communications Satellite with about 100 communications channels. Each of the channels has very flexible control including variable data rate, Doppler offset, and flexible routing.

INTRODUCTION

Conventional communications satellites weigh on the order of a ton and have on the order of 1 kilowatt of prime power available. Most communications satellites are transponders. They receive and filter uplink signals, translate the frequency, power amplify and retransmit the signal on the downlink. This method of communication has potentially serious interference problems. Any strong signal in the communications band can saturate the downlink transmitter and interferes with communications. The interference can be from a too strong user or jamming intentional or unintentional. The performance of a communications satellite can be greatly improved by using approximately 10% of the power and weight available (about 100 lbs and 100 watts) for signal processing. The 10% reduction in power available for downlink transmission will reduce transmitter power by less than 1 dB. With about 100 lbs and 100 watts several of the following signal processing techniques can be used: (1) frequency hopping (2) demodulation to digital bits (3) decoding and deinterleaving (4) encoding and interleaving (5) data buffering and formatting (6) remodulation (7) antenna nulling and beam steering. With the above techniques the uplink can be frequency division multiplexed and the downlink can be time division multiplexed. The bit rate can be adjusted to accommodate small uplink

* This work was sponsored by the Department of the Air Force.

The views and conclusions contained in this document are those of the contractor and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the United States Government.
transmitters and the time per bit can be adjusted on the downlink to accommodate disadvantaged downlink receivers. Noise and interference are stripped off in demodulation and are not retransmitted on the downlink. Data formatting and buffering will permit otherwise incompatible terminals to communicate.

A satellite payload with this much complexity and flexibility requires formidable control. The MARC (Microprogrammed Adaptive-Routing Controller) a compact, high-performance 16-bit minicomputer was designed to control the payload of a complex signal-processing communications satellite in synchronous orbit.

The environmental requirements for a component in a synchronous satellite are actually rather mild, with two exceptions: it must survive the shock and vibration of launch, plus the solar radiation. Shielding is of a very limited effectiveness. Most bipolar devices such as TTL can easily meet this requirement; however, it eliminates most one-chip microprocessors. The 8080, for example, would survive about a week at synchronous orbit.

Work is proceeding to radiation-harden CMOS and I^2L microprocessors. One of these technologies may be a good choice in the future. The MARC is realized with low-power Schottky TTL technology, of known and proven radiation hardness.

Weight and power are, of course, critical parameters in a synchronous satellite. For digital hardware, power is more important. The weight of digital IC’s is very small, but the weight required for generation and conversion of prime power and for extra metal to remove the heat add up to 1 to 1.5 pounds per watt. The MARC CPU (without I/O memory) requires approximately 10 watts. Potentially flyable one-chip microprocessors require anywhere from a few hundred milliwatts (CMOS) to about 2 watts (I^2L). The CMOS chips are far less capable than the MARC; the I^2L chips are somewhat less capable. Presently available radiation-hard memory (LS-TTL) requires about 3.5 watts per 1K words of 16 bits. Clearly, the power required for memory could easily amount to tens of watts even in a moderate sized system, swamping the power disadvantage of the MARC. Even when lower-power memory becomes available, the MARC might still be the better choice over potentially flyble one-chip CPU’s where its superior speed and flexibility were advantageous.

The MARC packaged for flight in a typical configuration including the CPU I/O and 6K 16 bit words of memory would require about 35 watts and weigh about 20 lbs.

The MARC has a 305 nanosecond microcycle time. This rather slow time allows for derating of component speeds due to radiation. The minimum programmed instruction, like Register to Register adds or logical functions, requires 4 microcycles (1.22 microseconds).
Memory reference instruction requires an extra microcycle to load the memory address register and extra microcycles for computing the memory address if the selected addressing mode requires computation.

**Typical MARC Instruction Execution Times**

<table>
<thead>
<tr>
<th>Types</th>
<th>Microcycles *</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register to Register Arithmetic and Logical</td>
<td>4-6</td>
</tr>
<tr>
<td>Load Immediate Word</td>
<td>4-6</td>
</tr>
<tr>
<td>Test without Jump</td>
<td>4-6</td>
</tr>
<tr>
<td>Memory ↔ Register Arithmetic and Logical</td>
<td>10-14</td>
</tr>
<tr>
<td>Set/Clear one bit in 32K memory table</td>
<td>10-14</td>
</tr>
<tr>
<td>Multiply</td>
<td>26-42</td>
</tr>
<tr>
<td>Divide</td>
<td>59-75</td>
</tr>
</tbody>
</table>

* Most testing at 305 nsec per microcycle. Maximum speed ~ 200 nsec. With new premium chips, probably could be 100-150 nsec.

