BINAR™ Technical Reference

Preliminary - Version 0.0

March 1, 1990

CONTENTS

1.0 INTRODUCTION

This document is preliminary Version 0.0. This version is the working document used by Harris engineers, and is not necessarily complete and accurate in all respects. If you have any questions or doubts about the technical content of this document, please contact Harris for clarification. A more comprehensive and coherent document to supersede this version is currently in preparation.

The BINAR Technical Reference has been developed to provide system design and programming information to assist the hardware engineer in evaluating the Harris Semiconductor 32-Bit BINAR Core Processor chip.

Technical information included here addresses BINAR Core Processor hardware and firmware, macrocode and microcode programming, the BINAR Core Processor Evaluation Board, and the BINAR Forth Microcode Simulator Environment. Detailed hardware schematics and test vector generation are not included in this volume.

The reader is understood to possess a knowledge of stack-oriented computer operation theory (including architectures involving the use of Data and Return Stacks), familiarity with microcoded CPU control concepts and techniques, and some experience with the Forth programming language. Although additional programming environments are under development as of this writing, the material presented here has been formulated to support use of the BINAR Core Processor Evaluation Board, the TIL Forth Language Compiler, and the BINAR Forth Microcode Simulator Environment operating in a host IBM-Compatible AT computer.

This manual has been designed for reference use. Accordingly, information is not generally presented in tutorial form. It is suggested that the reader preview the entire manual before attempting to understand any single aspect of processor operation in detail.

The term "BINAR" is a Harris Semiconductor trademark. All information contained within this BINAR Technical Reference is Harris Semiconductor Proprietary.

1.1 NOTATION AND TERMINOLOGY CONVENTIONS

The BINAR Technical Reference conforms to the following assignment and notation conventions:

BINARY EXPRESSIONS

- A BYTE is eight bits.
- A HALFWORD is sixteen bits.
- A WORD (sometimes called a FULLWORD) is thirty-two bits.
- Bit '0' is the Least Significant Bit (LSB) of a word.
- Bit '31' is the Most Significant Bit (MSB) of a word.

INTEGER EXPRESSIONS

- Numbers are decimal expressions except where identified as hexadecimal.

- Hexadecimal representations are identified with a prefixed "$" symbol, or are written in the form: "__ Hex", e.g., "35 HEX".

TERMINOLOGY

CLOCK CYCLE/MICROCYCLE: The term CLOCK CYCLE, or MICROCYCLE, refers to that time interval which commences with the occurrence of a CLOCK signal rising edge and terminates with the occurrence of the next CLOCK signal rising edge.

MEMORY CYCLE: the term Memory Cycle (sometimes called a BUS CYCLE) refers to that time interval which commences with the beginning of a clock cycle during which an address is asserted on the memory bus and terminates at the end of the clock cycle during which the bus operation is completed. In many cases of interest, a bus cycle is two clock cycles in length.

SIGNALS AND CLOCKS: In discussions which follow, BINAR signals are uniformly characterized as ACTIVE-HIGH, and clocks as RISING-EDGE. Although the BINAR Evaluation Board actually employs both active-low signals and falling-edge clocks, this convention has been adopted to facilitate discussion of processor operation. The inverted active-low signals are identified in hardware documentation by the assignment of signal names with an appended "#" symbol, e.g. "FRAM#".

SET and **CLEAR:** Where the terms SET and CLEAR are employed to refer to the status or value of a bit field in a data structure, SET signifies the assignment of bit value 1 and CLEAR, bit value 0. The word PRESET is used when a bit field is loaded with a mixture of 1 and 0 bit values.

The terms MICROASSEMBLER and MICROCODE ASSEMBLER; and MICROCODE SIMULATOR and SIMULATOR are used interchangeably.

1.2 BINAR TECHNICAL REFERENCE OVERVIEW

The following paragraphs provide a synopsis for major sections of the BINAR Technical Reference.

SECTION 1, INTRODUCTION: Introduces the BINAR TECHNICAL REFERENCE, defines notation and terminology conventions, and provides this overview.

SECTION 2, THE BINAR SYSTEM: Provides an overview of the BINAR System, including performance characteristics, packaging, instruction formats, and fundamentals of system operation.

SECTION 3, INTRODUCTION TO MICROCODING: Introduces BINAR microcoding through extensive use of examples and explanations correlated with machine architecture.

SECTION 4, HARDWARE THEORY OF OPERATION: Considers the BINAR system from a hardware perspective, and provides detailed discussions of machine implementation and operation.

2.0 THE BINAR SYSTEM

The 32-Bit BINAR Core Processor chip is the prototype processing core of what will, in a commercial version, be a single-package embeddable microcontroller design optimized for program execution speed, efficient coding, and minimum system complexity. The BINAR architecture provides for efficient utilization of program memory bandwidth without instruction pipelining. Execution of program instructions provides for overlapped accesses to stack memory, microcode memory, and program memory.

2.1 BINAR FEATURES AND ATTRIBUTES

A summary of BINAR features includes:

- 32-Bit Real-Time Control Stack Processing. Internal and external data paths are 32 bits.
- Stack Architecture. Data Stack (64 elements) and Return Stack (64 elements) are each implemented in hardware, and each benefits from stack bounds checking. Instructions are zero-operand, or one-operand with an embedded instruction literal value.
- Single-Chip Implementation: one 84-pin LCC.
- 16 MHz Operation (typical personal-computer operating conditions). This Performance is attained using 1.5 micron feature-size fully static CMOS technology; this is the same fabrication technology used on the RTX 2000 and Harris 80C286 processors.
- 32-Bit Fixed-Format Instructions. A single macroinstruction is capable of containing, alternatively: a single opcode, an opcode and a subroutine CALL, an opcode and a subroutine EXIT, or two opcodes. Both EXIT and Jump-To-Next instructions may include a 21-bit signed literal value within the instruction. The two-opcode (2OPS) format permits containment of two short signed literal values of 6 bits each. Opcode execution and subroutine threading are overlapped.
- One-Operation-Per-Clock-Cycle Performance. The two-opcode format permits execution of a two-opcode

instruction in two clock cycles. Subroutine CALL and EXIT instructions execute two operations (an opcode and a subroutine operation) in two clock cycles. Each memory bus operation occurs in two clock cycles.

- Overlapping Execution And Fetch. Both microinstruction and macroinstruction feature overlapping execution and fetch operation.

- Word-Aligned Instructions And Data. Memory is byte-addressable for both 8-bit data access and 16-bit data access (aligned on half-words).

- 32-Bit Linear Address Space For Data Accesses. Programs are organized in 8-Megabyte pages.

- Microcode RAM/ROM Mix: 2048 Words Of Microcode ROM (up to 256 opcodes); 128 Words Of Microcode RAM (16 opcodes) For User-Defined Opcodes. The architecture allows a RAM/ROM mix of up to 4K words (512 opcodes).

- ASIC (Application Specific Integrated Circuit) Bus Support. Design provides for use of Ram Data Bus pins as an I/O channel.

- Accessibility To Chip Internal Architecture. Design provides for use of Memory Data Bus as an access channel. This capability also provides unequalled accessibility for chip testing.

- C Programming Language Support. Efficient fast-literal, frame pointer-plus-offset addressing, and signed-character support.

- Instruction Compression. Typical stack operation instruction execution in one microcycle using the 20PS instruction format. Approximately 40% of all subroutine calls are executed "for free" in typical Forth programs. In a similar manner, almost all subroutine exits are "free". Microcoded opcodes permit more effective and flexible instruction compression than is attainable on hardwired machines (e.g., SWAP ! as a single instruction). Application-specific instructions in microcoded RAM may increase application program execution speed by a factor of 1.5 or 2.

2.2 BINAR CORE PROCESSOR BLOCK DIAGRAM

Figure 2-1 provides an architectural overview of the BINAR Core Processor, and introduces key terms and block functions which figure prominently in discussions throughout this manual.

2.3 BINAR INSTRUCTION FORMATS

An understanding of instruction formats is fundamental to understanding BINAR Core Processor operation. As Figure 2-2 illustrates, BINAR instruction formats are divisible into two basic categories: Macrocode formats and the single Microcode format.

2.3.1 MACROCODE INSTRUCTION FORMATS

There are four BINAR macrocode instruction formats. The first of these, 2OPS, is a two-opcode format. The remaining three, CALL, EXIT, and JNEXT (Jump-To-Next), are single-opcode formats. By definition, macrocode instruction formats apply to instructions that reside in program memory. They are the "machine code instructions" for the system.

In both two- and single-opcode formats, Bits 0-1 in the 2-bit control field specify and determine whether the instruction is to be interpreted as a 2OPS, a subroutine CALL, a subroutine EXIT, or a JNEXT sequential instruction.

In the single-opcode formats, the 9-bit opcode field specifies one of up to 512 possible stack-oriented operations. Because the BINAR Core Processor is microcoded, there is no one specific pattern in accordance with which opcodes uniformly perform their respective functions. At the hardware level, the opcode is used to form the high-order bits of the microcode memory address, thereby eliminating requirements for complex instruction decoding logic. In cases of subroutine CALLs, bits 2-22 hold bits 2-22 of the word-aligned memory address for the jump target. Bits 23-31 of the address are supplied by the PAGE register, which is initialized to zero upon system reset. Bits 0-1 of the address are forced to zero. In cases of a subroutine EXIT or JNEXT instruction, bits 2-22 hold a 21-bit signed literal that may be used by the opcode, if required (the literal field is shifted to the right two bits before use so that any integer value between -524288 and 54287 may be represented).

In the 2OPS format, bits 22-17 and 7-2 each hold 6-bit signed short literals for each of the two opcodes (the

Figure 2-1. Architectural Overview of the BINAR.

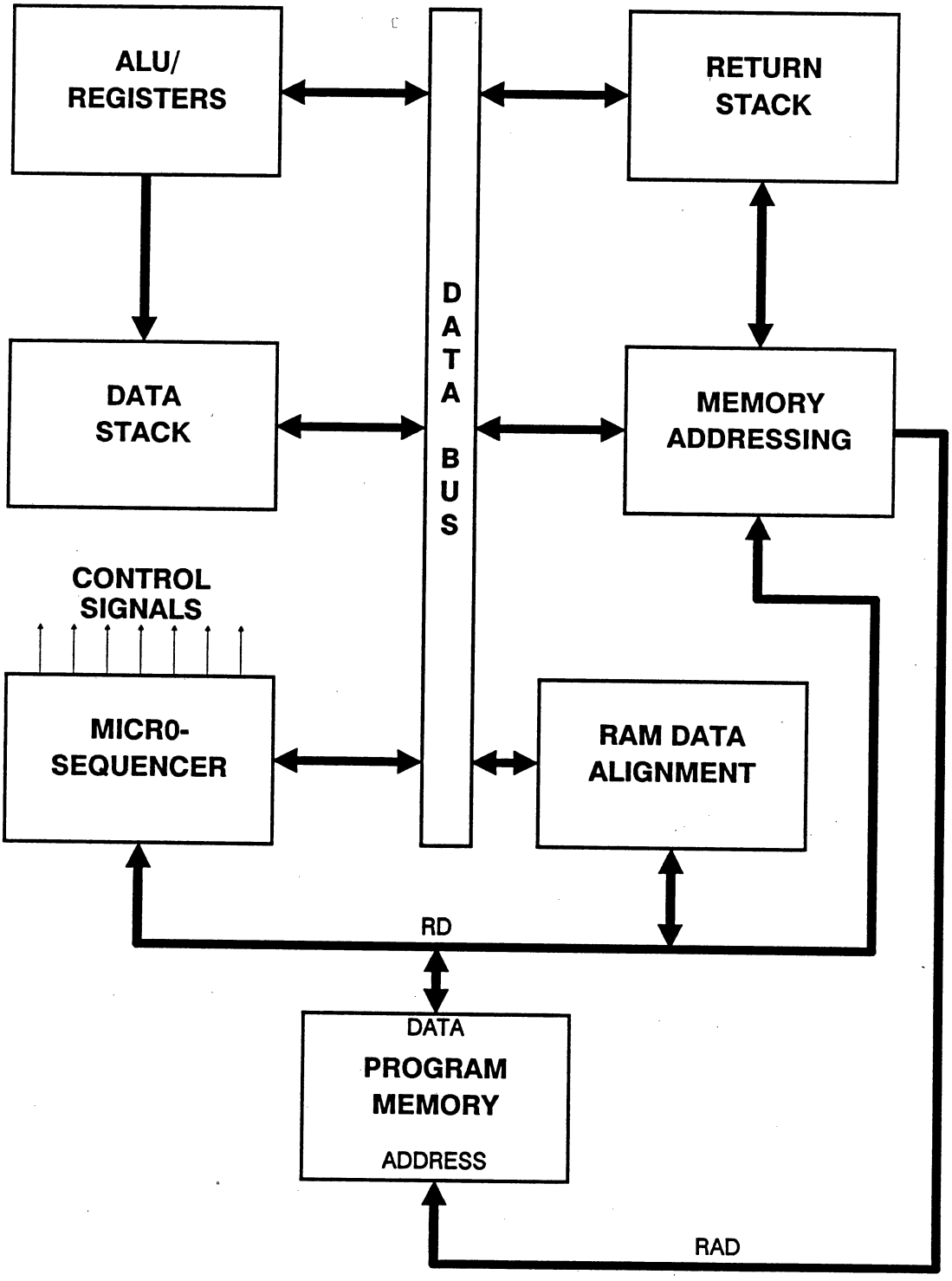


Figure 2-2. BINAR Instruction Formats.

CONTROL

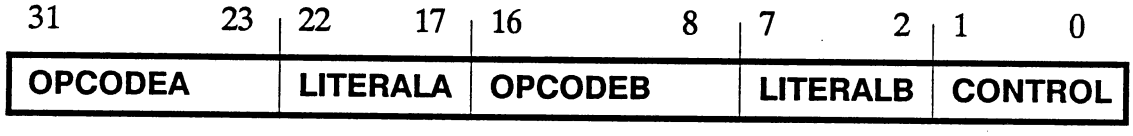
0	Subroutine Call (CALL)
1	Subroutine Return (EXIT)
3	Sequential Execution (JNEXT)

ADDRESS/LITERAL

subroutine address (word-aligned)	
sign-extended 21-bit literal	
sign-extended 21-bit literal	

OPCODE

0 to 511 Opcode value



CONTROL

2 Two Opcodes & Jump to Next (TWO-OPS)

OPCODEA & OPCODEB

0 to 511 Opcode values (A executed before B)

LITERALA & LITERALB

-32 to 31 Literal values for opcodes A and B

literal fields are extracted to permit representing any integer between -32 and 31).

The CALL and EXIT instructions make it possible to perform both an opcode and a control transfer (subroutine CALL, subroutine EXIT) in parallel. In all four instruction formats, The operation of fetching the next instruction from wherever it may be is overlapped with execution of the opcode. Thus, in the program source code, an opcode followed by a subroutine CALL or EXIT results in the instruction having achieved the subroutine CALL/EXIT "for free".

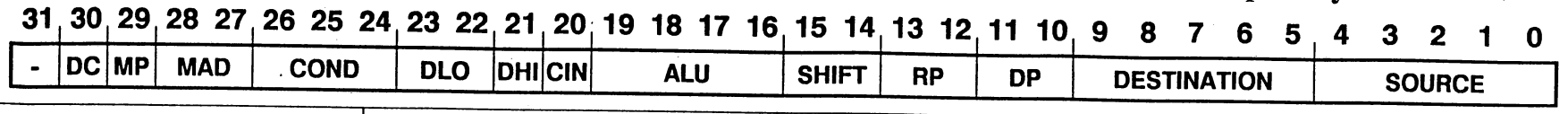
At the hardware level, the two opcodes in the 2OPS format pair are executed sequentially. There is a two-clock minimum imposed on opcodes used with other instruction formats because instruction fetches from program memory take a minimum of two clock cycles. However, opcodes in the 2OPS format are permitted to execute in one clock cycle because the fact that there are two of them guarantees that the two-cycle minimum instruction fetching time will be met, thereby yielding a maximum opcode execution rate of one opcode per microcycle. In the present chip implementation, this results in there being two versions of some opcodes, such as '+': a two-cycle version padded with a no-op microinstruction for use with CALL, JNEXT and EXIT instructions, and a one-cycle version for use with 2OPS instructions. Future chip iterations will require only one uniform opcode version, and hardware will insert a stall in non-2OPS instruction types.

2.3.2 MICROCODE INSTRUCTION FORMAT

By definition, microcode instruction format applies to control words that reside in microcode memory. They are the "microinstructions" for the system.

The BINAR format for microinstructions is shown in Figure 2-3, the BINAR Quick Reference. The format data structure is displayed horizontally at the top of the page and is shown to contain 14 independent fields. Also shown is a complete listing of all possible values for each of these fields with the corresponding Microassembler mnemonic for each value. The only exceptions to these listings are those for ALU functions, which omit the "ALU=" portion of the mnemonic and allow the inclusion of extra spaces for readability. Also, observe that although bit 31 has been implemented in microcode Ram, it is not currently used.

Figure 2-3. BINAR QUICK REFERENCE . • Harris Semiconductor Proprietary



DECODE CONTROL (30):
0 nop (default)
1 DECODE

INCREMENT MPC (29):
0 nop[MPC] (default)
1 INC[MPC]

MICRO-ADDRESS (27-28):
0 JMP=00x
1 JMP=01x
2 JMP=10x
3 JMP=11x

CONDITION CODE (24-26):
0 JMP=xx0 always 0
1 JMP=xxC Carry out
2 JMP=xxZ Zero
3 JMP=xxS Sign
4 JMP=xxL Lowest bit of DLO
5 JMP=xxV oVerflow
6 JMP=xxP Pending interrupt
7 JMP=xx1 always 1

DLO CONTROL (22-23):
0 nop[DLO] (default)
1 SR[DLO]
2 SL[DLO]
3 DEST=DLO

DHI (21):
0 DHI[0] (default)
1 DHI[1]

CIN (20):
0 CIN=0 (default)
1 CIN=1

ALU FUNCTION (16-19):

	CIN=0	CIN=1
0	A (default)	A
1	A or B	A or B
2	A + A	A + A + 1
3	-1	-1
4	A and B	A and B
5	B	B
6	A + B	A + B + 1
7	A + 0	A + 1
8	A - 1	A - B
9	A xor B	A xor B
10	not B	not B
11	A nand B	A nand B
12	0	0
13	A - B - 1	unused
14	A nor B	A nor B
15	not A	not A

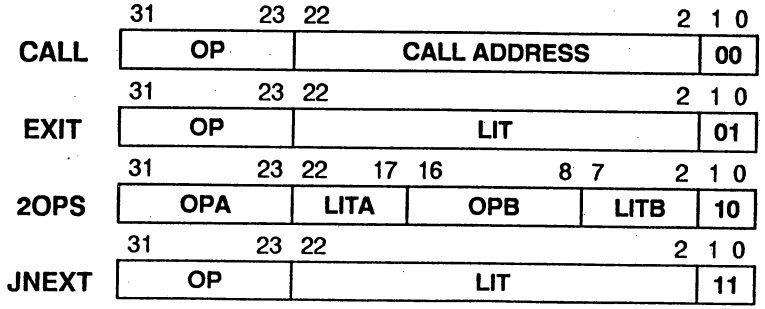
ALU SHIFT (14-15):
0 PASS[ALU] (default)
1 SR[ALU]
2 SL[ALU]
3 FPU (pass FPU output)

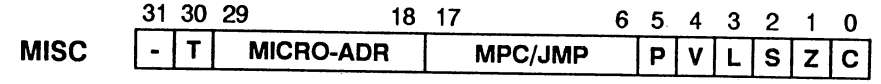
RP CONTROL (12-13):
0 DEC[RP]
1 INC[RP]
2 nop[RP] (default)
3 DEST=RP

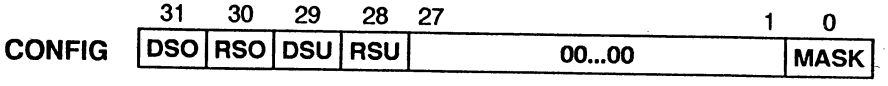
DP CONTROL (10-11):
0 DEC[DP]
1 INC[DP]
2 nop[DP] (default)
3 DEST=DP

DESTINATION (5-9):
0 none (default)
1 DEST=HOST
2 DEST=DS
3 DEST=DP-LIMIT
4 DEST=RS
5 DEST=RP-LIMIT
6 DEST=DHI[0]
7 DEST=DHI[1]
8 DEST=MRAM
9 DEST=MICRO-ADR
10 DEST=DS-FROM-DHI
11
12 DEST=SBASE
13 DEST=DBASE
14 DEST=FP
15 RAM-RMW
16 RAM-C@
17 RAM-W@
18 DEST=CONFIG
19 ASIC-@
20 DEST=RAM-C!
21 DEST=RAM-W!
22 DEST=RAM-!
23 DEST=ASIC-!
24
25 LATCH-INSTRUCTION
26 CYCLE-RAM
27 ADDR=BUS-4(CYCLE)
28 ADDR=BUS+0(CYCLE)
29 ADDR=BUS+SBASE(CYCLE)
30 ADDR=BUS+DBASE(CYCLE)
31 ADDR=BUS+FP(CYCLE)

SOURCE (0-4):
0 SOURCE=HOST
1 SOURCE=LIT
2 SOURCE=PAGE/NAR/CTL
3 SOURCE=ADDR
4 SOURCE=MRAM
5 SOURCE=MROM
6 SOURCE=SBASE
7 SOURCE=DBASE
8 SOURCE=FP
9 SOURCE=MISC
10 SOURCE=RP
11 SOURCE=DP
12 SOURCE=RS
13 SOURCE=RETURN-SAVE
14 SOURCE=CONFIG
15 SOURCE=I-LATCH
16 SOURCE=DHI[0]
17 SOURCE=DHI[1]
18 SOURCE=RP-LIMIT
19 SOURCE=DP-LIMIT
20 SOURCE=DLO
21 SOURCE=-1
22 SOURCE=0
23 SOURCE=4
24 SOURCE=unused-24
25 SOURCE=DS
26 MULTIPLY-STEP
27 DIVIDE
28 SOURCE=RD
29 SOURCE=RD-SIGNED
30 SOURCE=unused-30
31 SOURCE=unused-31







Additional operational details for each of these microcode formatting fields is provided in Section 3.0, INTRODUCTION TO MICROCODING.