**The MARC Function**

The MARC was designed to control the advanced technology signal-processing communications satellite payload, shown conceptually in Fig. 1. This system is designed to serve approximately 100 simultaneous communications users, in several uplink frequency bands, with a maximum of flexibility and efficiency. There are several receiver front ends, some with frequency-hopped local oscillators, several digital group demodulators capable of handling up to 16 simultaneous FDH signals with any of several digital modulation forms, and several Communications Output Processors (COP’s). The latter device is a combination switch, data buffer, and TDM signal formatter which generates a downlink data stream. Selected channels can use error-correcting decoders and encoders. Any uplink channel can be connected to any downlink channel or time slot, with any of several modulation types. Completely incompatible ground terminals can be interconnected. A communications satellite with this much flexibility clearly presents unprecendented control problems; these the MARC was designed to solve.
The MARC interprets commands from the ground controller, which sets policy/or makes specific connections. The MARC must interpret these command and change the necessary switches and data tables. The MARC also receives requests from users of the system, and assigned communications resources consistent with the priorities set by the ground controller. The MARC must set up the center frequencies, timing, data rates, and modulation formats for up to 16 uplink channels in each Group Demodulator. It must control the data rates, time division multiplex format, and data connections in the Communications Output Processor. It must also set the bus switches to interconnect modules.

MARC System Architecture

The major sections of the MARC are shown in Fig. 2. The CPU is four AMD 2901’s. The MARC is controlled by the microprogrammed control section. MARC program instructions reside in main memory. They are fetched by a microcode routine. The instruction op code specifies a jump to the appropriate microcode routine to implement the instruction. Microcode resides in ROM inside the “Microprogrammed control” section. The development interface connects the MARC to a NOVA 3/12 host minicomputer in which resides all real-time support and development system software. The NOVA minicomputer can be connected to an IBM 370 via a time share line. A special microcode assembler and MARC code assembler were written for the IBM 370. The assembled files are transferred via the time share line to a disk file on the NOVA development system.

The development system can load both main and microcode MARC memory. It can also examine and modify any of the CPU registers.
Microcode memory is loaded by controlling the microcode address bus from the development system and passing data via the 16 bit Data Bus. Registers and main memory are loaded or modified by loading and starting a short microcode program which transfers data. The development system also has built in traps which can halt the MARC on any access of any specific memory location. In addition, it has the ability to trace microcode steps and keep track of microcode address and bus contents. It was discovered, however, during the development that the use of a logic analyzer eliminated the need for both traps and trace. It was relatively easy to troubleshoot the MARC hardware by simply setting the logic analyzer to trigger on specific microcode addresses and Data Bus contents and then starting the program execution at specific location with registers and memory in the desired state. The minicomputer development system was also used to simulate external hardware during testing of the MARC and MARC software.

**Detailed Hardware Description**

The MARC is microcode controlled. A 40 bit microcode word out of the microcode ROM (RAM during development) is divided into fields as shown in Fig. 3. The bits in each field, with a minimal amount of gating, control the various functions of the MARC, as indicated. The MARC contains four AMD 2901’s in its central processor unit (see Fig. 4). The AMD 2901 has 16 general purpose registers and one Q register.

The microcode control section of the MARC contains two AMD 2909 program control chips. They provide 8 bits of microcode address. Two additional bits are also provided for page control. A multiplexor in the AMD 2909 allows for four selections for the next microcode address. Microprogram control is accomplished by specifying a Jump, Call, Return or Fetch while selecting one of the FLAGS. If the FLAG is true, the Jump, Call, Return or Fetch is executed. If not, the program continues to the next ROM address.
**Fig. 3** MICROCODE FIELD SUMMARY

<table>
<thead>
<tr>
<th>11 BITS</th>
<th>AMD 2901 CONTROL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>6 BITS</strong></td>
<td>CARRY</td>
</tr>
<tr>
<td>A ADDRESS</td>
<td>BUS</td>
</tr>
<tr>
<td><strong>3 BITS</strong></td>
<td>B ADDRESS</td>
</tr>
<tr>
<td><strong>2 BITS</strong></td>
<td>SEQUENCE CONTROL</td>
</tr>
<tr>
<td><strong>10 BITS</strong></td>
<td>MICROCODE ADDRESS</td>
</tr>
<tr>
<td><strong>4 BITS</strong></td>
<td>STROBES</td>
</tr>
<tr>
<td><strong>4 BITS</strong></td>
<td>FLAG SELECT</td>
</tr>
</tbody>
</table>

* MULTIPLE USE BITS SELECTED BY AMD 2901 CONTROL BITS.

**Fig. 4** MARC DATA FLOW BLOCK DIAGRAM
Vectored interrupts are accomplished using a multiplexor on the least significant four bits of the D input to the AMD 2909. The low order 4 bits are multiplexed between the normal D inputs and the input from a priority encoder. If the upper order 6 bits from the microcode ROM are all one, the multiplexor is shifted to the priority encoder, and 1 of 16 addresses is selected depending upon which interrupt is present.

The lines labeled MDS on the left hand side of the diagram stand for Minicomputer Development System. These lines allow the minicomputer development system to control the ROM address and pass data in and out of the Data Bus. During development, the microcode memory is a RAM rather than a ROM. The MDS lines are used for loading this microcode RAM, and also for loading the CPU registers and the main memory.

The program counter (as distinct from the microprogram) is kept in general purpose register 2 in the AMD 2901’s. During a normal fetch cycle, the program counter contents are passed to the memory address register; at the same time the program counter contents are incremented and placed back in register 2. The contents of the specified memory location are then transferred to register 1 in the AMD 2901 and to the instruction register. The upper order 8 bits of the instruction register go to the AMD 2909’s, and the lower order 8 bits provide the B and A addresses. The microcode then performs a fetch microinstruction by taking the next ROM address from the upper order 8 bits of the instruction. It then begins executing the microcode to implement the specified instruction. The instruction is placed in register 1 to allow part of the Instruction Field to be used as a modifier.

**Interrupt Programming**

The MARC has 15 vectored interrupts. Interrupts are only allowed when the microprogram fetches a new instruction. In fetching an instruction, the microprogram goes thru location 3F0. This location is hardwired to branch automatically to one of 15 locations (3F1 → 3FF) if an interrupt is pending. These locations contain jumps to the appropriate microcoded interrupt handlers. The individual handlers can either process the interrupt entirely or jump to a handler in main memory. A 16 bit hardware interrupt mask allows the user to selectively enable and disable any or all of the 15 interrupt lines. Interrupts are disabled by setting the hardware mask to all one’s. They are enabled by returning the mask to the original condition using a software copy of the mask word.

**Input/Output Programming**

The MARC has both parallel and serial interfaces. The parallel interfaces are 16 bits wide. Two instructions (WORDI, WORDO) are available to work with them. These instructions allow for full word transfers to or from a GPR. The serial interface is a multiplexor with
eight I/O ports, the CHARI, CHARO instructions provide 8 bit transfers to and from a GPR to this interface. Since it is a multiplexor, the I/O port number has to be specified as well as the device code. In addition to 1 word or 1 byte programmed I/O transfers, there are block transfers that perform like a software DMA. These block transfers are microprogrammed and allow a maximum data rate of 1 I/O word per instruction fetch. This is because transfers are interrupt driven and interrupts are acknowledged only on an instruction fetch. The DMAIN and DMOUT instructions are 2 word instructions which specify the device code, the number of words to transfer, and the starting address of the data block to be transferred. This address can be indirect.

**Microcode Bit Assignment Details**

As noted above, the microcode ROM has a 40 bit output. The 40 bits are assigned as follows: (see Fig. 3) and Appendix A. 10 bits are assigned for microcode branch addresses or microcode immediate data. 11 more bits control the AMD 2901 operations: three of these bits select the function (add, subtract, etc), three bits select the source of the data, (A, B, O, or D) three bits select the destination of the data, (B, Q, B shift, etc). and two control the clocks to allow byte operations. Two bits are used for sequence control. These select the choice of Jump, Call, Return and Fetch in the microcode. Four bits are used for FLAG selection allowing for 16 different branch criteria, including an unconditional branch and a don’t branch. Four bits are used to control strobing of data from the Data Bus for up to 15 different destinations plus one no-op. Three bits are used for specifying the B register address. Of the remaining six bits, four are used for the A register and two for the carry control. The A bits serve several functions. They are used for “effective-flag” selection which allows the main program easy access to the branch control. Whenever the A register is not being used in an ALU instruction, the four bits normally used for A can be used for Bus control. The six bits which are normally Carry and A are also shared for shift control, since Carry and A are not used during shifts. The six bits then select the source for both the Q and the B register. The cleverness in reusing microcode bits only slightly affected the efficiency of the MARC, and saved a considerable amount in power required for microcode memory. However, a processor in which power is not a prime issue, it would be a good idea to have a wider microcode field and avoid a lot of worry about potential conflicts between various instructions.