2.4 INTRODUCTION TO CLOCK SIGNALS AND THE CLOCK CYCLE/MICROCYCLE

Figure 2-4, Clock Cycle Overview, shows an idealized representation of the BINAR processor two-phase clock. The following discussions introduce specific BINAR system clock signals to provide a foundation for subsequent references to system clock-dependent actions and events.

OSC AND CLOCK

The OSC signal is the frequency base introduced from the "outside world". This signal is applied to the OSC pin and provides the edges used to generate the system master CLOCK and, ultimately, all other clock signals internal to the BINAR Core Processor chip. Chip-internal CLOCK transitions occur on the rising edge of the OSC signal. Accordingly, OSC must always cycle twice for each system master CLOCK signal the processor chip is required to generate for system operation. The only exception to this requirement occurs when OSC is used for control while the chip is placed in TEST mode, which is described in Section 4.1.1.1, below.

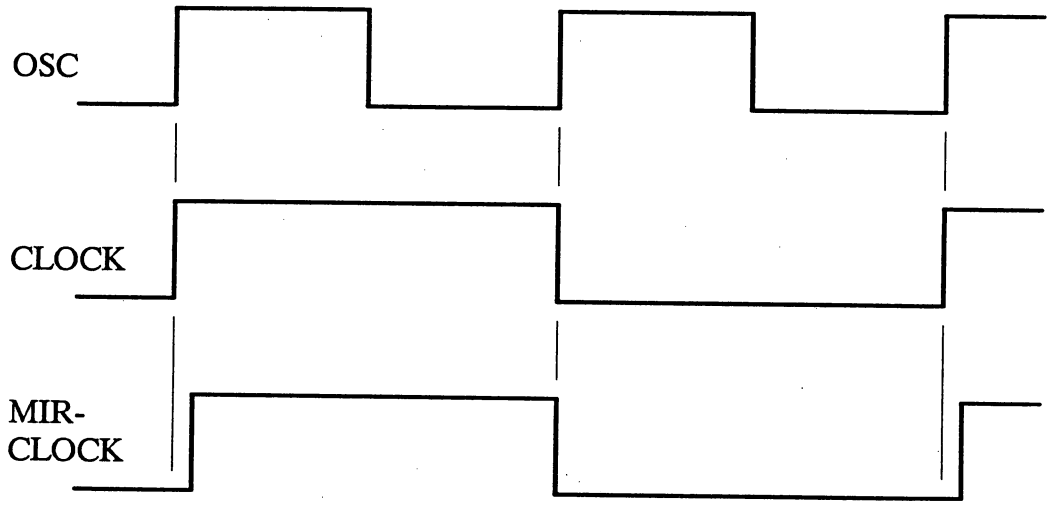
MIR-CLOCK

MIR-CLOCK is used by the microsequencer to load the microinstruction register. MIR-CLOCK is delayed slightly from CLOCK to provide assurance that the rising edge of CLOCK, when it occurs, always results in the capture of data into all registers before the microinstruction register changes control signals.

SINGLE-STEP MODE

The BINAR core processor chip may be operated in a single-step microcode mode. This is accomplished by first using the TEST pin to load a microcode word into the Single-Step Register, and then cycling the OSC pin twice to produce a CLOCK cycle. The internal microinstruction operations performed in single-step mode are identical with those for full-speed operation. The single-step mode is used extensively for demonstrating BINAR machine operation in Section 3.3, MICROCODING EXAMPLES, below and for chip testing.

Figure 2-4. Clock Cycle Overview



2.5 PACKAGING AND PINOUT

The BINAR Core Processor has been designed to fit within an 84-pin package. To achieve this compact packaging, only the lower-order 24 bits of RAM address have been pinned out. Package pin assignments are shown in Figure 2-5, BINAR Pinout Description. System functions for each pin group are discussed in detail in Section 4.1.1, below.

Figure 2-5. BINAR Pinout Description.

Pin #	Pin Name	Type	Description
1	VDD	Power	Double VDD pad
2	ASIC#	Output	Noninvert pad
3	INTA#	Output	Noninvert pad
4	FRAM#	Output	Noninvert pad
5	GND	Ground	Aux GND pad
6	NMI#	Input	TTL input pad
7	INTR#	Input	TTL input pad
8	WAIT#	Input	TTL input pad
9	OSC#	Input	TTL input pad
10	MEM#	Output	Noninvert pad
11	GND	Ground	Aux GND pad
12:19	RD<31:24>	BiDirect	TTL BiDirect pad
20	GND	Ground	Aux GND pad
21	VDD	Power	Aux VDD pad
22:29	RD<23:16>	BiDirect	TTL BiDirect pad
30	GND	Ground	Aux GND pad
31:38	RD<15:8>	BiDirect	TTL BiDirect pad
39	GND	Ground	Aux GND pad
40:42	RD<7:5>	BiDirect	TTL BiDirect pad
43	VDD	Power	Aux VDD pad
44:48	RD<4:0>	BiDirect	TTL BiDirect pad
49	GND	Ground	Aux GND pad
50:53	RAD<23:20>	Output	Noninvert pad
54	VDD	Power	Aux VDD pad
55:58	RAD<19:16>	Output	Noninvert pad
59	GND	Ground	Aux GND pad
60:63	RAD<15:12>	Output	Noninvert pad
64	VDD	Power	Aux VDD pad
65:68	RAD<11:8>	Output	Noninvert pad
69	GND	Ground	Aux GND pad
70:75	RAD<7:2>	Output	Noninvert pad
76	RES#	Input	TTL Schmitt Trigger pad
77	TEST#	Input	TTL input pad
78	CLK_OUT	Output	Noninvert pad
79	GND	Ground	Aux GND pad
80	OE#	Output	Noninvert pad
81:84	WR<0:3>#	Output	Noninvert pad

3.0 INTRODUCTION TO MICROCODING

3.1 SIMPLIFIED BINAR BLOCK DIAGRAM - FUNCTIONS OVERVIEW

Figure 3-1, Simplified Block Diagram, illustrates the general configuration of functional blocks within the BINAR chip.

DATA BUS

A central internal DATA BUS is used to transfer data between functional elements within the BINAR system. Although it is convenient in the discussions which follow to conceptualize these data transfers as occurring along a physical bus, this chip-internal data "bus" is actually a multiplexer implementation which links up to 32 possible sources with more than 32 possible destinations, all within the chip. Since these sources and destinations may be specified within microinstructions, and since all functional blocks interface with this central "bus", any register within the core processor may be directly written to or read from. In addition to providing optimum internal system configurability, this exhaustive approach to bus accessibility affords benefits for efficient program debugging and comprehensive chip testing.

DATA STACK (DS)

The DATA STACK (DS) functional block includes the stack RAM, stack pointer register, limit comparison logic, and stack buffer register. A special path from the ALU block to the DS block, which bypasses the Data Bus, is provided to facilitate efficient, single-cycle Forth SWAP and OVER operations.

ALU/REGISTERS

The ALU/REGISTERS block includes a 32-bit integer ALU and three working registers. DHI[0], by convention, is used to hold the top element of the programmer-visible data stack. DHI[1] and DLO are employed as scratch registers. The ALU and DLO registers may also be configured for 64-bit shift operations.

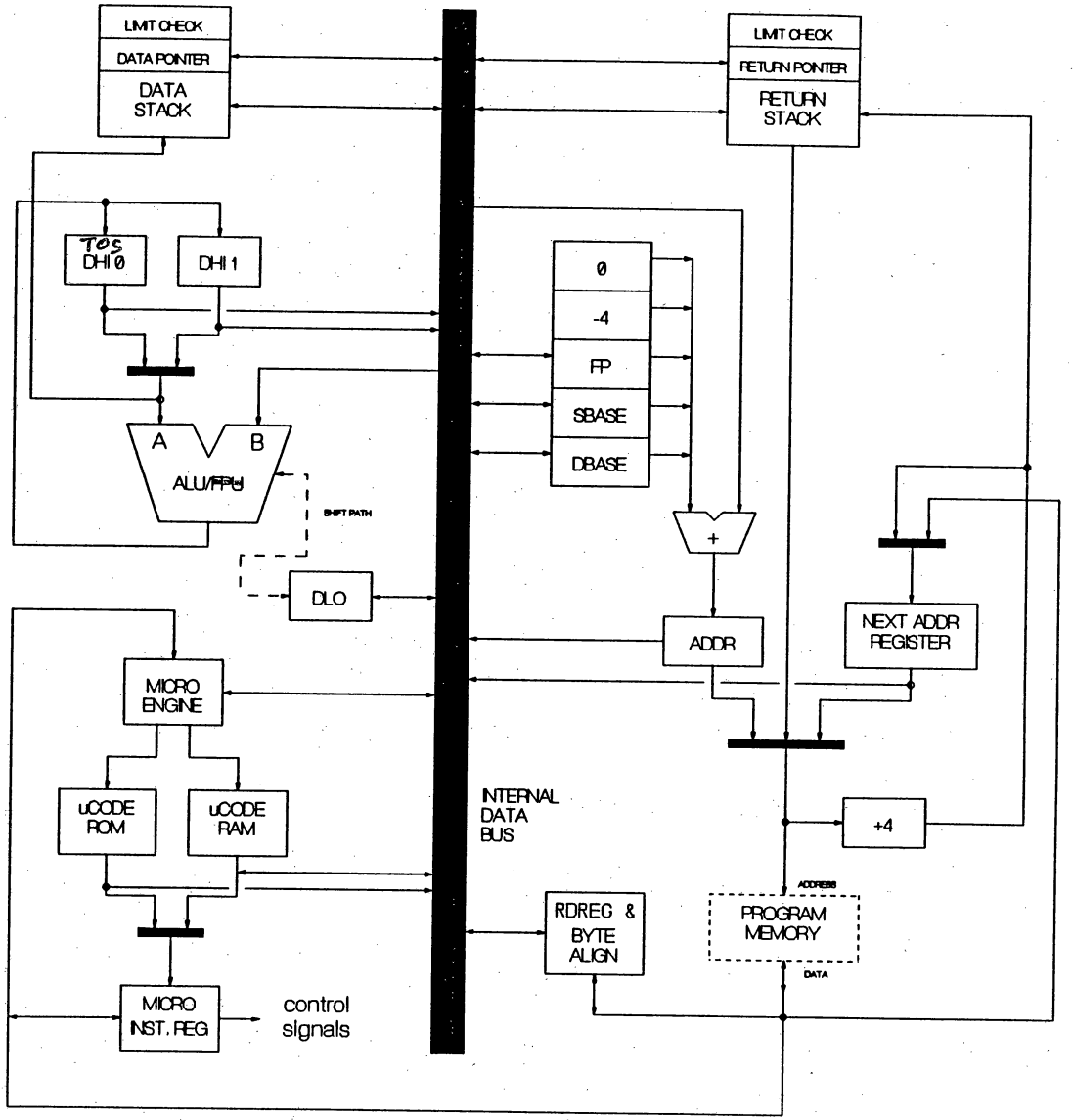


Figure 3-1. Simplified Block Diagram.

MICRO-SEQUENCER

The MICRO-SEQUENCER block includes microcode memory, the micro-engine, and the Microinstruction Register (MIR). The approach is that of a standard single-stage pipelined micro-controller. Conditional branching is accomplished by selecting the lowest bit of the next microaddress from a Condition Code Multiplexer. The MIR produces control signals for the rest of the system.

RETURN STACK (RS)

The RETURN STACK (RS) block is similar in function to the Data Stack (DS) block. It also provides direct connection with the Memory Addressing block to provide subroutine return addresses without tying up the System Bus.

MEMORY ADDRESSING

The MEMORY ADDRESSING block generates addresses for instruction fetches and data reads and writes. It has three sources for generating addresses: the Next Address Register, which is used for sequential program execution and subroutine calls, the top element of the Return Stack, which is used for subroutine returns, and the ADDR register, which is used for RAM data accesses and supports base pointer-plus-offset addressing. The FP, SBASE, and DBASE registers are three general-purpose base addressing registers for use with ADDR.

PROGRAM MEMORY

PROGRAM MEMORY consists of off-chip ROM and SRAM. Byte-alignment logic is employed to permit access to full-words, half-words, and bytes for both reads and writes. All quantities are forced to be aligned on natural boundaries (i.e. the bottom bit of half-word addresses is ignored, forcing alignment on even-byte boundaries).

3.2 THE FORTH MICROCODE SIMULATOR ENVIRONMENT

The BINAR Forth Microcode Simulator may be run on an IBM PC or PC-AT using the TIL Forth system. In order to preserve portability of examples and test code between 16- and 32-bit Forth systems, "W" words are used for all 32-bit quantities. Thus, W+ is used instead of D+ or + , and W. is used instead of D. or . .

Each microinstruction is formed by the Forth compiler as a series of bit patterns OR'ed into a 32-bit field. A new

microinstruction is begun with the word "<<" and concludes with the word ">>" . All micro-operations specified between << and >> are included in that single microinstruction. The order of micro-operations within the microinstruction specification is of no concern to the programmer.

For example,

```
<< >>
```

is a no-op microinstruction. The word >> sets default values for those fields which have not been explicitly specified. These defaults are shown in Figure 2-3.

The word >> also leaves the 32-bit microinstruction value on the Forth stack so that it may be readily accessed for subsequent operations.

The Microcode Simulator is designed to provide an environment equivalent to that which would be realized from a BINAR Core Processor installed in a plug-in board for a host PC. Thus, there is the functional equivalent of a host processor (which is the Forth system operating on the PC and the BINAR chip), and the target processor (which is the BINAR processor, itself).

To load data values into the BINAR system, an X! operation is performed. The value to be driven is taken from the top of the Forth data stack. For example, to load the DLO register, enter:

```
<< SOURCE=HOST DEST=DLO >> $12345678 X!
```

The X! actually transfers the 32-bit microinstruction to the BINAR chip, then executes this microinstruction while asserting the 32-bit value 12345678 Hex from the top of the Forth data stack onto the internal Data bus of the BINAR chip. Because a Bus destination of DLO has been specified, the value on the Bus is written into DLO.

The complementary function of X! is X@. X@ reads a 32-bit value from the Data Bus and returns the value on the top of the Forth data stack. To read the value that was just written into DLO in the previous example, type:

```
<< SOURCE=DLO DEST=HOST >> X@ W.
                          ( Result is 12345678 Hex)
```

The X@ transfers the bus value to the top of the Forth stack, and W. prints the top Forth 32-bit stack element.

Another important control function when interactively executing microcode is the word XGO. This word simply executes a microinstruction without either reading from or writing to the Data Bus. Because the BINAR chip employs a 2x clock, XGO actually cycles the OSC pin twice, effectively simulating a clock cycle.

Additional Microcode Simulator Environment Forth words are:

.MIR: Disassembles the contents of the most recently assembled microinstruction.

MIR@: Places the 32-bit value in the Microinstruction Register on top of the Forth stack.

RESET-SIM: Simulates the effects of asserting the RESET pin on the BINAR chip.

The Forth Microcode Simulator assumes that all program memory references are to two-cycle static RAM (SRAM), unless specified otherwise.

3.3 MICROCODING EXAMPLES

In discussions which follow, each of the BINAR functional blocks is examined in turn, and single-stepped microcode examples are considered for each. These discussions are based on the assumption that the reader has access to the BINAR Evaluation Board or the Forth Microcode Simulator program.

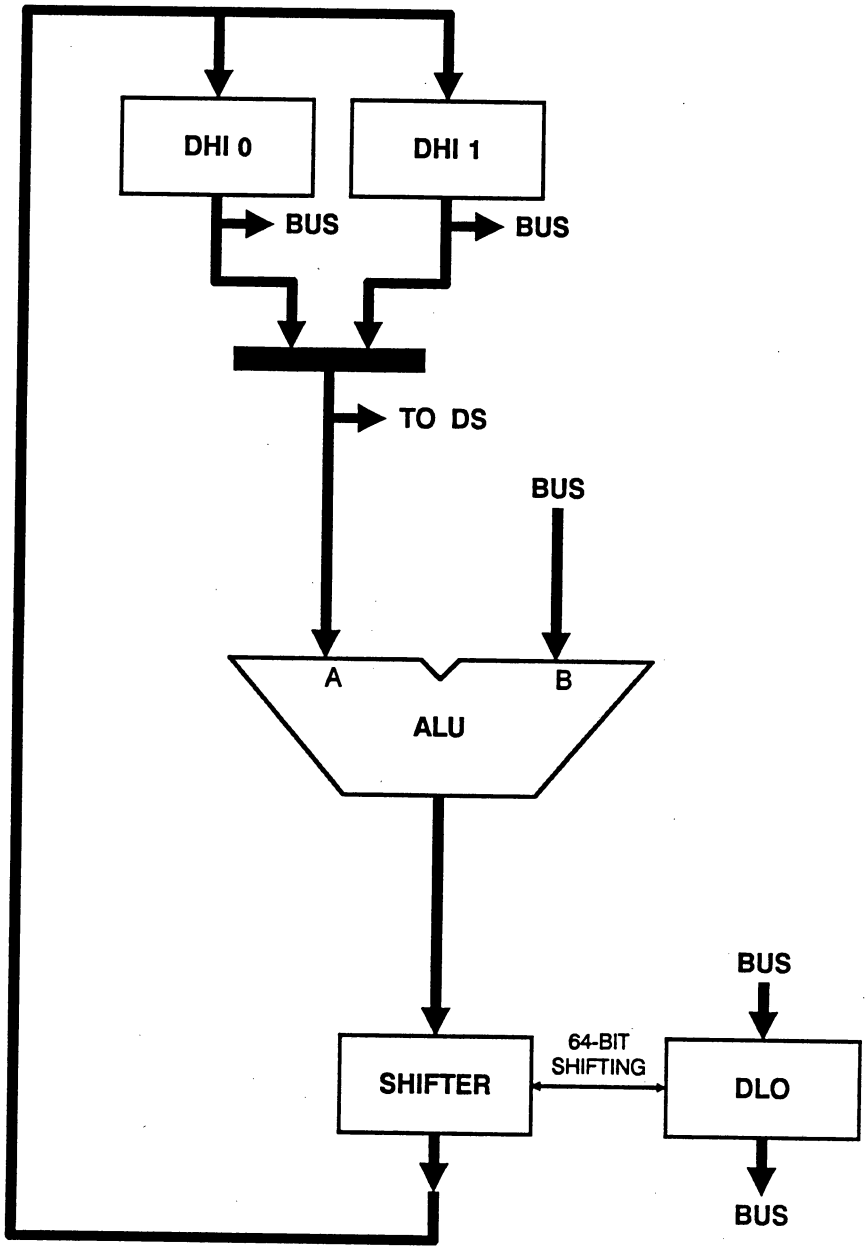
Because the Microassembler is the primary means for controlling hardware resources, all control signals are described in terms of their Microassembler mnemonics.

The examples which follow have been selected for their value in serving to convey understanding of BINAR machine operation. Occasionally, when clarity requires, discussion may gloss over the detailed "how's" and "why's" of machine operation. Further information concerning the intricacies of instruction decoding, execution, and other key issues and functions may be found in Section 4.0, Hardware Theory of Operation.

3.3.1 ALU/REGISTER FUNCTIONAL BLOCK

Figure 3-2 shows the ALU/Register functional block. This block includes a 32-bit integer ALU, a 32-bit multiplexer

Figure 3-2. ALU/Register Functional Block.



for shifting & selecting ALU outputs, two DHI registers, the DLO register, and Zero-Detect logic.

3.3.1.1 INTEGER ALU

The Integer ALU is based upon an optimized Harris ALU design, similar in capability to the 74181 TTL ALU with carry lookahead, but with function encoding compressed so as to reduce the number of control signals required by one bit.

The B side of the ALU is connected to the System Bus. The A side is connected to the DHI registers. The following discussion addresses the DHI[0] register. Consideration of DHI[1] is deferred.

Bits 16-19 of each microinstruction control ALU function. Conventional logic functions are supported except for XNOR. Addition and subtraction of the form A - B are also supported. The 74181 does not support B - A subtraction. Bit 20 specifies the carry-in bit for the ALU. The carry-in bit is fixed when microcode is written, instead of being supplied by a register bit as is sometimes seen on other designs. Conditional microcode branching is employed to perform add-with-carry and other conditional carry operations.

As a simple example of ALU operation, consider:

```
<< SOURCE=HOST ALU=B >> $1234 X!
```

In this example, the value 1234 Hex is placed on the Data Bus. The ALU is then instructed to pass the B side through to the output. This ALU output is then implicitly written into DHI[0]. ALU output is written to DHI on every clock cycle. Because the default ALU micro-operation is ALU = A, if no other operations are being performed, the data in DHI is, in effect, simply recirculated.

To read the value just placed in DHI[0], enter:

```
<< SOURCE=DHI DEST=HOST >> X@ W.
                          ( Result is 1234 Hex)
```

There is a certain laxity in terminology operating here in not making a distinction between DHI and DHI[0]. This is admissible because DHI[0] is the default DHI selection. Consequently, and until otherwise specified, DHI and DHI[0] will be treated as equivalent terms.

UNARY OPERATIONS

Either the A side or the B side of the ALU can be passed through unchanged:

```
<< SOURCE=HOST ALU=B >> $55 X!
<< SOURCE=DHI ALU=A DEST=HOST >> X@ W.
                              ( Result is 55 Hex)
<< SOURCE=DHI DEST=HOST >> X@ W.
                              ( Result is 55 Hex)
```

Observe that the second microinstruction simultaneously reads the DHI value and performs an ALU operation. In this instance, the ALU operation is the default ALU = A, which reloads DHI with its current value. This constitutes, in effect, an ALU no-op. We will see more examples of ALU operations performed in parallel with accesses to DHI in subsequent descriptions.