**Microcode Instructions**

Microcode instructions could be defined in several different ways. We have chosen to define four categories of instructions: ALU instructions, Data Moving instructions, Branch instructions, and Clock control. There are really only eight ALU instructions when defined according to function. They are Add, Subtract (S-R), Negative Subtract (R-S), OR, AND (R • S), Mask (R • S), Exclusive OR, and Exclusive NOR. For the arithmetic Instructions
the following modifiers apply: Source, Destination, Carry, A Address, B Address and Strobe. For Logical instructions, the Carry modifier is omitted. There are several variations of these eight instructions for ease of programming. For example, Add has two other definitions, INC which increments one General Purpose Register and puts the result on the Bus and INCR which allows a Register to Register transfer of one incremented value. The Data move instructions Load and Store are special cases of the OR ALU instruction.

There are eight Branch microinstructions: Call, Fetch, Return, JMP and the same four with the effective flag specified. Call and JMP require a specified address, Fetch and Return do not. All branch instructions require the specification of a flag, and execute if the specified flag is true. The clock instructions are used for Byte manipulation in the ALU. The microcode instructions are defined by the OPDEF’s in Appendix A.

**Main Code Instructions**

In addition to a full complement of register-to-register arithmetic and logical instructions (including a microcoded multiply and divide), the MARC has double precision instructions to add, subtract, load, store and shift. Also available are instructions to add or subtract the contents of memory locations to registers and store the result in either register or in memory. There are one-word immediate instructions which add or subtract 0-15 to a general purpose register (GPR).

There is a move block (MOVBF) instruction that moves data from one part of memory to another. The MARC instruction set was not modelled after that of any other particular machine. It combines some of the best features of the PDP-11 and NOVA instructions sets. Generally, one MARC instruction can do anything either of these popular machines can do in one instruction, plus some things neither can do, such as double-word operations, single-bit operations, and block move.

**MARC Instruction Format**

The MARC’s instruction set is composed of single word (16 bit) and double word instructions, each containing an 8 bit opcode. The 8 bit opcode allows for a maximum of 256 distinct instructions. Because the instruction set was designed for a flight system in which memory space is limited and power consumption is a prime concern, there was an effort to make as many single word instructions as possible. The single word instructions referring to memory either have an address entirely in an index register (that may automatically be incremented or decremented) or they have an 8 bit address that is either absolute (0-255) or relative to the program counter (PC ± 128). In addition, there are single word immediate data instructions for add, subtract, and load byte (left or right, clear other half or not).
Because the instruction set is microcoded, it is flexible and not cast in concrete. There are about 40 unassigned instructions and others that could be given up depending on the needs of the project. There is a tradeoff between microcode memory (40 bits wide, must be high speed) and main memory (16 bits wide, can be a variety of speeds). However, if a group of instructions is used often, it may be worth microcoding them as one instruction.

Since our project required the MARC to maintain large status tables, bit manipulation instructions were very important. We have instructions to set, clear and test any single bit in memory tables and in registers. The memory tables can contain up to 32768 bits. The MARC Instruction set is included in Appendix B.

**Registers**

The MARC has 16 (16 bit) registers, 9 of which are general purpose registers (CPR) that can be used in any of the following ways:
1. Accumulators for keeping the results of arithmetic and logical operations.
2. Address Registers that can be automatically incremented or decremented.
3. Index Registers where the contents are added to the word following the instruction to determine the effective address.

The remaining 7 registers are used for the program counter, the stack pointer, and scratch registers for the microprogram.

**Addressing Modes**

Because of the need to keep the program size small there are a variety of addressing modes, many needing only one word to describe the address. Where these are inadequate, two-word instructions are available. The addressing modes are as follow:

**Single Word Instructions**

- Page Zero - contains an 8 bit absolute address (0-255).
- Relative to Program Counter - contains a two’s complement 8 bit displacement that is added to the PC to determine the effective address
- Register indirect - full 16 bit address contained in a designated CPR.

**Double Word Instructions**

- Absolute Direct 16 bit address (0-32767)
- Absolute Indirect
- Relative to Index Register 16 bit address is added to the contents of a CPR to determine the effective address
- Relative to Index Register Indirect
Up to 4 levels of indirection are allowed. All instructions are either register-to-register or register-to-memory. Except for the move block instruction, which is memory-to-memory. Except for a full word load-immediate, all double-word instructions are memory-to-register operations where the second word is a 16 bit address. These instructions allow for direct addressing of 32768 words. By adding an X to the instruction mnemonic, an index register can be added to the 16 bit address to get an effective address. By adding an I to the mnemonic, the effective address becomes an indirect address.

**Branch Control**

The MARC has a hardware condition code register containing the results of arithmetic, logical, shift and “load and test” instructions. The MARC can branch on the condition of the last result as follows: zero, non-zero, negative, positive, greater than zero, less than or equal to zero, carry = 1, carry = 0, overflow, no overflow, bit shifted out = 1, bit shifted out = 0, and unconditional.

There are three instructions that test these conditions, jump (JMP), jump and store return (JSR), and return (RET). JMP(X)(I) and JSR(X)(I) are two word instructions containing a 16 bit address plus an index register if the X suffix is used. The effective address can be indirect if the I suffix is used. If the specified condition is true, a jump to the effective address is executed. In addition, the JSR instruction will push the old value of the program counter onto the stack before jumping. The RET on condition is a one word instruction that, if the specified condition is true, will pop the return address off the stack and add a specified displacement to it (0-15), storing the result in the program counter.

In addition to the jumps on condition, there are two instructions JIZ(X)(I), JDZ(X)(I) that will increment or decrement a CPR and will jump to the effective address if the result is zero.

Besides jump instructions, there are test and skip instructions that allow the testing of bits in registers or in tables in memory and skipping the following two locations if the bit is in the selected state (0 or 1).

**Stack**

Register 3, the Stack Pointer (SP), contains the address of a stack in main memory. The Stack expands and contracts as data is written onto it and read out of it. There are 8 instructions that affect the Stack: JSR, RET, SAVRG, RSTRG, SAVCC, RSTCC, SVALL, RSALL. The JSR and RET are described under branch control. The Save registers (SAVRG) and restore registers (RSTRG) instructions will push onto the stack and pop off the stack a specified number of registers starting with a specified GPR. The
SAVCC and RSTCC saves and restores the condition code on the stack. The SVALL, RSALL, will save and restore the contents of all 9 GPR’s and the condition code at an address contained in a register. That register can be the stack pointer (R3) or any other GPR. The Stack register must be set up by the main program before using any of the above instructions.

**Micro-Assembler**

In order to microprogram the MARC we needed a general purpose assembler that would allow us to define an instruction set and then use multiple instructions from this set to define a single microprogram word. We wrote this assembler to run on our large IBM 370 time-sharing computer. It allows word sizes up to 64 bits and numbers can be entered in decimal, octal, hex, or binary.

To define in the instruction set, the OPDEF command is used. It defines the bit fields associated with an instruction. There are five types of field specifiers.
- #F(I) defines a fixed field of length # bits
- #V(I) defines a variable field of length # bits.
- #X defines don’t care bits
- #T causes the next field to start at bit position

**Example**

```plaintext
ADD OPDEF 7T,3V(110'B),1X,4F(E'H),4T,3N
JUMP OPDEF 3F(1),11X,4V,4N(5),2F(2)
```

Using the above definitions, the microinstruction

ADD 1, 2 and JUMP FETCH

would be encoded as follows. (Assumes FETCH is at location 2).

```
0 0 1 1 0 1 0 0 1 X 1 1 1 0 0 0 1 0 0 1 0 1 1 0
```

1st 2nd 1st fixed 1st default 2nd
fixed var var field var 2nd fixed
field add add add jump var field
jump

Note that the 2 opeodes in the example have no conflicting fields. If they did, an error message would result when they were used together. If none of the opcodes in an instruction fill the don’t care bits, they are padded with zeros or with the master default word if one has been defined.
The assembler outputs an absolute core-image file that can be transferred over the IBM 370 time share terminal lines to a DATA I/O PROM programmer or to the development system for RAM loading. Also output is a listing file with a concordance map.

**Testing**

A breadboard version of the MARC and critical portions of the signal processing system including the group demodulator and COP were tested as a system to prove feasibility. The MARC was tested but not extensively operated in the programmed modes.