The ALU can also generate the constant values 0 and two's complement -1:

```
<< ALU=0 >> XGO
<< SOURCE=DHI ALU=-1 DEST=HOST >> X@ W.
                              ( Result is 0 Hex)
<< SOURCE=DHI DEST=HOST >> X@ W.
                              ( Result is FFFFFFFF Hex,
                                 Two's complement -1)
```

In this example, the second microinstruction reads the value of DHI set by the first microinstruction while the ALU is computing the next result. DHI is not updated with the new value of -1 until the microcycle has concluded.

One's complements of values (bit-wise logical complement) can be generated for either the A side or the B side:

```
<< SOURCE=HOST ALU=notB >> 5 X!
<< SOURCE=DHI ALU=notA DEST=HOST >> X@ W.
                              ( Result is FFFFFFFA Hex)
<< SOURCE=DHI ALU=notA DEST=HOST >> X@ W.
                              ( Result is 5 Hex)
<< SOURCE=DHI DEST=HOST >> X@ W.
                              ( Result is FFFFFFFA Hex)
```

The final unary operations to be considered are those for incrementing A, decrementing A, and doubling A:

```
<< SOURCE=HOST ALU=B >> 3 X!
```

```
<< ALU=A+0 >> XGO ( Clears passes A, forcing
                                    carry-out bit to 0)
<< SOURCE=DHI ALU=A+1 DEST=HOST >> X@ W.
                          ( Result is 3 Hex)
<< SOURCE=DHI ALU=A-1 DEST=HOST >> X@ W.
                          ( Result is 4 Hex)
<< SOURCE=DHI ALU=A+A DEST=HOST >> X@ W.
                          ( Result is 3 Hex)
<< SOURCE=DHI ALU=A+A+1 DEST=HOST >> X@ W.
                          ( Result is 6 Hex)
<< SOURCE=DHI DEST=HOST >> X@ W.
                          ( Result is D Hex)
```

TWO-OPERAND OPERATIONS

The remaining ALU operations require two operands. One of these operands originates from the Data Bus on the ALU B side, and the other from one of the DHI registers on the ALU A side. Note that one of the DHI registers in these instances will be overwritten with the result of the performed ALU operation:

```
<< SOURCE=HOST ALU=B >> 3 X!
<< SOURCE=DHI DEST=HOST >> X@ W.
                          ( Result is 3 Hex)
<< SOURCE=HOST ALU=A+B >> 4 X!
<< SOURCE=DHI DEST=HOST >> X@ W.
                          ( Result is 7 Hex)
<< SOURCE=HOST ALU=A-B >> 2 X!
<< SOURCE=DHI DEST=HOST >> X@ W.
                          ( Result is 5 Hex)
<< SOURCE=HOST ALU=AorB >> 3 X!
<< SOURCE=DHI DEST=HOST >> X@ W.
                          ( Result is 7 Hex)
```

The additional ALU operations: ALU=A+B+1 , ALU=A-B-1, ALU=AandB, ALU=AxorB, ALU=AnandB, and ALU=AnorB are employed in a similar manner. As can be seen from Figure 2-3, the carry-in bit can be used to select between slightly differing ALU functions. Those functions for which the CIN microcode bit is significant are termed "arithmetic" functions. The Microassembler forces the CIN field to the correct value for all arithmetic ALU functions. These functions also involve generation of an appropriate carry-out bit for the Condition Code Register. Logical operations (those not involving a plus or minus) do not depend upon the setting of the CIN bit. The CIN bit may be set to 0 or 1, as desired, for logical ALU operations.

To recap, functions supported by the ALU include:

ALU=A	Pass A side through unchanged
ALU=B	Pass B side through unchanged
ALU=notA	One's complement of A side
ALU=notB	One's complement of B side
ALU=AorB	A logical "or" B
ALU=AandB	A logical "and" B
ALU=AxorB	A logical "xor" B
ALU=AnorB	A logical "nor" B
ALU=AnandB	A logical "nand" B
ALU=0	Force ALU outputs to all 0
ALU=-1	Force ALU outputs to all 1
ALU=A+0	Pass A side through unchanged
ALU=A+1	Add 1 to A side
ALU=A-1	Subtract 1 from A side
ALU=A+A	Multiply A side by 2
ALU=A+A+1	Multiply A by 2 and add 1
ALU=A+B	Add A side to B side
ALU=A+B+1	Add A side to B side plus 1
ALU=A-B	Subtract B side from A side
ALU=A-B-1	Subtract B side from A side, then subtract 1

3.3.1.2 ALU MUX

The ALU Mux is a four-to-one multiplexer that selects one from the following: Pass ALU output through unchanged (PASS[ALU]); Shift ALU output 1 bit left (SL[ALU]), Shift ALU output 1 bit right (SR[ALU]), and Undefined (reserved for use with a floating point unit). In addition to their utility for logical shifting operations, these functions are also employed in microcoded multiplication and division.

The default function is PASS[ALU], which simply passes the output of the ALU through unchanged.

The SL[ALU] micro-operation shifts the ALU output one bit left, discarding the highest bit, and inserts the highest bit from the DLO register into the lowest bit position of the multiplexer output.

The SR[ALU] micro-operation shifts the ALU output right one bit, discarding the lowest bit (or shifts it into the DLO register if SR[DLO] is used), and shifts the CIN bit value into the highest bit position.

```
<< SOURCE=HOST DEST=DLO >> 0 X! ( Zero highest bit)
<< SOURCE=HOST ALU=B >> $400 X!
```

```
<< ALU=A SL[ALU] >> XGO
<< SOURCE=DHI DEST=HOST >> X@ W.
                           ( Result is 800 Hex)
<< ALU=A CIN=0 SR[ALU] >> XGO
<< ALU=A CIN=0 SR[ALU] >> XGO
<< SOURCE=DHI DEST=HOST >> X@ W.
                           ( Result is 200 Hex)
<< SOURCE=HOST DEST=DLO >> $80000000 X!
                           ( Set highest bit)
<< ALU=A  SL[ALU] >> XGO
<< SOURCE=DHI DEST=HOST >> X@ W.
                           ( Result is 401 Hex)
```

3.3.1.3 DHI REGISTERS

As has become apparent from earlier discussion, the DHI Registers are actually a pair of registers: DHI[0] and DHI[1]. Together, they form a holding area to provide second operands for the ALU (the first having been provided from the Data Bus). Most often, a single DHI register is all that is required. This accounts for the default situation being that all references to DHI employ DHI[0], and DHI[1] is ignored.

```
<< SOURCE=HOST ALU=B >> $9876 X!
<< SOURCE=DHI DEST=HOST >> X@ W.
                           ( Result is 9876 Hex)
<< SOURCE=DHI[0] DEST=HOST >> X@ W.
                           ( Result same as above)
```

When running high-level language programs, DHI[0] is employed to hold the top stack element. This permits convenient access to the top two ALU stack elements at all times; the next-to-top stack element perceived by the programmer is stored at the top of the hardware Data Stack.

Occasionally, one may wish to perform an ALU operation without having first to move the DHI[0] value out of the way and then restore it. Accordingly, a second register, DHI[1], has been provided for storing intermediate values. Either DHI[0] or DHI[1] may be selected as both the source and destination for ALU operations through use of the DHI[0] and DHI[1] micro-operations. These micro-operations control which of the two registers is clocked with the output of the ALU multiplexer, and which of the two is selected by the DHI Mux to be fed into the A side of the ALU. The same register is used for both input to the ALU and for depositing the result of the ALU. In all cases, DHI[0] is the default micro-operation. Note that these uses of DHI[0] and DHI[1]

are completely independent of SOURCE=DHI[0] and SOURCE=DHI[1] micro-operations.

```
<< SOURCE=HOST ALU=B DHI[0] >> $111111 X!
<< SOURCE=HOST ALU=B DHI[1] >> $222222 X!
<< SOURCE=DHI[0] DEST=HOST >> X@ W.
                              ( Result is 111111 Hex)
<< SOURCE=DHI[1] DEST=HOST >> X@ W.
                              ( Result is 222222 Hex)
<< SOURCE=HOST ALU=A+B DHI[0] >> $55 X!
<< SOURCE=HOST ALU=A+B DHI[1] >> $77 X!
<< SOURCE=DHI[0] DEST=HOST >> X@ W.
                              ( Result is 111166 Hex)
<< SOURCE=DHI[1] DEST=HOST >> X@ W.
                              ( Result is 222299 Hex)
<< SOURCE=DHI[0] ALU=A-1 DHI[1] DEST=HOST >> X@ W.
                              ( Result is 111166 Hex)
<< SOURCE=DHI[1] DEST=HOST >> X@ W.
                              ( Result is 222298 Hex)
```

Occasionally, it is desired to transfer a value between the DHI[0] and DHI[1] registers. To permit this, DEST=DHI[0] and DEST=DHI[1] have been provided. Unlike other bus destinations operators, however, they do not load the registers from the bus, but instead control which register is clocked with the output of the ALU. These operations override the DHI[0] and DHI[1] micro-operations for clocking results into the registers:

```
<< SOURCE=HOST ALU=B DHI[1] >> $888 X!
<< SOURCE=HOST ALU=A+B DHI[1] DEST=DHI[0] >> $222 X!
<< SOURCE=DHI[0] DEST=HOST >> X@ W.
                              ( Result is AAA Hex)
<< SOURCE=DHI[1] DEST=HOST >> X@ W.
                              ( Result is 888 Hex)
```

In this example, DHI[1] has been loaded with the value 888 Hex. DHI[1] is then read by the third microinstruction and added to the number 222 Hex. The DEST=DHI[0] micro-operation overrides the normal action of merely depositing the result in DHI[1] and, instead, deposits the result in DHI[0]. A non-destructive read of the value in DHI[1] is then performed for use in the addition process.

3.3.1.4 DLO Register

The DLO register is a 32-bit shift register that may be alternatively employed as either a temporary holding register or as the lower half of a 64-bit shifting

mechanism. In its role as holding register, SOURCE=DLO and DEST=DLO provide for storing a 32-bit value and reading it back:

```
<< SOURCE=HOST DEST=DLO >> $123456 X!
<< SOURCE=DLO DEST=HOST >> X@ W.
                           ( Result is 123456 Hex)
```

When employed as a shift register, DLO may be shifted independently using SL[DLO] or SR[DLO]. Note that DLO may not be both set and shifted within the same microcycle, but may be read and shifted within the same microcycle.

When shifting left, the lowest bit of DLO is set by the CIN bit. When shifting right, the highest bit of DLO is set by the lowest bit of the ALU output:

```
<< SOURCE=HOST DEST=DLO >> 4 X!
<< CIN=1 SL[DLO] >> XGO
<< SOURCE=DLO DEST=HOST >> X@ W.
                           ( Result is 9 Hex)
<< SOURCE=HOST DEST=DLO >> $FFFF0000 X!
<< ALU=-1 SR[DLO] >> XGO ( Shift-in bit is 1)
<< ALU=0 SR[DLO] SOURCE=DLO DEST=HOST >> X@ WU.
                           ( Result is FFFF8000 Hex)
<< SOURCE=DLO DEST=HOST >> X@ W.
                           ( Result is 7FFFC000 Hex)
```

DLO shifting operations may also be combined with the ALU multiplexer to form a 64-bit shift register in the following manner:

```
<< SOURCE=HOST DEST=DLO >> $44444444 X!
<< SOURCE=HOST ALU=B >> $11111111 X!
<< CIN=0 SR[ALU] SR[DLO] >> XGO
<< SOURCE=DLO  DEST=HOST >> X@ W.
                           ( Result is A2222222 Hex)
<< SOURCE=DHI  DEST=HOST >> X@ W.
                           ( Result is 8888888 Hex)
```

3.3.1.5 ZERO-DETECT AND CONDITION CODES

With almost all programs, ALU results must be evaluated/tested for purposes of conditional branching. While the BINAR processor does not provide Condition Code flags that are visible to the high level language programmer, Condition Codes are employed at the microcode level. Most of these Condition Codes have their source in the ALU:

The Z Condition Code bit, reflecting zero-detect logic status for the ALU, is set true at the end of the clock cycle if the ALU multiplexer output value is zero. In the case where the ALU multiplexer output has been written to DHI[0], this is equivalent to a flag that indicates whether the current top-of-stack element is zero. This particular capability is especially useful for implementing the Forth 0BRANCH instruction.

The S Condition Code bit, signifying Sign bit status, is set to a value corresponding to bit 31 of the ALU Mux output. The S Condition Code bit is 1 if bit 31 of the ALU Mux output is set (which signifies that the sign bit is set for negative two's complement results).

The C Condition Code bit corresponds to the condition of the carry-out bit of the ALU. The C Condition Code bit is true if there was a carry-out from the prior instruction. The Carry-out bit is valid only after an arithmetic ALU operation has been performed. In contrast to many other machines, the C bit is only valid during the clock cycle after the arithmetic ALU operation has been performed; it must therefore be tested using a conditional microbranch based on the C bit in the microinstruction immediately after the ALU operation.

The V Condition Code bit is set whenever an arithmetic ALU operation has produced an overflow. The value of the V Condition Code bit is implemented by the processor's "examining" the A and B inputs and determining if the ALU output has a valid sign. Note that overflow is checked at the output of the ALU, not at the output of the ALU Mux. Similar to the C bit, the oVerflow bit is valid only during the clock cycle after an arithmetic ALU operation has been performed.

The P Condition Code bit, or Interrupt Pending bit, signifies that an interrupt condition is waiting for service, and is useful with restartable instructions for relinquishing control to an interrupt. For example, a restartable block memory move can test the P bit after moving each memory element, then exit if an interrupt is pending, saving state for a later instruction restart. The P bit is set whenever a non-masked interrupt is pending (i.e. whenever the next instruction to be executed will be preempted by an interrupt).

Note that each of these Condition Codes is valid for only one clock cycle, and all are recomputed and loaded on each

successive clock cycle. This is especially important to remember for proper interpretation of the C and V Condition Code bits.

3.3.1.6 MULTIPLICATION AND DIVISION

Two very special modes of ALU operation have been provided for multiplication and division, respectively. Shift-and-conditional-add multiplication and non-restoring division (for unsigned numbers) may be performed conveniently in microcode.

MULTIPLICATION

When the bus source designation is specified as MULTIPLY-STEP, the ALU Multiplexer performs an arithmetic shift-right operation on the partial sum (which works by feeding the arithmetic carry-out bit into the multiplexer). At the same time, an implicit SOURCE=DS is performed for use with a possible ALU=A+B operation. This achieves 32-bit by 32-bit multiplication, yielding a 64-bit result, and is accomplished in 32 microcycles with only a few additional cycles of overhead required for setup. There is no special hardware for controlling whether or not the multiplier is added on each cycle, so a microcode branch must be performed on bit 0 of DLO for each clock cycle. Since the carry-out bit is not defined for logical ALU operations, the micro-operation ALU=A+0 must be used when performing a shift-without-add step. See the microcode for the UM* instruction for an example.

DIVISION

When the bus source designation is specified DIVIDE-STEP, the ALU Multiplexer performs non-restoring unsigned division on a 64-bit dividend and 32-bit quotient. When DIVIDE is used, the logical complement of the ALU sign bit is shifted into DLO shift-left input. Also, the ALU function is manipulated to provide either ALU=A+B or ALU=A-B operation, as required. In order for the hardware that manipulates the ALU function to work properly, the ALU function must be set to 14, and the carry-in bit set to 1 in the microinstruction. The DIVIDE micro-operation in the microassembler automatically sets the ALU control bits and carry-in field to the appropriate values. Division requires 32 clock cycles plus a small amount of overhead. See the microcode for the UM/MOD instruction for an example.

3.3.2 DATA STACK FUNCTIONAL BLOCK

Figure 3-3 shows the Data Stack functional block. This block includes a 16-bit stack pointer, a 64-element by 32-bit stack RAM, a stack buffer register, and stack pointer limit checking logic. When running programs, the data stack contains the machine's evaluation and parameter passing stack elements, except for the top-most stack element, which is contained in the DHI[0] register as described previously.

3.3.2.1 DATA STACK POINTER

The Data Stack Pointer (DP) register is a 6-bit up/down counter that may be incremented, decremented, or loaded on each microinstruction.

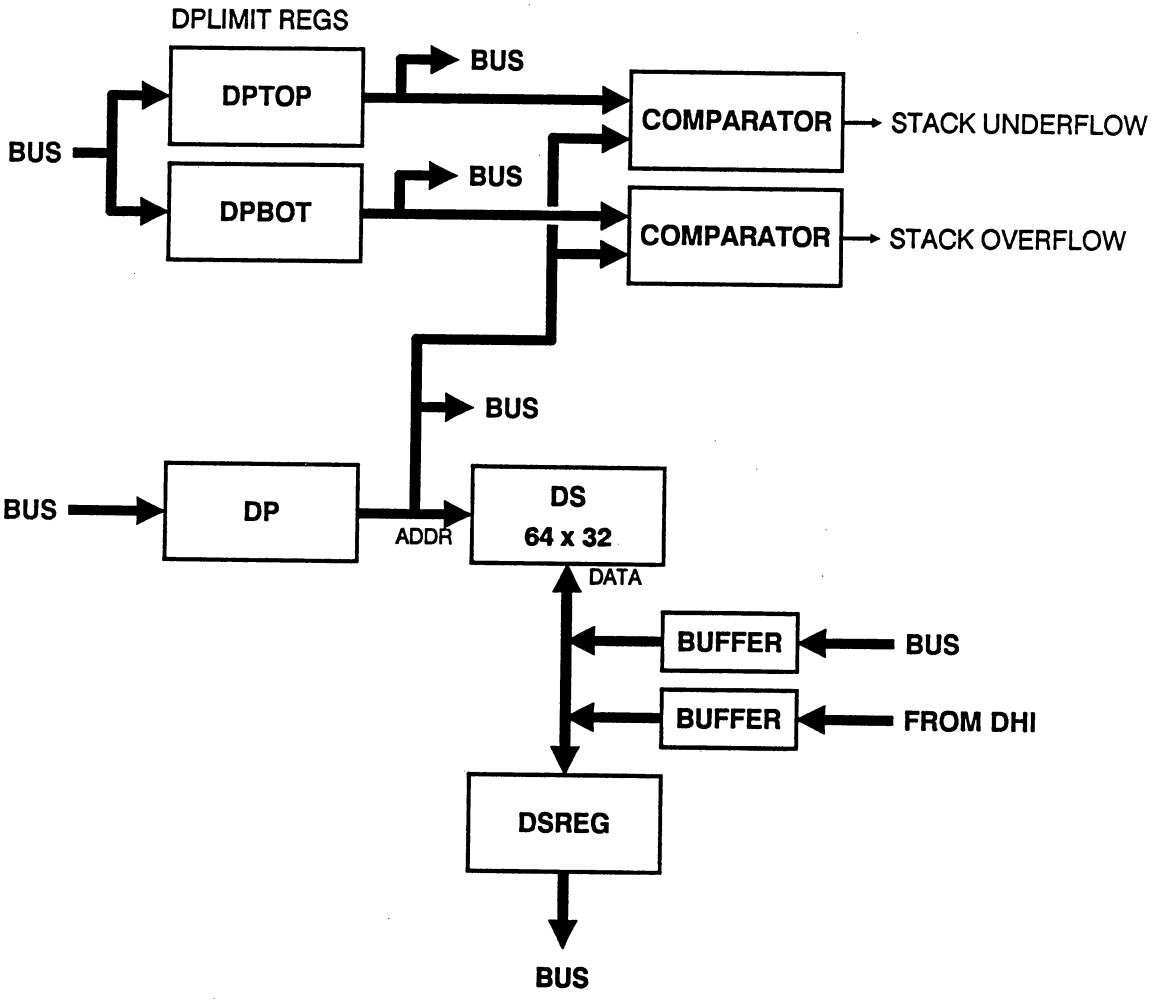
```
<< SOURCE=HOST DEST=DP >> $10 X!
<< SOURCE=DP DEST=HOST >> X@ W.
                          ( Result is 10 Hex)
<< INC[DP] >> XGO
<< SOURCE=DP DEST=HOST >> X@ W.
                          ( Result is 11 Hex)
<< SOURCE=DP DEC[DP] DEST=HOST >> X@ W.
                          ( Result is 10 Hex)
<< SOURCE=DP DEC[DP] DEST=HOST >> X@ W.
                          ( Result is 9 Hex)
<< SOURCE=DP DEC[DP] DEST=HOST >> X@ W.
                          ( Result is 8 Hex)
```

The DP register may be sourced in parallel with an increment or decrement. To support efficient access to the stack RAM, incrementing or decrementing of the DP takes place at the beginning of the microcycle. This constitutes an exception to the rule that most registers are updated at the end of the microcycle, but is in general transparent to the programmer. Note, however, that DEST=DP is executed at the end of the clock cycle, and works in a manner similar to that for other bus destinations.

Due to details of implementation, INC[DP] and DEC[DP] may not be executed on the microcycle immediately following execution of a DEST=DP operation. Failure to observe this precaution will result in the increment/decrement micro-operation being ignored. Note, however, that the DP register will be loaded with the correct value.

The following rules summarize the use of DP and DS. When writing to DS, INC[DP] acts as if DP is incremented before the write, and DEC[DP] acts as if DP is decremented before

Figure 3-3. Data Stack Functional Block.



the write. When reading from DS, INC[DP] acts as if DP is incremented after the read, and DEC[DP] acts as if DP is decremented after the read. When using DEST=DP, a single clock cycle pause should be allowed to allow the data stack to catch up to the new DP value.

3.3.2.2 DATA STACK RAM

The Data Stack (DS) is a 64-element by 32-bit synchronous RAM array. To enhance system performance, data stack elements are actually read during the cycle before they are needed (Note that this correlates with the DP register being incremented/decremented at the beginning of the clock cycle.) The stack elements are read into the DS Register described below.