The microcode ability of the MARC was tested by using it to control the time and frequency tracking for the demodulator. The demodulator provided accumulator contents for up to 16 channels to the MARC. The MARC would convert in-phase and quadrature amplitude to phase, subtract phase due to modulation, and accumulate the phase error to determine frequency offset. It also measured differences in phase on specific data combinations to measure timing offsets. The demodulator was capable of providing an output to the MARC a maximum once every 25 microseconds. The MARC software and hardware to provide this function were tested by providing a simulated input from the minicomputer development system. Inputs were taken from a simulation of the demodulator which was run on the IBM 370 and placed in a file. This file was transferred by the time share line to the disk in the minicomputer development system. Then the system provided approximately real time outputs to the MARC for testing the time and frequency tracking algorithms. We were able to determine the performance of these algorithms before they were actually hooked to the demodulator hardware. The MARC was tested with the demodulator and performed the time and frequency tracking task as expected. In a flight system the time and frequency tracking task would be performed on a special purpose microcoded device under the control of the MARC, using the algorithms developed on the MARC. Special purpose hardware would be more power efficient for this computational intensive task and this would avoid using up large quantities of MARC time which would be required for the control task.

**Conclusion**

The design of the MARC has demonstrated the feasibility of flying a processor with minicomputer capability in a Communication Satellite. The MARC has a full, versatile complement of all of the popular instructions plus some less common instructions which are tailored to the Control Job. These special instructions include Single bit instructions and Block move instructions. The MARC has 9 General Purpose Registers available to the programmer and is easily microcoded to provide special instructions and I/O handlers for speed intensive jobs.
Acknowledgments

We would like to acknowledge the assistance of B. H. Hutchinson, Jr., G. P. Gagnon and D. C. Rogers in the design of the MARC and preparation of this paper.

References


Appendix A
Field Assignments

<table>
<thead>
<tr>
<th>FUN</th>
<th>FUNCTION</th>
<th>DS</th>
<th>DESTINATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R PLUS S *</td>
<td>0 Q</td>
<td>Q REG &amp; F ON BUS</td>
</tr>
<tr>
<td>1</td>
<td>S MINUS R</td>
<td>1 FY</td>
<td>NONE &amp; F ON BUS</td>
</tr>
<tr>
<td>2</td>
<td>R MINUS S</td>
<td>2 BUS</td>
<td>B REG &amp; A REG ON BUS</td>
</tr>
<tr>
<td>3</td>
<td>R OR S</td>
<td>3 B</td>
<td>B REG &amp; F ON BUS</td>
</tr>
<tr>
<td>4</td>
<td>R AND S</td>
<td>4 BQR</td>
<td>B &amp; Q REG SHIFITED RIGHT F ON BUS</td>
</tr>
<tr>
<td>5</td>
<td>R AND S</td>
<td>5 BQL</td>
<td>B &amp; Q REG SHIFITED LEFT F ON BUS</td>
</tr>
<tr>
<td>6</td>
<td>R EXOR S</td>
<td>6 BQL</td>
<td>B &amp; Q REG SHIFITED LEFT ON BUS</td>
</tr>
<tr>
<td>7</td>
<td>R EXOR S NOT</td>
<td>7 BL</td>
<td>B REG SHIFITED LEFT F ON BUS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SO</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AQ</td>
</tr>
<tr>
<td>1</td>
<td>AB</td>
</tr>
<tr>
<td>2</td>
<td>ZQ</td>
</tr>
<tr>
<td>3</td>
<td>ZB</td>
</tr>
<tr>
<td>4</td>
<td>ZA</td>
</tr>
<tr>
<td>5</td>
<td>UNUSED</td>
</tr>
<tr>
<td>6</td>
<td>DQ</td>
</tr>
<tr>
<td>7</td>
<td>DZ</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FL  FLAG DEFINITION
0  ZF  ZERO FLAG (FROM PREVIOUS CLOCK PULSE)
1  UNC  UNCONDITIONAL
2  C  CARRY (FROM PREVIOUS CLOCK PULSE)
3  S  SIGN (FROM PREVIOUS CLOCK PULSE)
4  OV  OVERFLOW (FROM PREVIOUS CLOCK PULSE)
5  BF  B FLAG (FROM PREVIOUS CLOCK PULSE)
6  CNT  CONTINUE
7  QF  Q FLAG (FROM PREVIOUS CLOCK PULSE)
8  SZF  STRD ZERO FLAG
9  SBF  STRD B FLAG
10  SS  STRD SIGN FLAG
11  SGEZ  STRD GTR OR EQ TO ZERO FLAG (POS)
12  SNZ  STRD ZERO FLAG IS ZERO
13  BZ  B FLAG IS ZERO (FROM PREV CL OUTPUT)
14  SLEZ  STRD LESS THAN OR EQ TO ZERO FLAG
15  EF  EFFECTIVE FLAG (EXT FLAG)