By convention, the Data Stack grows from high addresses to low, with an INC[DP] acting as a "pop" operation, and a DEC[DP] acting as a "push" operation.

To the user, the stack appears to perform a pre-increment or pre-decrement when performing a DEST=DS, and a post-increment or post-decrement when performing a SOURCE=DS. This is, in fact, exactly the behavior that is desired when using a stack, since an element can be allocated on the stack while it is being written, as in:

```
... DEC[DP]  DEST=DS  ...
```

which writes the result into the DS register at the address of the decremented DP. Also, an element can be popped from the stack after it is read:

```
... INC[DP]  SOURCE=DS ...
```

which reads the result from DS before the DP is incremented.

A microcoding consideration imposes one microcycle of latency between the time that the DP is changed using a DEST=DP and the instant that the new DP value takes effect. Within the microcycle immediately following invocation of DEST=DP, a SOURCE=DS would produce a DS value corresponding to the old DP value. Note, however, that DEST=DS will work within the cycle immediately following DEST=DP. The latency experienced with DEST=DP is unavoidable and is an artifact of the embedded micro-engine and access method used on the stack. No latency is present when using INC[DP] and DEC[DP].

Detailed examples appear in the next section.

3.3.2.3 DATA STACK BUFFER REGISTERS

Since the Data Stack is read on the cycle before the value is needed, a buffer register is required to hold the result for use by the machine. This buffer register is the Data Stack Register (DSREG). Each time the DP is incremented, decremented, or loaded, DS RAM is read into the DSREG register. When the micro-operation SOURCE=DS is used, it is actually the DSREG that is being sourced onto the bus, thus accounting for the latency described above. When a DEST=DS micro-operation is used, however, Data Stack RAM and the DSREG register are both updated with the new value. To the user it appears that the DSREG register provides transparent access to the DS , taking into account the fact that the DP register is pre-incremented/decremented when writing to the DS and post-incremented/decremented when reading from the DS:

```
<< SOURCE=HOST DEST=DP >> $15 X!
<< SOURCE=HOST DEST=DS >> $111 X! ( 111H at addr 15H)
<< SOURCE=HOST DEC[DP] DEST=DS >> $222 X!
                          ( 222 Hex at addr 14 Hex )
<< SOURCE=HOST DEC[DP] DEST=DS >> $333 X!
                          ( 333 Hex at addr 13 Hex )
<< SOURCE=DP DEST=HOST >> X@ W.
                          ( Result is 13 Hex)
<< SOURCE=DS INC[DP] DEST=HOST >> X@ W.
                          ( Result is 333 Hex)
<< SOURCE=DS INC[DP] DEST=HOST >> X@ W.
                          ( Result is 222 Hex)
<< SOURCE=DS DEST=HOST >> X@ W.
                          ( Result is 111 Hex)
<< SOURCE=DP DEST=HOST >> X@ W.
                          ( Result is 15 Hex)
```

An additional dedicated bus path for writing the Data Stack is provided to permit single-cycle Forth SWAP and OVER operations. This path allows reading the DSREG register value over the normal Data Bus while simultaneously writing the selected DHI register directly to the Data Stack over the dedicated bus. For example, a Forth SWAP operation which exchanges the DSREG register value with the DHI register value could be written:

```
<< SOURCE=HOST ALU=B >> $88888 X!
<< SOURCE=HOST DEST=DS >> $99999 X!
<< SOURCE=DS ALU=B DS-FROM-DHI >> XGO
```

```
<< SOURCE=DS DEST=HOST >> X@ W.
                          ( Result is 88888 Hex)
<< SOURCE=DHI DEST=HOST >> X@ W.
                          ( Result is 99999 Hex)
```

3.3.2.4 DATA STACK POINTER LIMIT CHECKING

A pair of limit registers is provided to form upper and lower bounds for DP values. The DPTOP and DPBOT registers are each 6 bits, and are compared to the value in DP at the end of each microcycle. If the value of DP is greater than or equal to the DPTOP value, or less than or equal to DPBOT value, an interrupt is generated. DPTOP and DPBOT are together read and written as a 32-bit value pair. The following example shows the DPTOP register being set to 003F Hex and the DPBOT register to 0010 Hex:

```
<< SOURCE=HOST DEST=DP-LIMIT >> $003F0010 X!
<< SOURCE=DP-LIMIT DEST=HOST >> X@ W.
                          ( Result is 3F0010 Hex)
```

The DPTOP/DPBOT registers may be re-loaded with new values at run time to facilitate sharing the Data Stack between multiple processes. In other words, the user can change the limit register value at run time in order to partition the available stack memory into multiple non-overlapped smaller stack memories, e.g. for different tasks.

The interrupt generated by a stack overflow or underflow may be used to page the stack in and out of program memory to permit creating a virtual stack of essentially unlimited size. A word of caution is in order for this use however. Since interrupts are synchronized to instruction boundaries, and since an interrupt will not be recognized if it is generated on the last microcycle of an instruction, extra room must be left above and below the DPTOP/DPBOT values to accommodate over-run. The amount of room that must be left is equal to the maximum number of stack elements pushed/popped by any instruction plus one. Also, for virtual stack paging to work properly, these extra elements must contain valid data items to cover the case when stack elements beyond the boundary register are popped and then pushed before the interrupt can take effect. To imagine an example of this, consider what would occur if a ROT instruction were executed when the DP value is right at the DPTOP value while using a virtual stack with several hundred elements pushed to program memory.

3.3.3 RETURN STACK FUNCTIONAL BLOCK

Figure 3-4 shows the Return Stack functional block. This block includes a 16-bit stack pointer, a 64-element by 32-bit stack RAM array, two stack buffer registers, and stack pointer limit checking logic.

When running programs, the Return stack contains the machine's subroutine return address elements and loop variable information for Forth programs. Return Stack operation is identical in most respects to Data Stack operation, except that the alternative direct path for writing to the Return Stack originates from the Return-Save register rather than from the DHI registers.

3.3.3.1 RETURN STACK POINTER

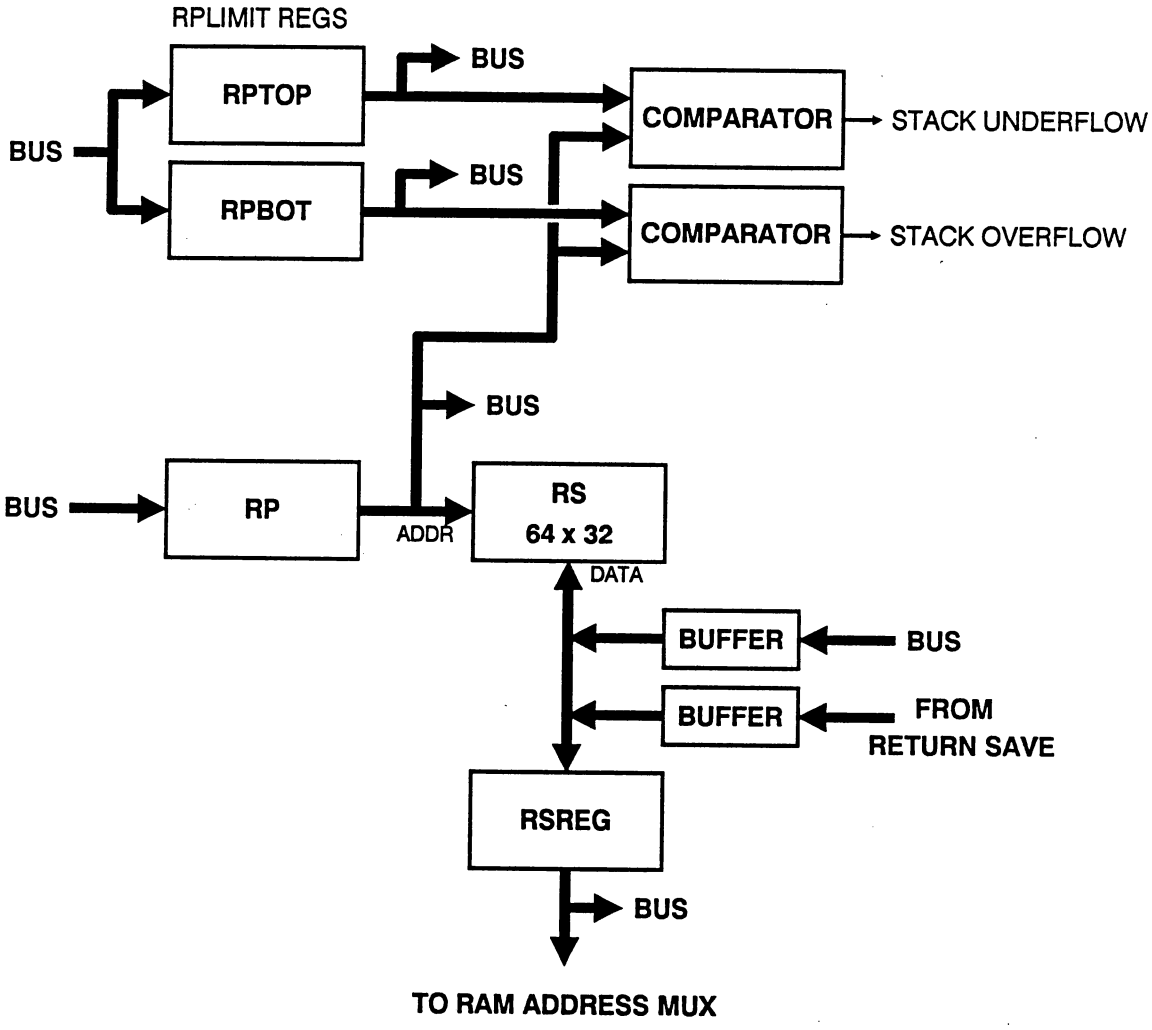
The Return Stack Pointer (RP) is a 6-bit up/down counter that may be incremented, decremented, or loaded on each microinstruction.

```
<< SOURCE=HOST DEST=RP >> $10 X!
<< SOURCE=RP DEST=HOST >> X@ W.
                          ( Result is 10 Hex)
<< INC[RP] >> XGO
<< SOURCE=RP DEST=HOST >> X@ W.
                          ( Result is 11 Hex)
<< SOURCE=RP DEC[RP] DEST=HOST >> X@ W.
                          ( Result is 10 Hex)
<< SOURCE=RP DEC[RP] DEST=HOST >> X@ W.
                          ( Result is 9 Hex)
<< SOURCE=RP DEC[RP] DEST=HOST >> X@ W.
                          ( Result is 8 Hex)
```

The RP counter register may be sourced in parallel with an increment or decrement. Incrementing or decrementing of the RP register takes place at the beginning of the microcycle. As in the DP register, this constitutes an exception to the rule that registers are updated at the end of the microcycle. Note, however, that DEST=RP is executed at the end of the clock cycle, and works in a manner similar to that for other bus destination.

INC[RP] and DEC[RP] may also not be executed on the microcycle immediately following execution of a DEST=RP operation. Failure to observe this precaution will result in the increment/decrement micro-operation being ignored. Note, however, that the RP register will be loaded with the correct value.

Figure 3-4. Return Stack Functional Block.



The following rules summarize the use of RP and RS. When writing to RS, INC[RP] acts as if RP is incremented before the write, and DEC[RP] acts as if RP is decremented before the write. When reading from RS, INC[RP] acts as if RP is incremented after the read, and DEC[RP] acts as if RP is decremented after the read. When using DEST=RP, a single clock cycle pause should be allowed to allow the return stack to catch up to the new RP value.

Additionally, in general the RP value may not be modified during the same microcycle as that in which the DECODE micro-operation occurs if there is any chance that the instruction being executed is of the CALL or EXIT type. If this precaution is violated, the value of the RP register will not bear a correct result, and the machine will crash. In summary, RP manipulation during the DECODE microcycle (i.e. last microcycle) of an opcode should be attempted only with extreme caution.

3.3.3.2 RETURN STACK RAM

The Return Stack (RS) is a 64 element by 32-bit synchronous RAM array. To enhance system performance, Return stack elements are actually read during the cycle before they are needed (hence the fact that RP is incremented/ decremented at the beginning of the clock cycle.) The stack elements are read into the RS Register described later.

By convention, the Return Stack grows from high addresses to low, with an INC[RP] acting as a "pop" operation, and a DEC[RP] acting as a "push" operation.

To the user, the stack appears to perform a pre-increment or pre-decrement when performing a DEST=RS, and a post-increment or post-decrement when performing a SOURCE=RS. This is in fact exactly the behavior that is desired when using a stack, since an element can be allocated on the stack while it is being written:

```
... DEC[RP]  DEST=RS  ...
```

which writes the result into the RS at the address of the decremented RP. Also, an element can be popped from the stack after it is read:

```
... INC[RP]  SOURCE=RS ...
```

which reads the result from RS before the RP is incremented.

A minor microcoding consideration is that there is one microcycle of latency between when the RP value is changed using a DEST=RP micro-operation and when the new RP value takes effect. The microcycle after the DEST=RP, a SOURCE=RS will produce the RS value at the old RP value. However, a DEST=RS will work properly on the cycle after a DEST=RP. This latency is a side-effect of the pipelined access to the stack. No latency is present when using INC[RP] and DEC[RP].

Detailed examples appear in the next section.

3.3.3.3 RETURN STACK BUFFER REGISTER

Since the Return Stack is read on the cycle before the value is needed, a buffer register is needed to hold the result for use by the rest of the machine. This register is the Return Stack Register (RSREG). Every time the RP is incremented, decremented, or loaded, the RS RAM is read into the RSREG. When the micro-operation SOURCE=RS is used, it is actually the RSREG that is being sourced onto the bus. When a DEST=RS micro-operation is used or an address is saved during a subroutine call, the RS RAM and the RSREG are both written with the new value. To the user it appears that the RSREG always contains the correct value taking into account the fact that the RP is pre-increment/ decrement when writing to the RS and post-increment/ decrement when reading from the RS.

```
     << SOURCE=HOST DEST=RP >> $15 X!
     << SOURCE=HOST DEST=RS >> $111 X! ( 111H at addr 15H)
     << SOURCE=HOST DEC[RP] DEST=RS >> $222 X! ( 222H at
addr 14H)
     << SOURCE=HOST DEC[RP] DEST=RS >> $333 X! ( 333H at
addr 13H)
     << SOURCE=RP DEST=HOST >> X@ W.
                               ( Result is 13 Hex)
     << SOURCE=RS INC[RP] DEST=HOST >> X@ W.
                               ( Result is 333 Hex)
     << SOURCE=RS INC[RP] DEST=HOST >> X@ W.
                               ( Result is 222 Hex)
     << SOURCE=RS DEST=HOST >> X@ W.
                               ( Result is 111 Hex)
     << SOURCE=RP DEST=HOST >> X@ W.
                               ( Result is 15 Hex)
```

A separate path between the RS and the Memory Address functional block is used to pass subroutine return addresses

back and forth without tying up the Data Bus. One path is from the RSREG to the Memory Address block RAM Address Multiplexer for fetching return addresses during subroutine returns. The other path is from the Return Save register in the Memory Address block to the RS for writing return addresses during subroutine calls.

3.3.3.4 RETURN STACK POINTER LIMIT CHECKING

A pair of limit registers is provided to form an upper and lower limit for the RP values. The RPTOP and RPBOT registers are each 6 bits, and are compared to the value in RP at the end of every microcycle. If the value of RP is greater than or equal to RPTOP, or less than or equal to RPBOT, an interrupt is generated. RPTOP and RPBOT are read and written as a 32-bit value pair.

```
<< SOURCE=HOST DEST=RP-LIMIT >> $003F0010 X!
<< SOURCE=RP-LIMIT DEST=HOST >> X@ W.
                              ( Result is 3F0010 Hex)
```

The above example sets the RPTOP to 003F Hex and RPBOT to 0010 Hex. The RPTOP/RPBOT registers may be re-loaded with new values at run time to share the Return Stack between multiple processes.

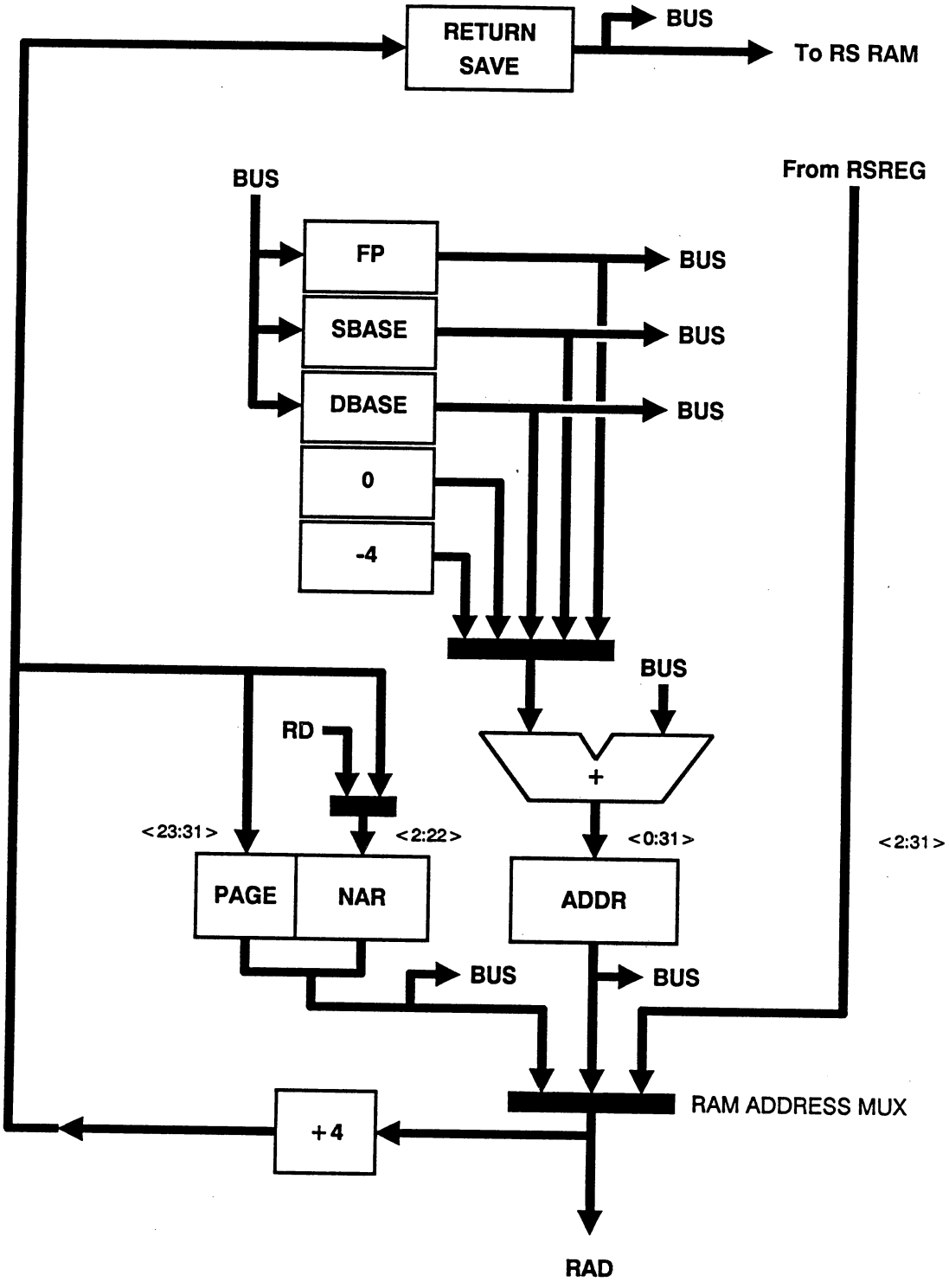
The interrupt generated by a stack overflow or underflow may be used to page the stack in and out of program memory to create a virtual stack of essentially unlimited size. As with the Data Stack, caution must be exercised to prevent problems due to stack overrun. In particular, don't forget about Forth instructions like DR> and J .

3.3.4 MEMORY ADDRESS FUNCTIONAL BLOCK

Figure 3-5 shows the Memory Address functional block. This block contains a 3-input multiplexer for selecting one of three sources for memory addresses. Two of the sources are for instruction fetching. The RSREG path is used when performing subroutine returns. The PAGE/NAR path is used for fetching sequential instructions as well as for Jumps and subroutine CALLs. The incrementer that feeds back into PAGE/NAR and the Return Save register is used to compute subroutine return addresses and addresses for sequential instruction fetches.

The ADDR register path is used for data accesses to memory. A hardware adder with three base registers and two constant inputs allows adding an offset to a base value for data

Figure 3-5. Memory Address Functional Block.



accesses. This provides high speed frame-pointer-plus-offset addressing for memory-resident stack frames, heaps, and other data structures.

3.3.4.1 ADDR REGISTER

The ADDR register holds the memory address used for all data fetches and stores. It is loaded using one of several ADDR=... micro-operations.

The simplest way to load the ADDR register is:

```
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $12345678 X!
<< >> XGO
<< >> XGO
<< SOURCE=ADDR DEST=HOST >> X@ W.
                          ( Result is 12345678 Hex)
```

In this example, the ADDR register is loaded with the ADDR=BUS+0(CYCLE) bus destination micro-operation, then read back. The mnemonic for the load operation indicates that ADDR is loaded with the value on the Data Bus and a memory cycle is initiated. For the time being we will ignore the consequences of the memory cycle initiation on this and other micro-operations, and just concentrate on the values being placed in ADDR. The two no-op microinstructions are executed to give the memory cycle initiated by the use of an ADDR=... micro-operation time to complete.

3.3.4.2 ADDR BASE REGISTERS

The path from the Data Bus to ADDR goes through an adder. In the case of the ADDR=BUS+0(CYCLE) micro-operation, the bus value was passed through unchanged (by adding 0). A more general case for loading ADDR is to add one of four values to the quantity on the Data Bus and load the result into ADDR. One of these quantities is the constant -4.

```
<< SOURCE=HOST ADDR=BUS-4(CYCLE) >> $16 X!
<< >> XGO
<< >> XGO
<< SOURCE=ADDR DEST=HOST >> X@ W.
                          ( Result is 12 Hex)
```

The use of ADDR=BUS-4(CYCLE) simply subtracts four from the value on the Data Bus before loading the result into ADDR. This capability is useful in performing return from interrupts, where the saved address is the desired restart address plus 4. It is also quite useful when executing tree

data structures, as it allows computing the address of the calling node from the saved subroutine return address (which is the calling node location plus 4).

The other three values that can be added to the Data Bus value when loading ADDR are the three base registers FP, SBASE, and DBASE. The three registers are identical, but by convention FP is used to hold the Frame Pointer value for C and other environments that use program memory stack frames. SBASE and DBASE may be used for anything, but are named for their obvious uses as a source and destination base pointer for data structure manipulations.

```
<< SOURCE=HOST DEST=FP >> $4567 X!
<< SOURCE=FP DEST=HOST >> X@ W.
                          ( Result is 4567 Hex)
<< SOURCE=HOST ADDR=BUS+FP(CYCLE) >> $1111 X!
<< SOURCE=ADDR DEST=HOST >> X@ W.
                          ( Result is 5678 Hex)
```

Examples for SBASE and DBASE may be obtained by substituting "SBASE" or "DBASE" for "FP" in the above example. SBASE, DBASE, and FP are not incrementable nor decrementable. This is because in the case of FP, the frame is changed relatively infrequently. Most accesses are made by offsets without changing the FP. In the case of SBASE and DBASE used for string processing, a changing index value that is incremented by the ALU can be used to provide efficient string handling.

An important capability is efficient Frame Pointer-plus-offset addressing. This can be obtained by compiling an instruction that holds the offset in the short literal field of a Literal or Subroutine Return instruction and executing an opcode that uses the micro-operations:

```
...  SOURCE=LITERAL  ADDR=BUS+FP(CYCLE) ...
```

3.3.4.3 MEMORY FETCHING AND STORING

Until now, we have ignored the purpose of the phrase "(CYCLE)" at the end of all the ADDR loading micro-operations. This phrase initiates a memory access sequence for data fetching or storing.

Accessing memory is a multi-step sequence. The first step is to load the ADDR register with the desired address for

fetching or storing. In the case of a write to memory, the second step is to perform a RAM write using the microcode destination field.

```
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $100 X!
<< SOURCE=HOST DEST=RAM-W! >> $1234 X!
<< >> XGO
$100 MEMORY @ W.
                                ( Result is 1234 Hex)
```

This sequence will write the value 1234 Hex to Program Memory location 100 Hex. Location 100 Hex is a byte address aligned on a word boundary. All program memory addresses on the BINAR are byte addresses. The simulator sequence MEMORY @ fetches a value from program memory given an address. For now we will only consider the case of word-aligned 32-bit memory accesses, but all other writes on the Ram Data Bus follow the same pattern. Other methods of accessing memory will be discussed on the section covering the Data Alignment functional block.

The only restriction on writes to memory is that the DEST=RAM-WORD must be executed on the microcycle immediately following the ...(CYCLE) micro-operation. Note that since the BINAR assumes two-cycle memory, a second no-op microinstruction must be executed in order to complete the bus cycle.

In the case of a read from memory, the sequence after the ADDR=... microinstruction is somewhat different. A special buffer register called RDREG (which is discussed in the Data Alignment functional block section) captures the data read from memory and holds it until needed.

```
$6543 $200 MEMORY ! ( Store 6543 Hex at addr 200 Hex)
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $200 X!
<< >> XGO ( Wait for fetch)
<< >> XGO ( Wait for fetch)
<< SOURCE=RD DEST=HOST >> X@ W.
                                ( Result is 6543 Hex)
```

The simulator word sequence MEMORY ! stores a value at an address in program memory. The first microinstruction to be executed sets ADDR to 200 Hex and initiates a memory access cycle. The second microinstruction is a no-op that allows the memory read to take place. The absence of a DEST=RAM-WORD indicates that this is a memory read instead of a write. SOURCE=RD in the fourth microinstruction reads the result of the memory access from the RDREG. The microcycle

after an ADDR=... for a memory read is not restricted in usage. The only feature that need be understood is that a SOURCE=RD on the microcycle immediately following an ADDR=... microcycle will return the old contents of the RDREG (this actually is quite useful when some memory transfers). The SOURCE=RD micro-operation may be delayed as long as desired after the ADDR=... microinstruction, and multiple SOURCE=RD micro-operations may be performed on the same data.

An additional micro-operation that initiates memory cycles is the bus destination value CYCLE-RAM . This micro-operation initiates a microcycle without modifying the contents of the ADDR register. Any microinstruction that initiates a data fetch or store (using ADDR=... or CYCLE-RAM) may be overlapped with the last microcycle of another memory access. This is because the memory cycle initiation does not actually affect the memory bus pins, but merely fires up a finite-state machine that initiates a memory access starting on the following microcycle. As an example of how this feature may be exploited, consider the microcode to perform a memory-to-memory move from the address in SBASE+8 to the address in DBASE+8:

```
$5555 $108 MEMORY !
<< SOURCE=HOST ALU=B >> 8 X! ( Offset for accesses)
<< SOURCE=HOST DEST=SBASE >> $100 X!
<< SOURCE=HOST DEST=DBASE >> $200 X!
<< SOURCE=DHI ADDR=BUS+SBASE(CYCLE) >> XGO
<< >> XGO
<< SOURCE=DHI ADDR=BUS+DBASE(CYCLE) >> XGO
<< SOURCE=RD DEST=RAM-W! >> XGO
<< >> XGO
$208 MEMORY @ W.
                                ( Result is 5555 Hex)
```

In this example, the fourth microinstruction initiates a memory read. The sixth microcycle initiates a subsequent memory write while the read is taking place. The seventh microcycle gets the contents of the memory read (which is available on the third microcycle following the read initiation) and writes that value to memory (which must be accomplished on the first microcycle following the memory write initiation.)

Another important capability is the read/modify/write bus sequence. This sequence is provided to allow for an indivisible memory update function in a multi-processing environment.

```
$13 $40 MEMORY !
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $40 X!
<< RAM-RMW ALU=0 DHI[1] >> XGO
<< >> XGO
<< SOURCE=RD ALU=A+B+1 DHI[1] CYCLE-RAM >> XGO
<< SOURCE=DHI[1] DEST=RAM-W! >> XGO
<< >> XGO
$40 MEMORY @ W.
                                ( Result is 14 Hex)
```

The micro-operation RAM-RMW-WORD must be executed on the microcycle immediately following the initiation of the read cycle. The FRAM control line for the memory chips is kept activated until the
subsequent write cycle.

Memory addresses for RAD are taken from the ADDR register, except when fetching an instruction using the DECODE micro-operation (described later).

3.3.4.4 PAGE/NAR REGISTERS

The PAGE and Next Address Register (NAR) combination, along with the address incrementer fed from the address multiplexer output, form a 32-bit program counter for the BINAR. Whenever an instruction is being fetched from memory by a 20PS instruction, subroutine CALL, or next sequential instruction fetch, the PAGE/NAR pair are driven to the output of the RAM Address Multiplexer.

The reason that there are separate PAGE and NAR registers has to do with the BINAR instruction format. Each instruction only has a 21-bit field for specifying a Jump or subroutine CALL address. This field is extended with two low-order zero bits to form a 23-bit word-aligned address. This 23-bit value corresponds to the value stored in the NAR register. Sometimes, however, more than 23 bits of address space may be needed for program memory. This is where the PAGE register comes in. The PAGE register forms a 9-bit high order bit page number that is appended to the 23-bit NAR value for subroutine calls. Subroutine returns and sequential program execution correctly set the PAGE register to an appropriate value. Subroutine calls use the value of the PAGE register used for the previous instruction fetch. Microcode can accomplish a "far call" using a full 32-bit address that changes the PAGE register contents if desired.

PAGE and NAR may be read using the SOURCE=PAGE/NAR/CTL micro-operation. The 21-bit word address held in the NAR is

asserted on bits 2-23 of the value read using this bus source. The NAR and PAGE registers are a little tricky to modify, so microcode examples of reading and writing them will be deferred to a later section.

3.3.4.5 RETURN SAVE REGISTER

The Return Save register (RETURN-SAVE) is used to save the address of the most recently fetched instruction plus 4 to be written to the Return Stack in case a subroutine call is processed. RETURN-SAVE may be read using the SOURCE=RETURN-SAVE micro-operation. Writing a value into RETURN-SAVE is more difficult, and is covered in the following section on Instruction Fetching. Note that the "plus 4" operation is applied to the full the 32-bit address asserted on RAD before writing the value to RETURN-SAVE.

3.3.4.6 INSTRUCTION FETCHING

The finer points of instruction fetching are discussed in Section 4.0, Hardware Theory of Operation. However we can discuss here some techniques that can be used to load and store the PAGE, NAR, and RETURN-SAVE registers.

PAGE and NAR are loaded any time an instruction is loaded from memory. PAGE is always loaded with the highest 8 bits of the incremented RAD value. NAR is either loaded with the incremented RAD value (if the instruction being fetched is an Jump-To-Next or 2OPS), or bits 2-22 of the incoming instruction value (if the instruction being fetched is a CALL or EXIT). Instruction loading occurs on the microcycle after a DECODE micro-operation is performed, or on the microcycle when a LATCH-INSTRUCTION micro-operation is performed.

The DECODE micro-operation is the usual way to fetch and decode an instruction. If the currently executing instruction specifies a subroutine return, the RAM Address Mux is set to pass the value of RSREG. In all other cases the RAM Address Mux passes the value of PAGE/NAR. The DECODE micro-operation also initiates a memory read cycle to fetch the instruction. While the memory read is in progress, the increment-by-4 unit connected to the RAM Address Mux output computes the value of the address being fetched plus 4. This "plus 4" operation is carried out on the full 32-bit address value, potentially changing the memory page value.

When the instruction is returned from memory, it is loaded into the RDREG just like any other memory read operation.

However, it also controls the loading of the NAR. If the instruction that is being fetched is a subroutine CALL, bits 2-22 of the instruction are loaded into the NAR. If the instruction being fetched is a fetch-next-sequential instruction or is a two/instruction class, the output of the increment-by-4 unit is loaded into the NAR instead. The action in the case of a subroutine return being loaded is unimportant, but is in fact a load of bits 2-22 of the incoming instruction (it is unimportant, because the RS will supply the address for subroutine returns). In all cases, the PAGE register is also loaded with the high 9 bits of the increment-by-4 unit's output. This allows a subroutine return to automatically restore the PAGE register to the proper value, and allows sequential code to cross page boundaries. After the instruction fetch has taken place, RETURN-SAVE is automatically loaded with the increment-by-4 unit's output. This value serves as the subroutine return address in case the instruction that was just fetched turns out to be a subroutine CALL.

When using LATCH-INSTRUCTION, the memory read cycle is initiated like a data memory read, using an ADDR=BUS...(CYCLE) micro-operation. The address passed through the RAM Address Mux is that in the ADDR register. On the last cycle of the memory read (two cycles after the ADDR=... micro-operation), the LATCH-INSTRUCTION micro-operation is used, indicating that the RAM read is to be interpreted as an instruction fetch instead of a data fetch. Instructions fetching in this manner does not affect the Return Stack, but does update the contents of all other registers that are affected by instruction fetching (i.e. the Pending Instruction Latch and the CTL register). This capability is very useful for chip testing, and is essential for implementing conditional branches.

The following sequence shows how a jump to location 100 Hex is fetched from memory and the results on NAR and RETURN-SAVE. Note that RESET-SIM automatically sets PAGE to 0.

```
RESET-SIM
$100 $40 MEMORY !
                ( 100 Hex is a call to address 100 Hex)
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $40 X!
<< >> XGO
<< LATCH-INSTRUCTION >> XGO
<< SOURCE=RD DEST=HOST >> X@ W.
                          ( Result is 100 Hex)
<< SOURCE=PAGE/NAR/CTL DEST=HOST >> X@ W.
                          ( Result is 00000100 Hex)
```

```
<< SOURCE=RETURN-SAVE DEST=HOST >> X@ W.
                          ( Result is 44 Hex)
```

In this example, we store a CALL instruction to address 100 Hex (this instruction is value 100 Hex). The CALL instruction is stored at location 40 Hex. The instruction is then fetched and decoded. The result is that RDREG contains the value fetched from memory (100 Hex), PAGE contains the page address of 0, NAR contains the address of the instruction to be fetched next (100 Hex), and RETURN-SAVE contains the incremented-by-4 value of the address from which this instruction was fetched (44 Hex).

3.3.5 DATA ALIGNMENT FUNCTIONAL BLOCK

Figure 3-6 shows the Data Alignment functional block. This functional block contains multiplexers and control logic to allow access to memory bytes, half-words, short literals, and the pending instruction latch. This functional block interacts intimately with the memory access logic described in previous sections.

BINAR memory addresses are aligned on the unit of the access size. Word (32-bit) accesses should be aligned on even-word boundaries (byte addresses evenly divisible by 4). Half-word (16-bit) accesses should be aligned on even-half-word boundaries (byte addresses evenly divisible by 2). Byte accesses may be to any byte. In the event that the address given for the memory access is not appropriately aligned, the low-order bits of the address are disregarded. For example, address 107 would be truncated to address 104 for a word access, and 106 for a half-word access, while a byte access would use the value 107.

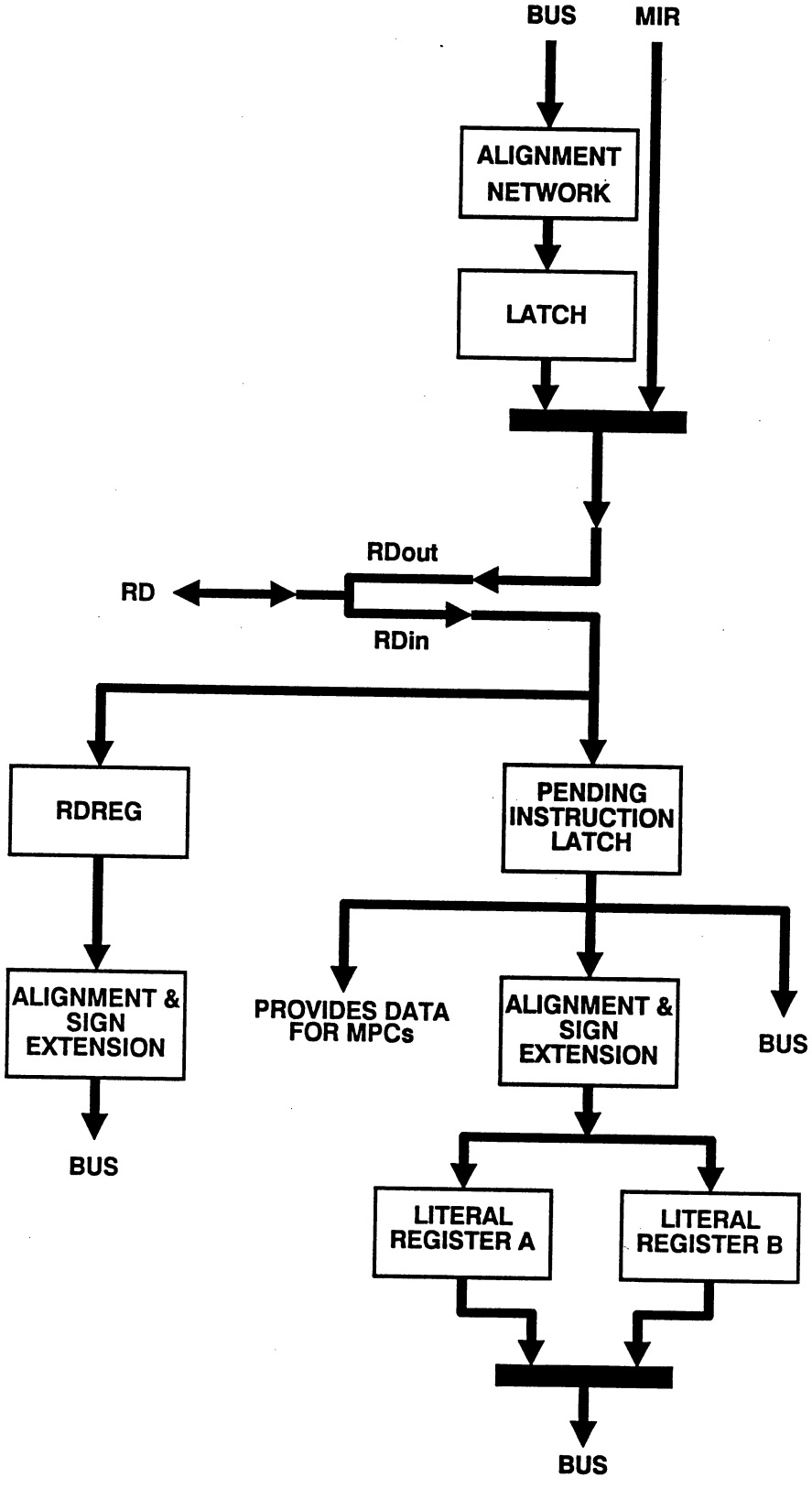
The BINAR places the lowest order byte of each word at the lowest memory location. Thus the hex value 12345678 if stored at word address 100 would have byte value 78 at address 100, value 56 at address 101, value 34 at address 102, and value 12 at address 103.

3.3.5.1 WRITING TO MEMORY

When doing any memory writes, the ADDR register is set and the memory cycle is initiated as described for word memory writes previously. The size of the data being written is specified by using one of the micro-operations DEST=RAM-W!, DEST=RAM-C!, DEST=ASIC-! or DEST=RAM-! to perform the write.

The write alignment network takes the lowest bits from the value on the Data Bus and places them on the appropriate

Figure 3-6. Data Alignment Functional Block.



data lines of the 32-bit RAM Data Bus. Only the Write-Enable pins for the bytes that are being written are asserted for the write operation.

```
$12345678 $100 MEMORY !
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $100 X!
<< SOURCE=HOST DEST=RAM-C! >> $99 X!
<< >> XGO
$100 MEMORY @ W.
                              ( Result is 12345699 Hex)
$12345678 $100 MEMORY !
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $101 X!
<< SOURCE=HOST DEST=RAM-C! >> $99 X!
<< >> XGO
$100 MEMORY @ W.
                              ( Result is 12349978 Hex)
$12345678 $100 MEMORY !
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $100 X!
<< SOURCE=HOST DEST=RAM-W! >> $7777 X!
<< >> XGO
$100 MEMORY @ W.
                              ( Result is 12347777 Hex)
$12345678 $100 MEMORY !
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $102 X!
<< SOURCE=HOST DEST=RAM-W! >> $7777 X!
<< >> XGO
$100 MEMORY @ W.
                              ( Result is 77775678 Hex)
$12345678 $100 MEMORY !
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $103 X!
<< SOURCE=HOST DEST=RAM-! >> $99 X!
<< >> XGO
$100 MEMORY @ W.
                              ( Result is 99 Hex)
```

The DEST=ASIC-! micro-operation performs exactly as a DEST=RAM-!, except the chip's external control pins are driven through an ASIC-bus cycle instead of a RAM Data Bus cycle.

3.3.5.2 READING FROM MEMORY

When doing any memory reads, the ADDR register is set and the memory cycle is initiated as described for word memory reads previously. The part that was not explicitly described before was that the specifier for the type of read must be given in the microcycle immediately following the initiation of the memory cycle. Since the default memory cycle is a word read from RAM, the cycle was simply left as

a no-op in previous examples. The size of the data being written may be changed from the default full-word by using one of the micro-operations RAM-C@, RAM-W@, ASIC-@ or RAM-RMW in the microcycle following the read initiation. When performing a read operation, the entire 32-bit word containing the data of interest is read into RDREG. The read alignment network then extracts the bits of interest and places them on the lowest bits of the Data Bus when a SOURCE=RD micro-operation is executed. The highest bits of the word are filled with zeros.

```
$12345678 $100 MEMORY !
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $100 X!
<< >> XGO ( Default operation is word)
<< >> XGO
<< SOURCE=RD DEST=HOST >> X@ W.
                          ( Result is 12345678 Hex)
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $103 X!
<< RAM-C@ >> XGO
<< >> XGO
<< SOURCE=RD DEST=HOST >> X@ W.
                          ( Result is 12 Hex)
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $100 X!
<< RAM-W@ >> XGO
<< >> XGO
<< SOURCE=RD DEST=HOST >> X@ W.
                          ( Result is 5678 Hex)
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $102 X!
<< RAM-W@ >> XGO
<< >> XGO
<< SOURCE=RD DEST=HOST >> X@ W.
                          ( Result is 1234 Hex)
```

The ASIC-@ micro-operation performs exactly as a RAM word read, except the chip's external control pins are driven through an ASIC-bus cycle instead of a RAM Data Bus cycle.

RAM-RMW behaves as a RAM word read, except that it leaves the memory bus active until a subsequent RAM write operation is performed to form an indivisible read/modify/write bus cycle.

Sign extended memory reads of byte and half-word values may be performed by using SOURCE=RD-SIGNED.

```
$123489AB $100 MEMORY !
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $103 X!
<< RAM-C@ >> XGO
<< >> XGO
```

```
<< SOURCE=RD-SIGNED DEST=HOST >> X@ W.
                              ( Result is 12 Hex)
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $101 X!
<< RAM-C@ >> XGO
<< >> XGO
<< SOURCE=RD-SIGNED DEST=HOST >> X@ W.
                              ( Result is FFFFFF89 Hex)
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $100 X!
<< RAM-W@ >> XGO
<< >> XGO
<< SOURCE=RD-SIGNED DEST=HOST >> X@ W.
                              ( Result is FFFF89AB Hex)
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $102 X!
<< RAM-W@ >> XGO
<< >> XGO
<< SOURCE=RD-SIGNED DEST=HOST >> X@ W.
                              ( Result is 1234 Hex)
```

3.3.5.3 SHORT LITERALS

The LITERAL REGISTER A holds a 21-bit literal/address field of the instruction currently being executed for CALL and JNEXT instructions. The literal value is treated as a 21-bit signed integer with sign extension to 32 bits. That means that the hardware shifts the value extracted from the instruction to the right two bits, so that bit 2 of the instruction forms bit 0 of the short literal field.