EF  EFFECTIVE FLAG (EXTERNAL FLAG)
0  ZF  ZERO FLAG *
1  UNC  UNCONDITIONAL
2  C  CARRY
3  S  SIGN (NEGATIVE) (LESS THAN ZERO)
4  OV  OVERFLOW
5  BF  B FLAG (RAM SHIFTED OUTPUT)
6  GZ  GREATER THAN ZERO
7  SW  SENSE SWITCH
8  NZ  NOT ZERO
9  CONTINUE
10  CZ  CARRY IS ZERO
11  GEZ  GREATER OR EQUAL TO ZERO
12  VZ  OVERFLOW IS ZERO
13  BZ  B FLAG IS ZERO (RAM SHIFTED OUTPUT)
14  LEZ  LESS OR EQUAL TO ZERO
15  SWZ  SENSE SWITCH IS ZERO

BSH  B SHIFT
0  Z  ZERO *
1  ONE
2  C  PRESENT CARRY
3  S  PRESENT SIGN
4  BS  B SHIFT (PRESENT CYCLE)
5  BD  B DELAYED (FROM PREV CLK CYCLE)
6  QS  Q SHIFT (PRESENT CYCLE)
7  QD  Q DELAYED (FROM PREV CLK PULSE)

QSH  Q SHIFT
(SAME AS BSH)

REG A ADDRESS
0  A  MICROCODE REGISTER ONLY *
1  PC  PROGRAM COUNTER
2  SP  STACK POINTER
3  AE  GENERAL PURPOSE REGISTER
4  B  GENERAL PURPOSE REGISTER
15  B  REG A ADDRESS EXTERNAL SOURCE

SC  SEQUENCE CONTROL
0  CALL  CALL MICROCODE SUBROUTINE
1  FET  FETCH AN INSTRUCTION
2  RET  RETURN FROM MICROCODE SUBROUTINE
3  JMP  JUMP

S  STROBE
0  S  NO STROBE
1  IR  LOAD INSTRUCTION
2  MA  LOAD MEMORY ADDRESS REG
3  M  INCREMENT MEMORY ADDRESS REG
4  M  WRITE MEMORY
5  ABI  INCREMENT EXTERNAL A & B INST
6  ABD  DECREMENT EXTERNAL A & B INST
7  LC  LOAD CONDITION CODES
8  SC  SET CONDITION CODES
9  LM  LOAD I/O MASK (0 ENABLE, 1 DISABLE)
10  IO  WRITE I/O (A FIELD SELECTS I/O DEVICE)
11  DM  WRITE DINAUX DEVELOPMENT SYS INTR
12  HLT  HALT MARC
13  LCH  LOAD SERIAL I/O CHANNEL NUMBER
14  SF  STORE FLAGS

BC  BUS CONTROL
0  CPU  CENTRAL PROCESSOR UNIT*
1  UDR  MICRO DATA RIGHT 10 BITS FROM ROM LEFT
       6 BITS ZERO
2  PNL  FRONT PANEL SWITCH
3  MSK  MASK (A EXTERNAL FIELD SELECTS ZERO BIT)
4  M  READ MEMORY
7  CC  READ CONDITION CODE
10  IO  READ I/O
11  DM  READ DINAUX

* Default
Appendix B

MARC INSTRUCTION SET

ARITHMETIC

*ARM(X)(I) RB,RX,M Add RB to memory
*SBRM(X)(I) RB,RX,M Subtract RB from memory
*ARM(X)(I) RB,RX,M Add memory to reg
*SBRM(X)(I) RB,RX,M Subtract memory from register
ADDA RB,RA RB=RA+RB
SUBA RB,RA RB=RB-RA
MUL RA,RA Multiply RA by RA
DVA RB,RA RB=RA+1
DCA RB,RA RB=RB-1
DADD RB,RA (RB,RA+1)=(RB,RA+1)+(RA,RA+1)
ADDV RB,θ RB=RB+θ
SUBV RB,θ RB=RB-θ

LOGICAL

OR RB,RA RB=RB RA
AND RB,RA RB=RB RA
COM RB,RA RB=one's complement of RA
NEG RB,RA RB=two's complement of RA
XOR RB,RA RB=RB RA
TSTR RB set condition code based on RB
RCP RB,RA Compare RA to RB, set CC
*MCP(X)(I) RB,X,M Compare RB to memory. Set CC.