For 2OPS instructions, two literal values are extracted from the pending instruction latch, and stored in the LITERAL REGISTER A and LITERAL REGISTER B. LITERAL REGISTER A provides a sign-extended 32-bit value from bits 17-22 of the instruction for use with OPCODEA of that instruction. LITERAL REGISTER B provides a sign-extended 32-bit value from bits 2-7 of the instruction for use with OPCODEB of that instruction. LITERAL REGISTER A and LITERAL REGISTER B are automatically multiplexed by the hardware into the SOURCE=LIT bus source so that each of the opcodes in a 2OPS instruction is fed the proper sign-extended literal value.

As an example of how to use a short literal, consider the following example where an instruction with opcode 0 and a short literal field of 1234 Hex is used to add the literal to the top-of-stack element in DHI. Note that 1234 Hex must be multiplied by 4 and then OR'ed with 3 to form an instruction that jumps to the next sequential instruction with a literal field of 1234. We will assume that the instruction's opcode is 0 for this example.

```
RESET-SIM
$48D3 $40 MEMORY !
<< SOURCE=HOST ALU=B >> $3333 X!
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $40 X!
<< >> XGO
<< LATCH-INSTRUCTION >> XGO
<< DECODE >> XGO
<< SOURCE=LIT  ALU=A+B >> XGO
<< SOURCE=DHI DEST=HOST >> X@ W.
                          ( Result is 4567 Hex)
<< SOURCE=LIT DEST=HOST >> X@ W.
                          ( Result is 1234 Hex)
<< SOURCE=PAGE/NAR/CTL DEST=HOST >> X@ W.
                          ( Result is 44 Hex)
```

3.3.5.4 PENDING INSTRUCTION LATCH

The Pending Instruction Latch holds instructions read from memory until they are executed. This latch allows simultaneous fetch and execute of instructions. The Pending Instruction Latch is loaded on the microcycle after a DECODE micro-operation (as the new instruction is read in from memory), and at the end of a microcycle having a LATCH-INSTRUCTION micro-operation. The LATCH-INSTRUCTION micro-operation allows over-riding the automatically fetched next instruction with microcode, which makes possible conditional branching.

```
RESET-SIM
$12345678 $40 MEMORY !
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $40 X!
<< >> XGO
<< LATCH-INSTRUCTION >> XGO
<< SOURCE=I-LATCH  DEST=HOST >> X@ U.
                          ( Result is 12345678 Hex)
```

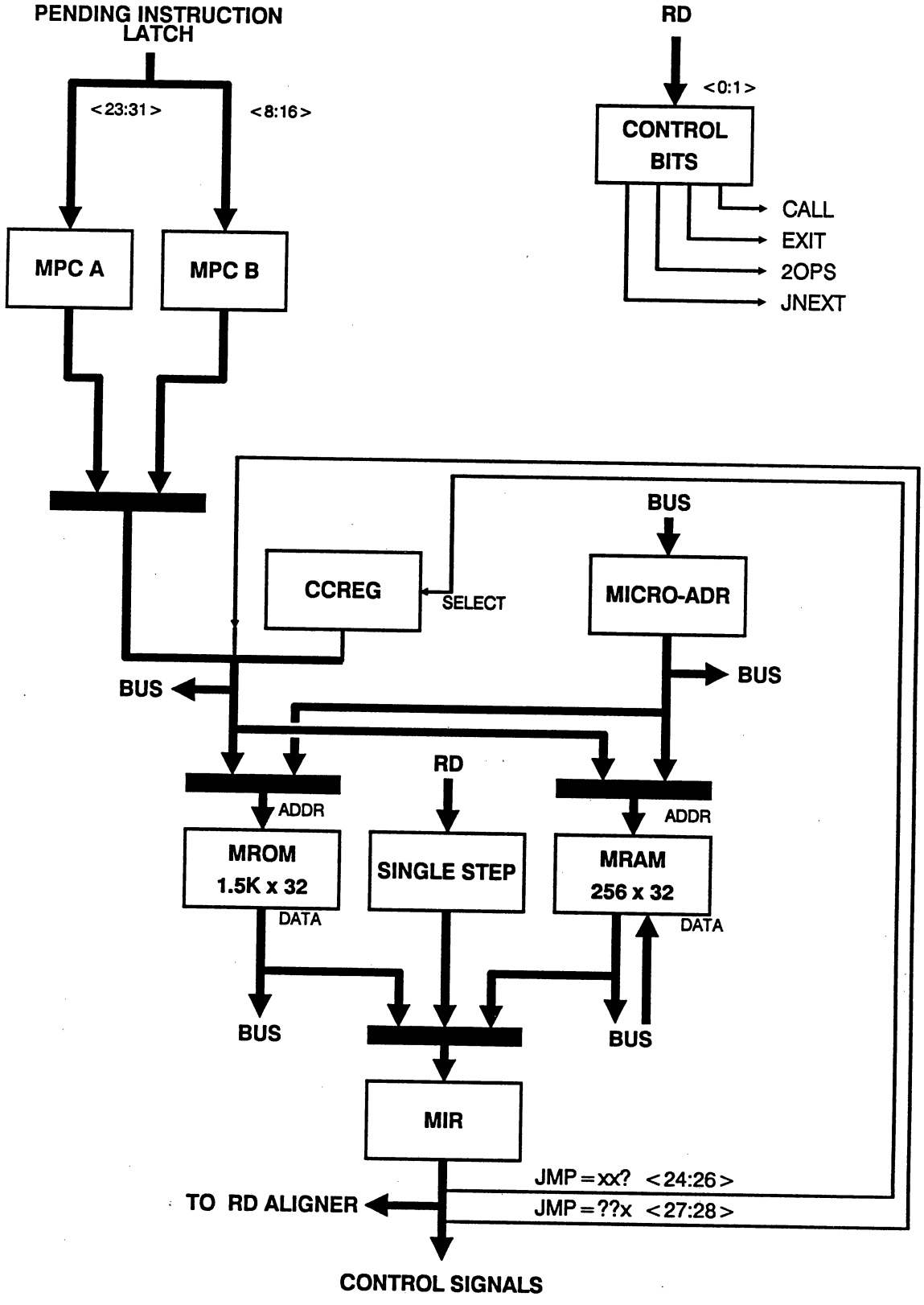
3.3.6 MICRO-SEQUENCER FUNCTIONAL BLOCK

Figure 3-7 shows the Micro-Sequencer functional block. This functional block contains the micro-program memories, various micro-program address registers, and the Microinstruction Register.

3.3.6.1 MICRO-PROGRAM COUNTER

The BINAR uses the two Micro-Program Counters (MPCA and MPCB) to generate the highest 9 bits of the micro-program address when executing microcode. "MPC" refers to MPCA in the following discussion, since MPCA is the primary MPC, and

Figure 3-7. Micro-Sequencer Functional Block.



is the MPC that is used everywhere except in 20PS instructions. MPCA and MPCB are automatically multiplexed by the hardware so that each of the opcodes in a 20PS instruction is fed the proper MPC opcode value.

The 9 bits loaded into the MPC exactly correspond to the 9-bit opcode of an instruction. Thus, "instruction decoding" for the opcode consists merely of copying a Pending Instruction Latch value into an MPC value. The 9 bits of the MPC form the high 9 bits of a 12-bit micro-program address for executing microcode. For this reason, micro-program memory is broken up into 512 8-word pages. Each of the possible 512 opcodes maps onto its own page.

Loading the MPC is accomplished by first loading the Pending Instruction Latch with a value, then performing a DECODE micro-operation to transfer the value into the MPC. Details of exactly how this decoding operation works are covered in Section 4.0, Hardware Theory of Operation.

```
RESET-SIM
$73800002 $40 MEMORY !
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $40 X!
<< >> XGO
<< LATCH-INSTRUCTION >> XGO
<< DECODE >> XGO
<< SOURCE=MISC DEST=HOST >> X@ 6 LSRN $0FF8 AND U.
                          ( Result is 738 Hex)
<< INC[MPC] >> XGO
<< SOURCE=MISC DEST=HOST >> X@ 6 LSRN $0FF8 WAND U.
                          ( Result is 740 Hex)
```

The value of the MPC is returned on bits 0-5 of the MISC bus source. The MPC is a counter, which may be incremented using the INC[MPC] micro-operation. This allows instructions to span consecutive pages in microcode memory if necessary.

When executing 20PS instructions, MPCA is loaded with the OPA field of the instruction (bits 23-31), while MPCB is loaded with the OPB field of the instruction (bits 8-16). During the first opcode execution, MPCA is selected to address microprogram memory. During the second opcode's execution, MPCB is selected to address microprogram memory.

3.3.6.2 CONDITION CODE REGISTER

The Condition Code Register (CCREG) is used to hold status information for use in conditional microcode branches. It

is updated at the end of every clock cycle, and is not directly accessible to high level language programs. The value of one of the 8 possible Condition Codes is selected by the 3-bit microinstruction register conditional branching field to form the lowest bit of the micro-program memory address for the next microinstruction. The 8 Condition Code values are: 0 (always 0), C (ALU carry-out), Z (ALU zero), S (ALU sign) , L (lowest bit of DLO), V (ALU overflow), P (pending interrupt), and 1 (always 1). The sense of these bits is that they are true if the associated condition is true. For example, ALU carry-out is 1 if there has been a carry-out from an arithmetic ALU operation.

The CCREG is clocked at the end of every microcycle, so the results of a condition are not available until the microcycle after the condition was generated. For example, an ALU carry-out is visible in the CCREG during the microcycle after the addition was performed. CCREG values do not stay valid for more than a single microcycle. However, the S, Z, and P bits do have some persistence because of the way the machine is used. The P bit indicates a pending interrupt, which will stay pending. The S and Z bits are based on the ALU mux output value. If the default ALU micro-operation of ALU=A is used, then the value in DHI is recycled, refreshing the values of S and Z on each clock cycle to reflect the sign and zero tests for the value currently in DHI.

The easiest way to test the CCREG values is to use the MISC source field for the Data Bus. This field brings the CCREG values out on bits 0-5 of the bus: C-bit 0, Z-bit 1, S-bit 2, L-bit 3, V-bit 4, and P-bit 5.

```
<< SOURCE=HOST ALU=B >> -1 X!
<< SOURCE=HOST ALU=A+B >> 1 X!
<< SOURCE=MISC DEST=HOST >> X@ $3F AND W.
                          ( Result is 3 Hex)
<< SOURCE=HOST ALU=B >> $77777777 X!
<< SOURCE=HOST ALU=A+B >> $66666666 X!
<< SOURCE=MISC DEST=HOST >> X@ $3F AND W.
                          ( Result is 14 Hex)
```

In the first example, bits 0 and 1 are set, indicating that the addition of -1 to 1 produced a carry-out and a zero result. In the second example, bits 2 and 4 are set, indicating that the result produced a negative result and an arithmetic overflow.

The real use for the CCREG is in conditional microcode branching, which will be described later. The CCREG is used

to produce the lowest bit of the micro-program address when executing microcode.

3.3.6.3 JMP MICRO-OPERATIONS/CONDITIONAL MICRO-BRANCHING

We have seen that the MPC forms the high 9 bits of the micro-program address. The CCREG has a multiplexer that is used to form the lowest-order bit of the micro-program address. This multiplexer allows selecting one of the 8 possible Condition Code values as address bit 0.

The specification of the CCREG mux selection is made using the JMP=... micro-operations. The eight possible selections controlled by bits 24-26 of the microinstruction format are: JMP=xx0, JMP=xxC, JMP=xxZ, JMP=xxS, JMP=xxL, JMP=xxV, JMP=xxP, and JMP=xx1. 0 stands for always 0, C for ALU carry-out, Z for ALU zero, S for ALU sign, L for DLO lowest bit, V for ALU arithmetic overflow, P for interrupt pending, and 1 for always 1. The "xx" stands for any one of "00", "01", "10", or "11" which form bits 1 and 2 of the microprogram address. These 2 bits are directly fed from bits 27-28 of the microinstruction.

The complete list of all possible JMP=... micro-operations is, therefore:

JMP=000	JMP=010	JMP=100	JMP=110
JMP=00C	JMP=01C	JMP=10C	JMP=11C
JMP=00Z	JMP=01Z	JMP=10Z	JMP=11Z
JMP=00S	JMP=01S	JMP=10S	JMP=11S
JMP=00L	JMP=01L	JMP=10L	JMP=11L
JMP=00V	JMP=01V	JMP=10V	JMP=11V
JMP=00P	JMP=01P	JMP=10P	JMP=11P
JMP=001	JMP=011	JMP=101	JMP=111

So, we have a 12-bit micro-program address used to fetch the next microinstruction from micro-program memory during program execution. This microaddress is formed by a combination of the MPC value and control values from the current microinstruction. Bit 0 of the address is the output of the Condition Code Multiplexer. Bits 1 and 2 of the address are directly taken from bits 27 and 28 of the current microinstruction. Bits 3-11 of the address are taken from the current MPC value. The SOURCE=MISC micro-operation asserts the complete 12-bit microaddress on bits 0-11 of the Data Bus.

```
RESET-SIM
$12000002 $40 MEMORY !
```

```
    << SOURCE=HOST ADDR=BUS+0(CYCLE) >> $40 X!
    << >> XGO
    << LATCH-INSTRUCTION >> XGO
    << DECODE ALU=-1 >> XGO
<< SOURCE=MISC JMP=000 DEST=HOST >> X@ 6 LSRN $0FFF AND W.
                                  ( Result is 120 Hex)
<< SOURCE=MISC JMP=111 DEST=HOST >> X@ 6 LSRN $0FFF AND W.
                                  ( Result is 127 Hex)
<< SOURCE=MISC JMP=01S DEST=HOST >> X@ 6 LSRN $0FFF AND W.
                                  ( Result is 123 Hex)
<< SOURCE=MISC JMP=10Z DEST=HOST >> X@ 6 LSRN $0FFF AND W.
                                  ( Result is 124 Hex)
```

Remember that the default ALU operation is ALU=A, so the -1 value is continually recirculated forcing the sign condition to be true and the zero condition to be false.

3.3.6.4 MICRO ADDRESS REGISTER

The Micro Address Register (MICRO-ADR) is an alternate source for providing addresses to the micro-program memories. It is used to provide a second address for reading and writing these memories during program execution.

```
    << SOURCE=HOST DEST=MICRO-ADR >> $12345678 X!
    << SOURCE=MISC DEST=HOST >> X@ W.
                                  ( Result is 678 Hex)
```

Only the lowest 12 bits of the value are used by the MICRO-ADR register. All 12 bits are read and written at the same time. The JMP fields and MPC have nothing to do with the value in MICRO-ADR.

3.3.6.5 MICROCODE RAM AND ROM

The Micro-Program Memory is a combination of RAM (called the MRAM) and ROM (called the MROM) that may be up to 4K words of 32-bit memory. As discussed previously, Micro-Program Memory is broken up into 512 pages of 8 words, with each page corresponding to an opcode. The MRAM and MROM may each take their addresses from either the MPC/JMP address source or the MICRO-ADR register.

In the BINAR, there are 2K words of MROM that are mapped into addresses hex 000-7FF of the Micro-Program Memory space, giving 256 pages. There are also 128 words of MRAM mapped into addresses hex F80-FFF of the Micro-Program Memory space, giving 16 pages.

The MPC/JMP address source always takes priority for accessing Micro-Program Memory. That means that the value in the MPC is always used as the address source for either MRAM or MROM, as appropriate. The Micro-Program Memory address mux that is not required by the MPC is used to route the MICRO-ADR register to supply memory addresses. That means that if the MPC register has a page address between 0 and 256, the MROM address mux selects the MPC/JMP address source, while the MRAM address mux selects MICRO-ADR. If the MPC register has a page address between 496 and 511, the MRAM address mux selects the MPC/JMP address source, while the MROM address mux selects MICRO-ADR. If the MPC register has a page address between 256 and 495 inclusive, a TRAP opcode (opcode 3 from microcode ROM) is executed by overriding the MPC value with the value 3.

In operation, this means that microcode residing in the MROM can be executed (which means that it is being addressed by MPC/JMP) while reading or writing data addressed by MICRO-ADR from/to the MRAM. Similarly, microcode residing in the MRAM can be executed while reading data addressed by MICRO-ADR from the MROM.

MRAM and MROM are read and written simply by using bus sources and destinations once the addresses have been properly set up.

```
RESET-SIM
$12345678 $122 U-ROM ! ( Init simulator MROM)
$12000002 $40 MEMORY !
<< SOURCE=HOST ADDR=BUS+0(CYCLE) >> $40 X!
<< >> XGO
<< LATCH-INSTRUCTION >> XGO
<< DECODE >> XGO
<< SOURCE=HOST DEST=MICRO-ADR >> $FF0 X!
                              ( MRAM address)
<< SOURCE=HOST DEST=MRAM >> $88776655 X!
<< SOURCE=MROM JMP=010 DEST=DLO >> XGO
<< SOURCE=DLO  DEST=HOST >> X@ U.
                              ( Result is 12345678 Hex)
<< SOURCE=MRAM DEST=DLO >> XGO
<< SOURCE=DLO  DEST=HOST >> X@ U.
                              ( Result is 88776655 Hex)
```

Note the use of DLO as an intermediate holding register for MROM and MRAM sourcing; this is required because of the use of synchronous memories for implementing MROM and MRAM. In this example, the instruction hex 12000002 loads an opcode

value of hex 24 (decimal 36) into the MPC. Opcode 36 offset 2 is address 36 * 8 + 2 = 290 = hex 122. Thus, a destination of MROM while the MPC value is 36 and the JMP field selects offset 2 returns the correct result. Meanwhile, the MICRO-ADR value selects address hex FF0 which is opcode 510 offset 0 for a write followed by a read. Notice that the MRAM read and write do not affect MROM operation. It should be noted that the high bits of the MICRO-ADR register do not participate in the address steering process. A MICRO-ADR value that is meant to address an MROM location will be truncated and mapped into the MRAM address space if MPC/JMP also addresses an MROM location.

3.3.6.6. MIR AND SINGLE-STEP REGISTERS

The Microinstruction Register (MIR) is a 32-bit register that sends control signals throughout the CPU/32 hardware. Various bits of the MIR control different control fields as shown in Figure 3-7. The MIR is loaded with a new value on every microcycle.

The SINGLE-STEP register provides a buffer register for feeding microinstructions to the system via the TEST pin microinstruction load sequence.

The MIR may be loaded directly from the outside world using the TEST pin. From the Microassembler/Simulator, a special function does this loading automatically. Every time an X@, X!, or XGO command is processed, the MIR is loaded before cycling the clock.

When single-stepping microinstructions by using the TEST pin, the SINGLE-STEP register is automatically selected as the source from which to load the MIR value. This loading of the MIR through the SINGLE-STEP register is needed because of detailed hardware design considerations, and is related to the need to increment/decrement the stack pointers at the beginning of the microcycle.

When executing microinstructions in normal operation, the source of the next microinstruction is selected by the MPC/JMP address, and the mux feeding the MIR is used to select between MROM and MRAM.

3.3.7 SYSTEM DATA BUS

We have hinted at how the System Data Bus (usually just called the Data Bus) works, but until now we have not gone

into specifics. The Bus is the heart of the BINAR. It provides a path which can connect almost any two registers in the system with each other for data transfers. It is what provides the great flexibility of the BINAR when executing microcode.

Bus operation is controlled by the SOURCE and DESTINATION fields of the microinstruction (bits 0-4 and 5-9). The Bus may be conceptually thought of as a tristate bus connecting one of 32 bus sources to one of 32 bus destinations. In reality, the bus is constructed of a 32-to-1 multiplexer for speed reasons. Also, as will be seen, there may be more than one simultaneous bus destination specified by using special values of micro-operations other than the bus destination field.

3.3.7.1 BUS SOURCES

There are 32 possible bus sources as indicated in Figure 2-3. Most of them we have already explored. The only ones that we have neglected are a few sources that place constant values on the bus and the TRAP bus source.

```
<< SOURCE=0  DEST=HOST >> X@ W.
                          ( Result is 0 Hex)
<< SOURCE=-1 DEST=HOST >> X@ W.
                          ( Result is -1 Hex)
<< SOURCE=4  DEST=HOST >> X@ W.
                          ( Result is 4 Hex)
```

Need we say more about these three bus sources?

Obviously, only one bus source at a time may be specified. There are, however, several direct connections that bypass the Data Bus for efficiency. These include: the DHI to DS bypass, the RS to Memory Address Mux connection, and the RETURN-SAVE to RS connection.

Most bus sources are simply 32-bit values of the register specified by the micro-operation. Those bus sources that do not fit this description are described below:

SOURCE=HOST routes the Program Memory Data pins directly to the Data Bus without passing through the RDREG. This allows direct assertion of data from the outside world for single-stepping microcode. This bus source must be used to make X! work properly.

SOURCE=LIT returns a sign-extended 32-bit value of the 21-bit or 6-bit signed short literal value.

SOURCE=RD & SOURCE=RD-SIGNED return different versions of the RDREG that depend on whether byte, half-word, or other reads were specified by previous microinstructions.

SOURCE=PAGE/NAR/CTL returns the 2-bit CTL register on bits 0-1, the 21-bit NAR register on bits 2-22, and the 9-bit PAGE register value on bits 24-31.

SOURCE=MRAM and SOURCE=MROM do not behave as expected in single-step mode, because of the use of synchronous memories. They should not be used as a source when using DEST=HOST. Instead, use DLO or some other register as an intermediate holding point.

SOURCE=DP & SOURCE=RP return 6-bit values extended to 32 bits with zeros.

SOURCE=DP-LIMIT & SOURCE=RP-LIMIT return the upper bound 6-bit value (DPTOP/RPTOP) on bits 16-31 and the lower bound value (DPBOT/RPBOT) on bits 0-15.

SOURCE=MISC returns the current CCREG bits as follows: C-bit 0,
Z-bit 1, S-bit 2, L-bit 3, V-bit 4, P-bit 5, the current MPC/JMP selected address bits are returned in bits 6-17, MICRO-ADR on bits 18-29, and bits 30-31 are returned as zero.

MULTIPLY-STEP & DIVIDE-STEP both have very specific control functions. They also route the DSREG value to the Data Bus.

3.3.7.2 BUS DESTINATIONS

There are 32 possible bus destinations that may be selected with bits 5-9 of the microinstruction as indicated in Figure 2-3. We have already described the usage of all of these. Additionally, DP, RP, and DLO may be specified as bus destinations in parallel with other bus destinations since they are specified using independent micro-operations. The one other bus destination that is available on every microcycle is the B side of the ALU.

Many of the bus destination fields are used partially or solely to perform a variety of control functions. These functions may be summarized as follows:

DEST=HOST routes the Data Bus value directly to the Memory Data pins. This allows direct assertion of data to the

outside world for single-stepping microcode. It must be used with X@ for correct operation.

DEST=DP-LIMIT & DEST=RP-LIMIT set the upper bound 16-bit value (DPTOP/RPTOP) with bits 16-31 and the lower bound value (DPBOT/RPBOT) with bits 0-15.

DEST=MICRO-ADR sets the 12-bit MICRO-ADR register with the lowest 12 bits of the Data Bus.

DEST=DS-FROM-DHI routes the output of the DHI mux directly to the DS without using the Data Bus at all. A SOURCE=... may then feed data to the DP, RP, DLO, or DHI (via the ALU) registers simultaneously.

RAM-RMW, RAM-C@, RAM-W@, and ASIC-@ all specify memory read control signals.

DEST=RAM-C!, DEST=RAM-W!, DEST=RAM-!, and DEST=ASIC-! all specify memory write control signals as well as writing the lowest 8, 16, or 32 bits of the Data Bus to memory.

LATCH-INSTRUCTION treats the Memory Data Bus as an incoming instruction and sets the Pending Instruction Latch, PAGE/NAR, and RETURN-SAVE accordingly.

CYCLE-RAM initiates a memory cycle using the current value in ADDR.

ADDR=BUS-4(CYCLE), ADDR=BUS+0(CYCLE), ADDR=BUS+SBASE(CYCLE), ADDR=BUS+DBASE(CYCLE), and ADDR=BUS+FP(CYCLE) all route the Data Buss to ADDR while doing an addition and initiating a memory cycle.

DEST=DP & DEST=RP transfer 6-bit values from bus bits 5-0 to the selected stack pointer register. Bits 31-6 are "don't cares" in these transfers.

3.4 ON-CHIP ASIC PERIPHERALS

The BINAR core processor has no on-chip ASIC peripherals. The commercial version will have a more sophisticated interrupt controller, counter/timers, and other peripherals.

4.0 HARDWARE THEORY OF OPERATION

4.1 HARDWARE INTERFACE DESCRIPTION

4.1.1 CHIP PINOUT

The BINAR chip is defined for an 84-pin package as shown in Figure 2-5, BINAR 84-Pin Chip Pinout. The two major pin groups are the RAM Data Bus pins RD<0:31> and the RAM Address pins RAD<2:23>. Other groupings of signals are the external control signals, memory interface signals, interrupt pins, and power supply pins. I/O is TTL-compatible, with 0 Volts representing logic value 0 and +5 Volts representing logic value 1.

4.1.1.1 EXTERNAL CONTROL SIGNALS: OSC, RESET, AND TEST

The external control signals consist of the RESET pin, TEST pin, and OSC pin.

The OSC pin (asserted low) is the system Oscillator, which directly generates the clock edges. When running the system at operating speed, OSC acts as a 2x clock input, with internal clock edges generated for each rising edge of OSC. When in TEST mode, OSC must be held high, then pulsed low to read or write MIR values (OSC is only pulsed once in TEST mode for each operation). Since the BINAR is fully static CMOS, OSC may be halted or run at any speed desired up to the maximum speed for the chip.

The RESET pin (asserted low) resets the system. This involves loading initialization values into many of the registers in the system while the pin is asserted and OSC is running. Once the pin is released, the chip begins execution at program memory location $FFFFFFF0. Section 4.7 on system initialization covers the effects of asserting the RESET pin in more detail. The RESET pin has a Schmitt trigger input so that it may be directly connected to a resistor/capacitor network to generate a power-on reset pulse.

The TEST pin (asserted low) is used to load and read back MIR values for single-stepping microcode and chip testing. When the TEST pin is not asserted, microcode execution proceeds as desired. In order to use the TEST pin, the OSC pin should first be halted in the high state, then TEST

should be asserted. The first low-going OSC pulse in TEST mode writes an MIR value via the RD bus, while subsequent (optional) OSC pulses read the MIR value onto the RD bus.

4.1.1.2 MEMORY INTERFACE SIGNALS

The majority of BINAR pins are dedicated to the memory interface. The BINAR has a 22-bit Ram Address Bus (RAD), a 32-bit Ram Data Bus (RD), and several control signals. The RD bus is used not only for memory reads/writes, but also for ASIC bus transactions (and I/O bus address space that uses RD and RAD) and interrupt acknowledge cycles.

RD is a bidirectional tristate bus for transferring information on and off the chip. RAD supplies word addresses (i.e. it does not supply bits 0 or 1 of the byte address to memory).

FRAM# is designed to be connected to CS# pin of standard static RAM chips. It must be decoded externally to support multiple banks of SRAM chips.

ASIC# is an enable for ASIC devices that communicate with the BINAR. It is intended to be used, when coupled with RAD decoding, as a chip select signal for ASIC devices. It is impossible for FRAM# and ASIC# to be active at the same time.

OE# is the output enable for FRAM and ASIC peripheral chips. It is designed to be connected directly to the output enable pins of SRAM devices. OE# is asserted during bus read operations.

WE0#, WE1#, WE2#, and WE3# are the write enable signals designed to be connected to the WE# pins of all RAM chips and ASIC devices system-wide. WE0# should be connected to bits 0-7 of memory, WE1# to bits 8-15, WE2# to bits 16-23, and WE3# to bits 24-31. One, two, or four of these WE signals are asserted for byte, half-word, and word write cycles, respectively.

WAIT# is an asynchronous wait-state input to the BINAR for externally generated wait states.

INTR# and NMI# are the two asynchronous interrupt inputs used to generate maskable and non-maskable interrupts.

INTA# is an output that signals an interrupt acknowledge cycle is taking place on the RD/RAD pins. INTA is asserted

on the second clock cycle of the memory fetch to the interrupt vector activated. The type of interrupt being responded to may be determined by examining the low bits of RAD during the INTA pulse.

4.1.1.3 POWER AND GROUND

The only other pins are the power and ground pins. Power (VCC) is +5V, ground (VDD) is 0V.

4.2 NORMALIZED MICROCODE OPERATION

This section deals with the actions that take place during normal microcode operation when running the processor at full speed. Figure 4-1, Clock Cycle Timing, shows how the different clock phases appear in a less idealistic manner than that depicted in Figure 2-4, Clock Cycle Overview.

Each clock cycle consists of a full cycle of the signal CLOCK. All references to clock signals refer to the rising edge of the signal when using the terminology "...is clocked by...", "...triggers...", "...CLOCK loads the register..." or any other reference to an edge-triggered event. References to clock signals that make use of level-sensitive properties like transparent latches and synchronous memory clocks refer to the signal pulsing high. For example, a transparent latch whose clock enable was CLOCK would be transparent during the period when CLOCK was high.

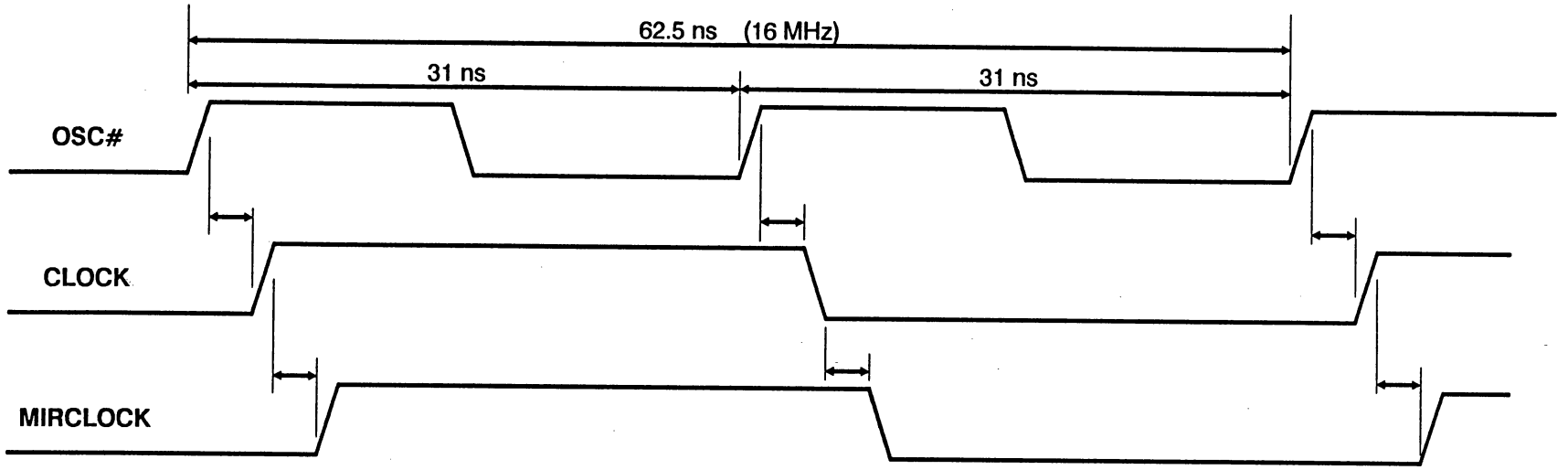
The OSC pin provides the raw edges used to generate other signals. It is used in a divide-by-two mode to generate the edges of CLOCK, and in a non-divided mode to control transfers while the TEST pin is asserted.

The CLOCK signal is the main system clock. A clock cycle starts at the rising edge of CLOCK and ends at the next rising edge. Most registers in the system are clocked using the CLOCK signal. MIR-CLOCK is a slightly delayed version of CLOCK used for driving the microinstruction register and the microcode memory units.

4.2.1 ACTIONS BASED ON CLOCK

CLOCK is the usual signal used for clocking registers in the BINAR. It is the basis for indicating what a "clock cycle" otherwise known as a "microcycle" is. The timing of all other signals within the chip is measured with respect to the rising edge of CLOCK.

Figure 4-1. Clock Cycle Timing.



The register macros used in the BINAR are edge-triggered D-registers with a clock enable input. This clock enable allows a low register clock value to be seen by the rising-edge detection logic only when the enable is activated. (Once again, assertion voltage levels are ignored for the purposes of this discussion.) An important consideration is that these enable signals are generated by the destination field of the microcode using a 5 to 32 decoder. Therefore, false clock pulses are possible if CLOCK goes low before the destination field is properly decoded. Using a 50% duty cycle signal on the clock allows ample time for decoding the enable signals before CLOCK goes low.

One may think of the high half of CLOCK as the time when the bus sources are fetched onto the bus, and the low half of CLOCK as the time when the results are placed into the bus destination. Essentially all data registers in the system are clocked by the rising edge of CLOCK. All RAMs are enabled using CLOCK, so that they precharge during the high phase of CLOCK and are read/written during the low phase of CLOCK. The only exceptions to this rule are the microinstruction resources such as the MIR, MRAM, and MROM, which use MIRCLOCK instead of CLOCK.

4.2.2 ACTIONS BASED ON MIR-CLOCK

MIR-CLOCK is used solely to control the microsequencer. It is a copy of CLOCK that is delayed very slightly. While CLOCK provides the edge for most registers within the system, MIR-CLOCK provides the edge that loads the MIR from MRAM/MROM or from the SINGLE-STEP register. The reason that it is slightly delayed from CLOCK is to avoid race conditions between the CLOCK rising edges and the changing output of the MIR.

The rising edge of MIR-CLOCK loads the MIR from MRAM, MROM, or the SINGLE-STEP register. If the TEST pin has been asserted since the last rising edge of CLOCK, the MIR source is forced to the SINGLE-STEP register for single-stepping. Otherwise, the machine is in the free-running mode. In this mode the MIR source is appropriately selected to MRAM or MROM depending on the page address in the MPC.

MIR-CLOCK also serves as the CE input to both MRAM and MROM, which are synchronous memories. The high portion of MIR-CLOCK is used as precharge time, while the low portion of MIR-CLOCK is used as access time.

4.3 SINGLE-STEP MICROCODE OPERATION

Single-stepping microcode is not much different from running the machine in normal operating mode. By single-stepping microcode we mean the process of loading a microinstruction value from an external host and executing it directly. Since the BINAR is a fully static design, the OSC input may be cycled as slowly as desired, even to the point of feeding in individual pulses. However, this is not considered single-stepping for the purposes of our discussion since no direct control of the MIR values being executed is asserted by the host. For the following sections, single-stepping shall refer to the process of loading a microinstruction directly into the MIR, then cycling the clock to execute that microinstruction.

Single-stepping microcode may be done at any time, including in the middle of executing a program on the BINAR. In fact, "service calls" to the host processor on BINAR plug-in boards may be accomplished using single-stepped microcode routines to access internal chip values (although the currently available evaluation board only uses this mode of operation for loading the bootstrap loader code that is then used to load the Forth kernel). In order to accomplish single-stepping, the OSC pin must be stopped and held high. This should be synchronized with OSC to avoid generating runt OSC pulses into the BINAR.

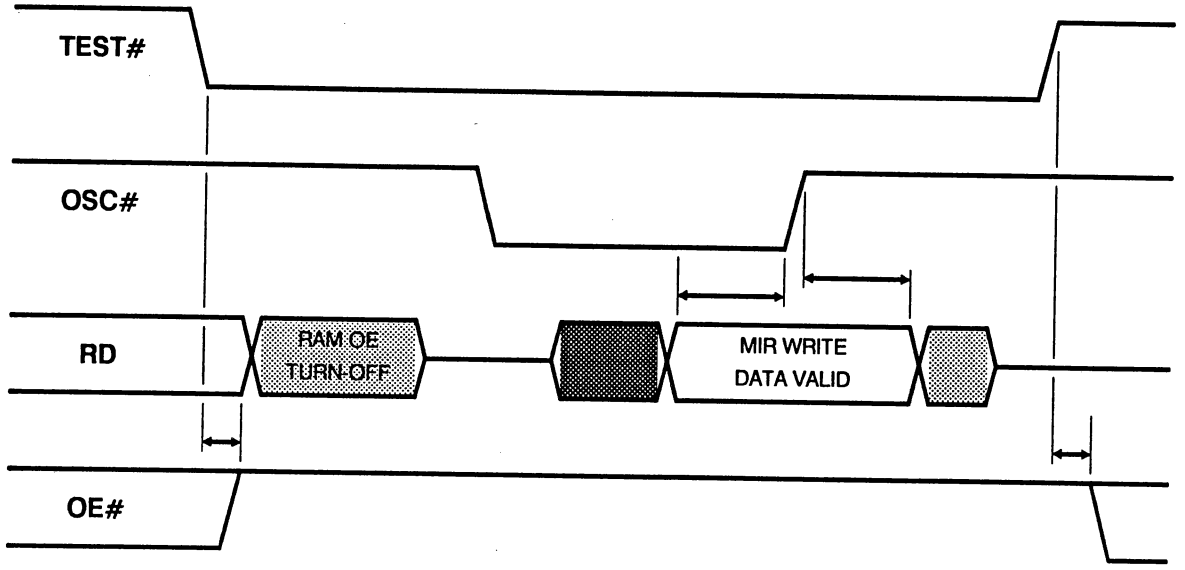
When the OSC input is stopped and held high, the system is frozen in whatever part of the clock cycle it is in. When the TEST pin is subsequently asserted, the system is synchronized to be in the low half of the clock cycle.

4.3.1 LOADING AND READING A MICROINSTRUCTION

The TEST pin is used to load and read back MIR values for single-stepping microcode and chip testing. When the TEST pin is not asserted, microcode execution proceeds as desired. In order to use the TEST pin, the OSC pin should first be halted in the high state, then TEST should be asserted.

Figure 4-2, TEST pin - Write MIR, shows the sequence used for writing a value into the SINGLE-STEP register in preparation for a single-step execution. Once TEST is asserted, the OSC pin is brought low, and the host asserts the new MIR value to be loaded onto the BINAR's RD pins. The data is captured by the BINAR on the rising edge of OSC,

Figure 4-2. TEST Pin - Write MIR.



then the TEST pin is de-asserted. It is important that the OSC pin be completely high before TEST is de-asserted to avoid a false clock pulse to the chip. Activating the TEST pin disables the OE pin of the BINAR, so that there is no possibility of bus contention if the BINAR is in the middle of a memory cycle.

Figure 4-3, TEST pin - Read MIR, shows the sequence used for reading the MIR value. The TEST pin is asserted, and OSC is cycled low one time to write an MIR value as described above. The second OSC pulse during the same TEST pin assertion will cause the freshly written MIR value to be read back out on the RD bus. The rising edge of the second OSC pulse is an excellent time for the host system to latch the values asserted by the BINAR onto its RD bus.

4.3.2. EXECUTING A MICROINSTRUCTION

Once the SINGLE-STEP latch has been loaded with a value using the MIR! function of the Microassembler, the microinstruction may be executed. Figure 4-4, Single-Step Clock Timing, shows the general format of signals for a single-stepped microinstruction.

Figure 4-5, X@ Timing, shows the sequence of events required to load and execute a single microinstruction in single-step mode. To begin the process, the TEST pin is asserted and an OSC pulse is used to load the MIR with the microinstruction that is to be executed. Once the TEST pin is de-asserted, a pair of OSC pulses is used to cycle the clock. At the rising edge of CLOCK, the new microinstruction is loaded into the MIR and begins execution (actually, this occurs with MIRCLOCK, which is a few nanoseconds after CLOCK). If a DEST=HOST bus destination has been selected or if a RAM write is in progress, the RD bus will have data asserted from the time the CLOCK pulse begins until the next TEST falling edge. An appropriate place for a host to sample data is at the
falling edge of the clock pulse.

The sequence of events described corresponds to the events triggered by an X@ Microassembler operation. Any microinstruction may be executed using this method, including RAM reads and writes.

During single step operation, CLOCK and MIR-CLOCK, do the same things they do during normal operation. The only interesting feature that was glossed over in the normal operation section is the fact that the falling edge of CLOCK

Figure 4-3. TEST Pin - Read MIR.

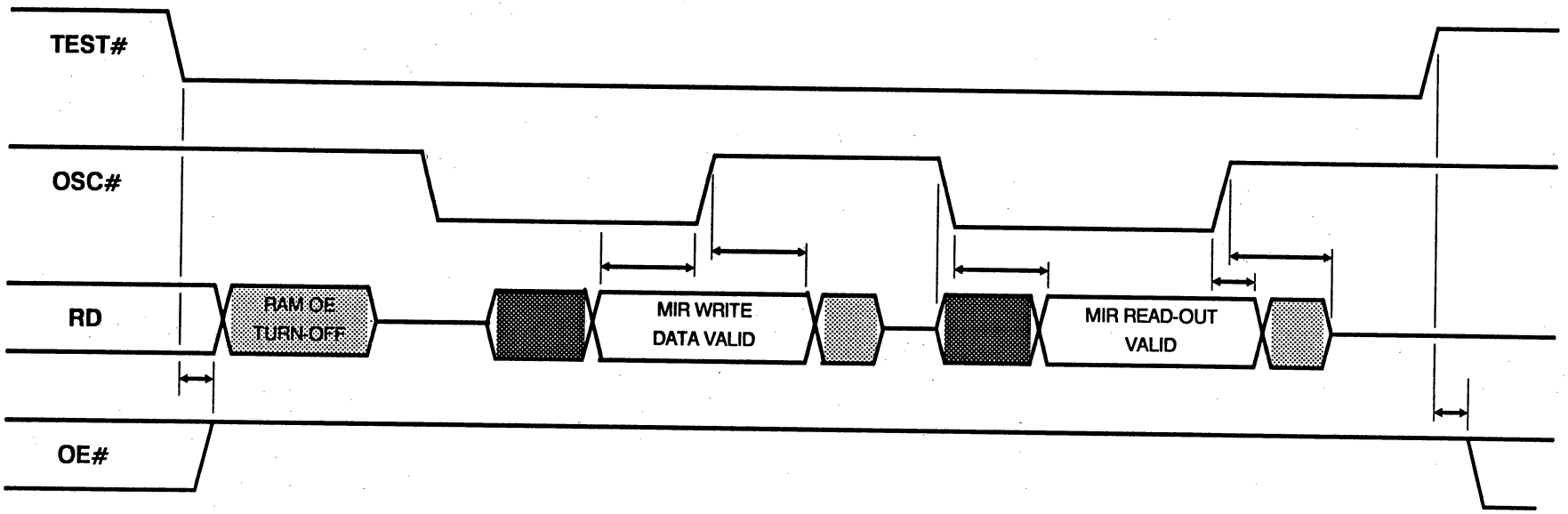


Figure 4-4. Single-Step Clock Timing.