DATA MOVING

#LDM(X)(I) RB,RX,M load RB with contents of memory
#DLM(X)(I) RB,RX,M double load RB,RA+1 from memory
LDIR8 RB,θ RB=RB RA
LDL(B) RB,RM load RB from page zero addr M
LDMR RA,RM load RB from memory (2's comp)
LDR RA,RX load RB from address in RX
LDMX RX,RM load RB from memory (2's comp)
LDLB RX,RM load left byte of memory into RX
LDRL RX,RM load right byte of memory into RX

STORE REGISTER

*STM(X)(I) RB,RX,M store from RB to memory
*DST(X)(I) RB,RX,M double store RB,RA+1 to memory
STPZ RB,M store RB to page zero addr M
STRL RB,M store RB to PCM+M (+128)
STRX RB,RX store RB to address in RX
STIX RB,RX same as STX except increment RX
STDX RB,RX same as STX except decrement RX
STLB RB,RX store RB to left byte memory(RX)
STRB RB,RX store RB to right byte memory

BLOCK MOVES

MOVBF RB,RA move from addr in RB to addr in RB+1, RA=θ words
SVALL RA save all GPR's and CC in memory, pointed to by RB.
RSALL RB Add registers on stack, starting with RB
SAVRC RB,θ save θ registers on stack, starting with RB
RSTRG RB,θ restore θ registers from stack, starting with RB
SAVCC  save condition code on stack
RSTCC  restore CC from stack

SHIFTS

SLA  RB, #  arith left shift RB, #bits
SLL  RB, #  logical left shift RB, #bits
SRA  RB, #  arith right shift RB, #bits
SRL  RB, #  logical right shift RB, #bits
DCA  RB, #  arith left shift, RB, RB+1, #bits
DLL  RB, #  logic left shift, RB, RB+1, #bits
DRA  RB, #  arith right shift, RB, RB+1, #bits
DRL  RB, #  logic right shift RB, RB+1, #bits
SLV  RB, RA  logical left shift RB, RA= #bits
RRR  RB, #  rotate right RB, # bits
RRL  RB, #  rotate left RB, # bits
SWAPB RB, RA  swap bytes of RA, store in RB

BIT MANIPULATIONS

*SBIT(X)(I) RB, RX, M  set bit in table, RB= bit addr,
      X+M = table address
*CBIT(X)(I) RB, RX, M  clear bit in table. See above
RSETB RB, #  set bit # of RB=1
RVLRB RB, #  clear bit # of RB

PROGRAM COUNTER CONTROL

*JMP(X)(I) CC, RX, M  if CC true, jump to EA
*JSR(X)(I) CC, RX, M  if CC true, jump to EA and
      store PC on stack
*JIZ(X)(I) RB, RX, M  RB= RB+1, if RB=0, jump to EA
*JIZ(X)(I) RB, RX, M  RB= RB-1, if RB=0, jump to EA
*SBM  RB, RX, M  skip next two locations if bit=1
      RB= bit addr, X+M= table address
*SKBZM RB, RX, M  skip if bit in table is zero
*SKBSC RB, RX, M  skip if bit=1 and clear bit
SKBZ  RB, #  skip if bit # of RB is zero
SKBSR RB, #  skip if bit # of RB is set
HALT  halt MARC
RET  CC, #  return if CC true, pop PC off
      stack, set PC=PC+#

INPUT/OUTPUT

WORDO RB, #  output RB(16 bits) to device #
WORDI RB, #  input to RB from device #
CHARO RB, RA  output byte from RB to device
      and port specified in RA
CHARI RB, RA  input byte to RB from device
      and port specified in RA
INTEN  master enable interrupts
INTDS  master disable interrupts
SMM  RB  interrupt mask=RB and enable
DMOUT(I) RB, #, M  DMA out, #=device, M=buffer addr,
      RB=number of words
DMAIN(I) RB, #, M  DMA in

Note:
RA, RB refer to general purpose registers
RX is a GPR used as an index register
M refers to a memory address
CC is the condition code
EA is the effective address
* are 2 word instructions