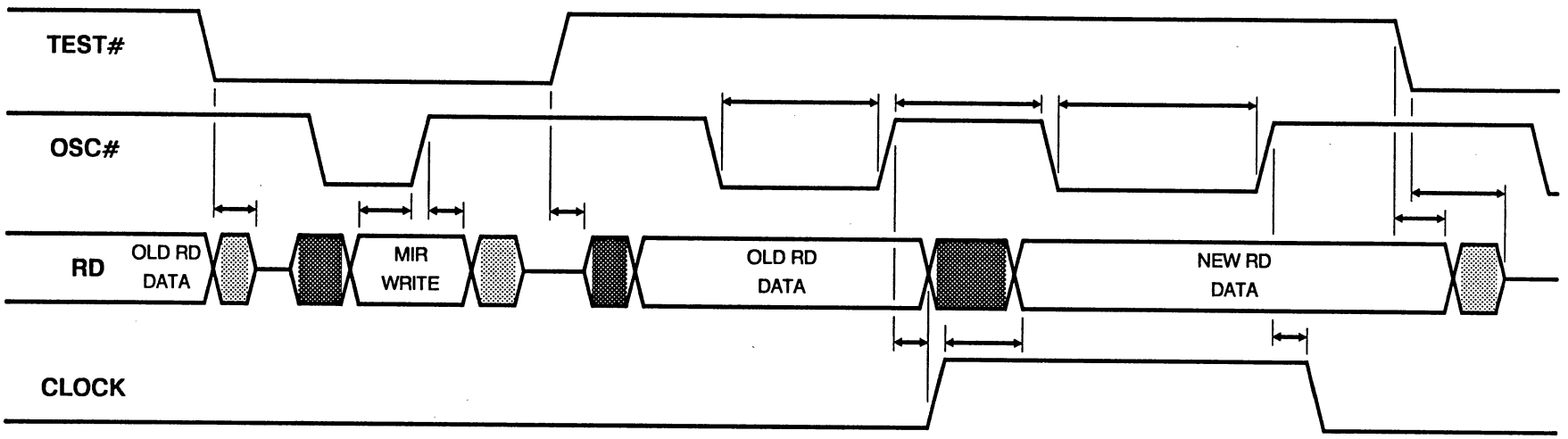
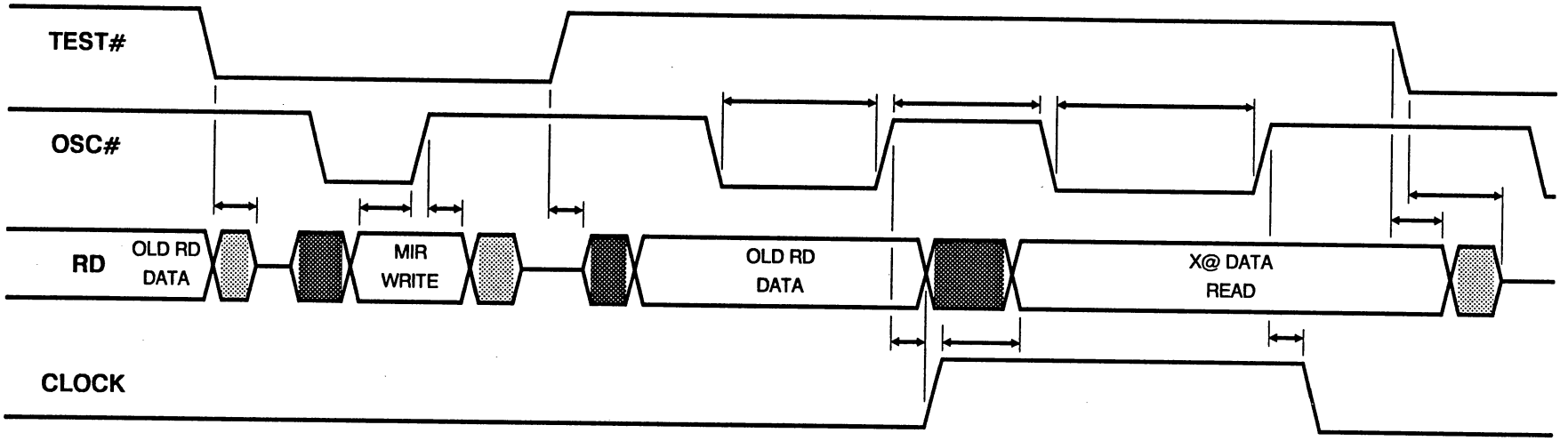


Figure 4-5. X@ Timing.



resets the flip-flop that indicates the microinstruction was single-stepped. This is necessary to prepare the system to resume normal operation from MROM and MRAM after the single-step cycle. The implication of this is that a single-stepped microinstruction may only be used for one cycle, then the SINGLE-STEP register must be reloaded for the next clock cycle.

The astute reader will notice that there is no rising edge of CLOCK at the end of the microcycle. This means that while the values generated for the Data Bus and the ALU outputs for the microinstruction will be correct, they will not be captured as destinations anywhere. This is OK, because the outside world monitors the RD bus, not the actual registers. The beginning of the next microcycle will finish the job of loading the bus destinations when it asserts its rising CLOCK edge. If the next microcycle is single-stepped, the MIR value is preserved until MIR-CLOCK since the host only loads the SINGLE-STEP register. If the next microcycle is executed in normal mode, the rising edge of CLOCK will precede the MIR-CLOCK load from MROM/MRAM.

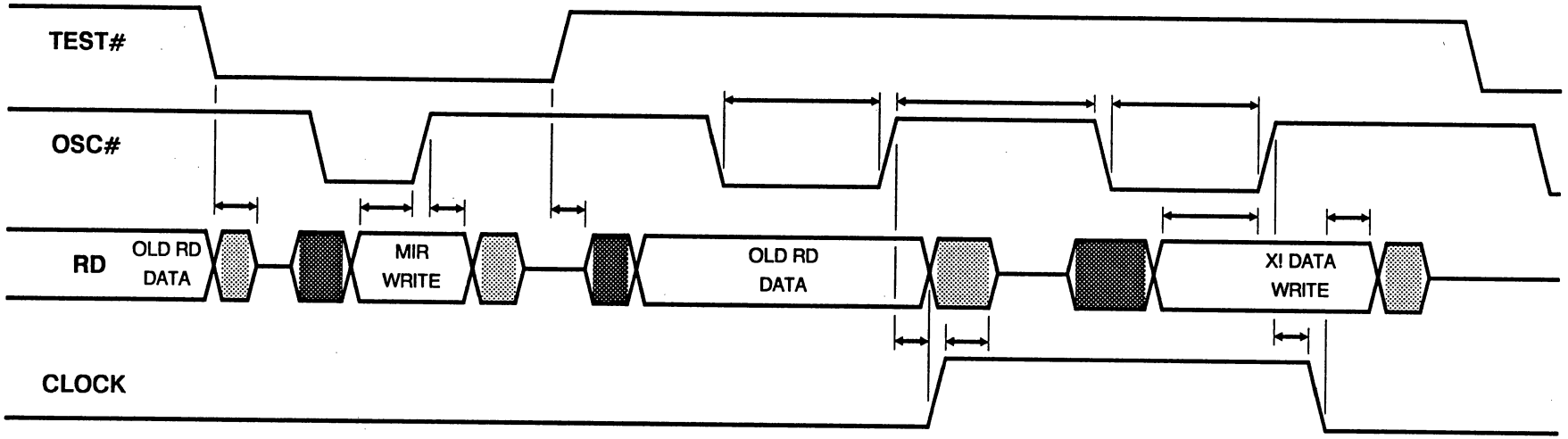
4.3.3 X@ OPERATIONS

The basic single-step operation is the X@ cycle just discussed. Whether or not a single stepped microinstruction needs the bus returned to X@ or not, the same sequence of operations is performed. The only added criterion for a proper X@ operation is that a DEST=HOST must be used in the microinstruction in order to assert the desired value onto the RD bus. Note that the Microassembler uses a default bus destination assignment of DEST=HOST for single-stepped microcode. That means that by default, the Data Bus value is driven directly to the RD pins.

4.3.4 X! OPERATIONS

The X! Microassembler operation is used to store values onto the BINAR as shown in Figure 4-6, X! Timing. This is done by using a bus source assignment of SOURCE=HOST for microcode. That means that the Data Bus value is read directly from the RD pins whenever the MIR has a SOURCE=HOST micro-operation loaded. The RD pins should be driven with the desired input value during the OSC pulse, and must meet setup and hold times around the rising edge of OSC. The falling edge of CLOCK captures the written data in a transparent latch internal to the chip to maintain the MIR! value to the bus mux through the next CLOCK rising edge.

Figure 4-6. X! Timing.



4.4 MICROCONTROLLER OPERATION

Figure 4-7, Microcontroller Execution Path, shows the paths used to run the BINAR microcontroller. The microcontroller consists of the MIR, which holds the current microinstruction; the MPC/JMP/CCREG logic, which generates the address for the next microinstruction; and the MRAM/MROM memories, which form the memory for the microcontroller. The details of the MRAM/MROM addressing and selection logic are hidden in this picture for clarity. Unfortunately, it is not practical to make microcode examples within this section interactive -- don't try to type them into the simulator, or you will get confusing results.

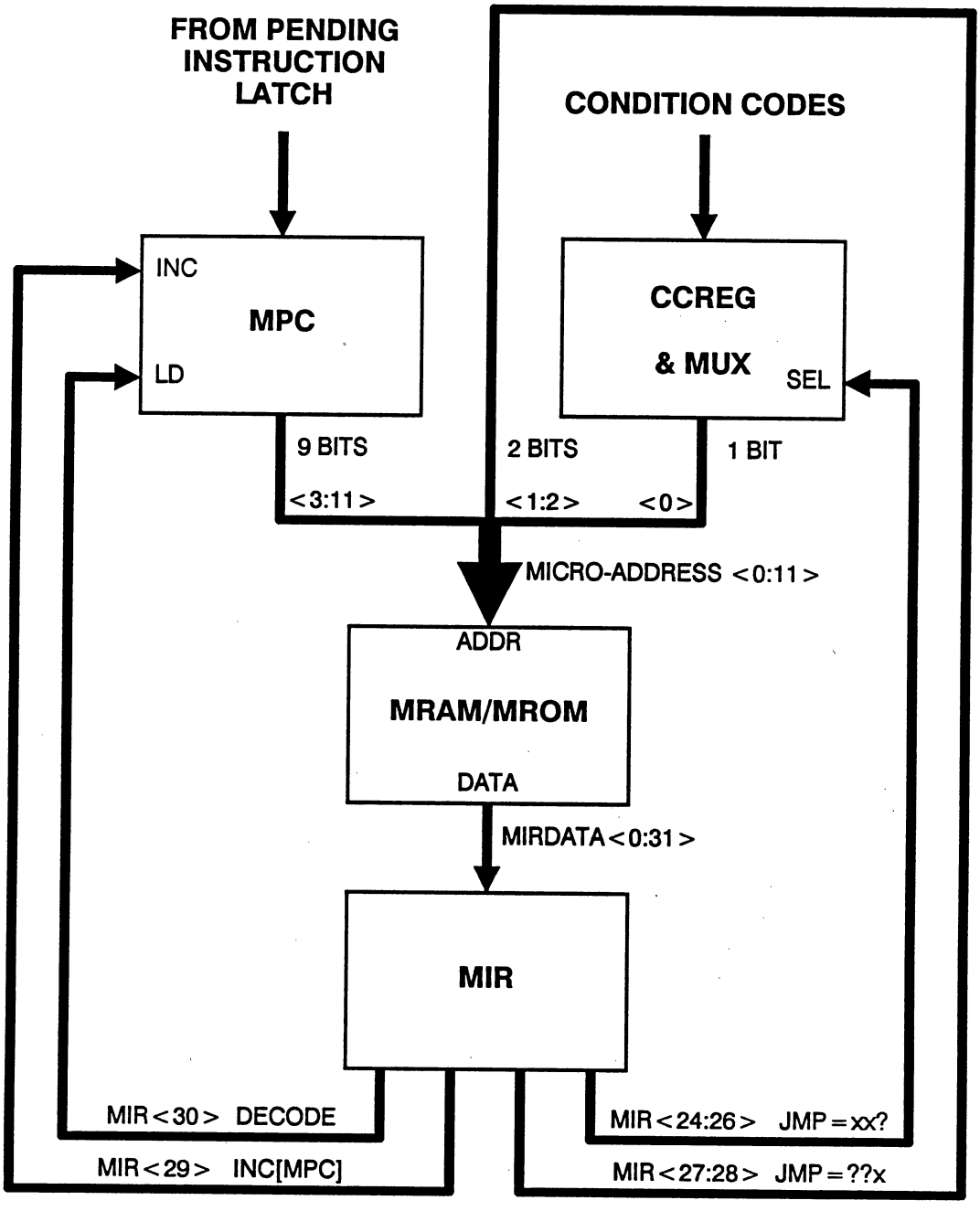
4.4.1 MICROADDRESS OFFSET CONTROL

The microcontroller executes a new MIR value every clock cycle, synchronized to the rising edge of MIRCLOCK. Once a new value is loaded into the MIR, the JMP=xxx control bits of the current microinstruction begin addressing the next microinstruction. For the time being, let's assume that the MPC has an unchanging value that addresses the microprogram memory page of interest.

Bits 24-26 of the MIR form a 3-bit select field in the Condition Code Multiplexer, generating bit 0 of the new micro address. This bit is simply the value of one of the 8 condition codes selected. For example, if MIR<24:26> has a value of 1 (from micro-operation JMP=xxC,) the C bit is selected, and MICROADDRESS<0> will be one if the last clock cycle generated an ALU carry out. Remember that the CCREG is clocked at the end of each clock cycle, so the condition code is valid on the cycle after the ALU function is performed.

Bits 27-28 of the MIR directly form bits 1-2 of the next microinstruction address. That is to say MIR<27> is directly wired to MICROADDRESS<1>, and MIR<28> is directly wired to MICROADDRESS<2>. MIR<27:28> are set using the JMP=??x micro operations (i.e. JMP=00x, JMP=01x, JMP=10x, JMP=11x.) These bits, along with the selected output of the CCREG mux, form the 3-bit offset address within a microcode page. These 3 bits are completely controlled using JMP=xxx micro-operations. The "xxx" may be thought of as the binary value of the offset within the microcode page for fetching the next instruction. The only trick is that the lowest bit may be a conditional branch specification instead of a constant. Here are some examples, with notation as to the values of MICROADDRESS<0:2>:

Figure 4-7. Microcontroller Execution Path.



```
<< JMP=000 >> ( Value is 0)
<< JMP=001 >> ( Value is 1)
<< JMP=110 >> ( Value is 6)
<< JMP=00C >> ( Value is 0 if carry-out is clear,
                         1 otherwise)
<< JMP=00Z >> ( Value is 0 if zero bit is clear,
                         1 otherwise)
<< JMP=10C >> ( Value is 4 if carry-out is clear,
                         5 otherwise)
```

To see an example of how these are used in writing microprograms, we must first introduce some notation. Up until now, we have used the "<<" symbol to start a microinstruction. Actually, the "<<" is a special case that specifies a microinstruction which is to be single-stepped. If microinstructions are written for actual use in executing programs, we need to specify an offset within the microprogram memory page (let's not worry about which page for the moment.) This is accomplished using a number prepended to a : instead of << . Thus, the microinstruction:

```
3: SOURCE=DS  ALU=A+B ;;
```

would be assembled into offset 3 of the currently active microprogram memory page. Valid offset values are from 0 to 7.

Now, it's time for an example of how to control MICROADDRESS bits 0-2. Consider the following microcode that would be resident in a single page:

```
0:  ALU=A+1  JMP=001 ;;
1:  ALU=A+1  JMP=010 ;;
2:  ALU=A+1  JMP=101 ;;
5:  ALU=A+1  JMP=011 ;;
3:  ..............
```

This microcode sequence adds 4 to the number in DHI[0], then continues on at offset 3 with some other unspecified operations. The sequence of control for executing the microinstructions is specified by the JMP micro-operations. Offset 0 contains a JMP=001, which means that offset 1 is the microinstruction executed next. Offset 1 contains a JMP=010, which points to offset 2 as the next microinstruction (remember, the JMP specification is in binary.) Offset 2 points to offset 5, and offset 5 points to offset 3.

It is important to realize that each microinstruction points to the next microinstruction within the microprogram memory page. The order of execution of microinstructions is completely arbitrary (but, to facilitate writing microcode, offset 0 is by convention the first microinstruction to be executed within an opcode.)

Let's see how conditional branching works:

```
0: SOURCE=DS ALU=A-B INC[DP] JMP=001 ;;
1: JMP=10Z ;;
4: ( non-zero result) ALU=0 JMP=110 ;;
5: ( zero result) ALU=-1 JMP=110 ;;
6:  ................
```

This example subtracts the top DS element from DHI and sees whether the result is zero. If the values were equal, DHI[0] is forced to -1, otherwise it is forced to zero. Then, execution continues on at offset 6. This example is one way to implement the Forth = function.

Note first of all that the zero test was done at offset 1. This is the clock cycle after the ALU function being tested was performed. All of the ALU conditional branches worked this way -- they use a one-cycle delayed branch. The actual conditional branch in offset 1 is a branch to either offset 4 or 5, depending on the value of Z. If the Z bit is clear (i.e. the ALU result was not zero), then offset 4 is executed. If the Z bit is set (i.e. the ALU result was zero), then offset 5 is executed. Both offsets 4 and 5 jump to offset 6 to resume execution.

Since the order of execution of offsets is arbitrary, microcoded loops can be implemented. The following example loops while the value in DHI is non-zero.

```
0: ALU=A-1 JMP=010 ;;
2: JMP=00Z ;;
1: ( done with loop) ................
```

This microcode jumps back and forth between offsets 0 and 2 until the DHI value reaches zero, then it jumps to offset 1. If the initial value of DHI were 100, offset 0 would be execute 100 times, and the final value in DHI would be 0.

Let's try a twist on the count-down loop:

```
0: ALU=A-1 JMP=00Z ;;
```

```
1: ( done with loop) ..............
```

This microcode simply loops on offset 0 until the DHI value reaches zero, then it jumps to offset 1. However, since the Z bit is always 1 clock cycle behind the ALU operation. Therefore, if the initial value of DHI were 400, offset 0 would be execute 401 times, and the final value in DHI would be -1.

Taking into account all the effects of the "delayed branching" used by the microcontroller requires some practice. We recommend that you examine the microcode listings for the ROM'ed instruction set on the BINAR to gain more insight.

4.4.2 MICROADDRESS PAGE CONTROL

Up until now, we have assumed that the MPC contains a valid, unchanging value for generating the high 9 bits of MICROADDRESS. But, how does that value get there? Well, the MPC value exactly corresponds to the value of the opcode being executed by the machine. That is, if the current instruction being executed has an opcode field (instruction bits <23:31>) value of 123 decimal, the MPC contains the value 123, and page number 123 of microprogram memory contains the microcode to execute that opcode.

Let's assume that the MPC contains the value of the currently executing instruction. This value will not be changed while the microcode within the page is executing. The question is, what do we do when we are done with that opcode? Obviously, we want to begin execution of the next opcode. This is accomplished by loading the MPC with the value in the IL. We'll assume for now that the IL always has the value for the next instruction to be executed.

Loading the MPC is done using the DECODE micro-operation. Whenever a DECODE micro-operation is used, MPC is loaded with the value of the IL at the beginning of the microcycle. This means that any microinstruction that contains the DECODE micro-operation will be the last microinstruction executed before jumping to a new page of microprogram memory. The following sequence of microcode adds 3 to the value in DHI, then jumps to the next microinstruction:

```
0: ALU=A+1 JMP=001 ;;
1: ALU=A+1 JMP=010 ;;
2: ALU=A+1 DECODE ;;
```

At the end of an opcode's execution, DECODE forces a jump to the microprogram memory page holding the microcode for the next opcode. However, this only affects the MPC value, and leaves the bottom three bits of MICROADDRESS unspecified. For this reason, the microassembler automatically compiles up a JMP=000 micro-operation whenever the DECODE micro-operation is used. This forces a jump to offset 0 on whichever page holds the microcode for the next instruction. This creates a microcoding convention that microcode for all instructions begins at offset 0 of a microprogram memory page.

The DECODE micro-operation may be used in as many microinstructions within a page as desired (but of course only one of them will be executed for each time the opcode is used.) This is well illustrated by a revised version of the Forth = microcode:

```
0: SOURCE=DS ALU=A-B INC[DP] JMP=001 ;;
1: JMP=01Z ;;
2: ( non-zero result) ALU=0 DECODE ;;
3: ( zero result) ALU=-1 DECODE ;;
```

Here, we realize that the instruction is finished at either offset 2 or 3, and just continue on with the next opcode.

Another way to update the MPC is by incrementing it. This technique is used when an opcode takes more than 8 microinstruction words so that the microcode may spill across page boundaries. The MPC is incremented at the beginning of the clock cycle, so any microinstruction with an INC[MPC] micro-operation will be the last microinstruction executed on the page. Like DECODE, INC[MPC] only affects the high 9 bits of MICROADDRESS. Unlike DECODE, however, INC[MPC] does not specify a JMP micro-operation. The following example shows microcode that executes at offsets 3, 4 and 5 of one microprogram memory page, then spills over to start executing at offset 3 of the next page:

```
( This microcode is in page 123):
    ..................................... JMP=011 ;;
3:  ALU=A-1     JMP=100 ;;
4:  ALU=A-1     JMP=101 ;;
5:  ALU=A-1     INC[MPC] JMP=011 ;;
( This microcode is in page 124):
3:  ALU=A+1     JMP=100 ;;
4:   ..............................
```

The microinstructions on the second page may be shared by two opcodes. Offsets 0, 1 and 2 on page 124 might be used by a simple opcode, while offsets 3-7 are used for a continuation of the opcode started on page 123. The INC[MPC] may be used to access as many sequential pages as desired.

There is no way to decrement the MPC. However, execution of large loops may be accomplished by having the high level instruction do a
subroutine call to itself, an having the microcode pop the return stack to loop, or perform a microcode controlled subroutine exit to leave the loop. The details of implementing this "trick" are beyond the scope of this document.

4.5 EXECUTING A MACROINSTRUCTION

We now know how the microcontroller executes the microcode for a single instruction. We also know how the DECODE micro-operation loads the MPC value with the next instruction opcode that resides in IL. The next obvious question is, how does the IL value get loaded? This question leads us into a discussion of the most difficult to understand areas of the BINAR: instruction fetching and decoding.

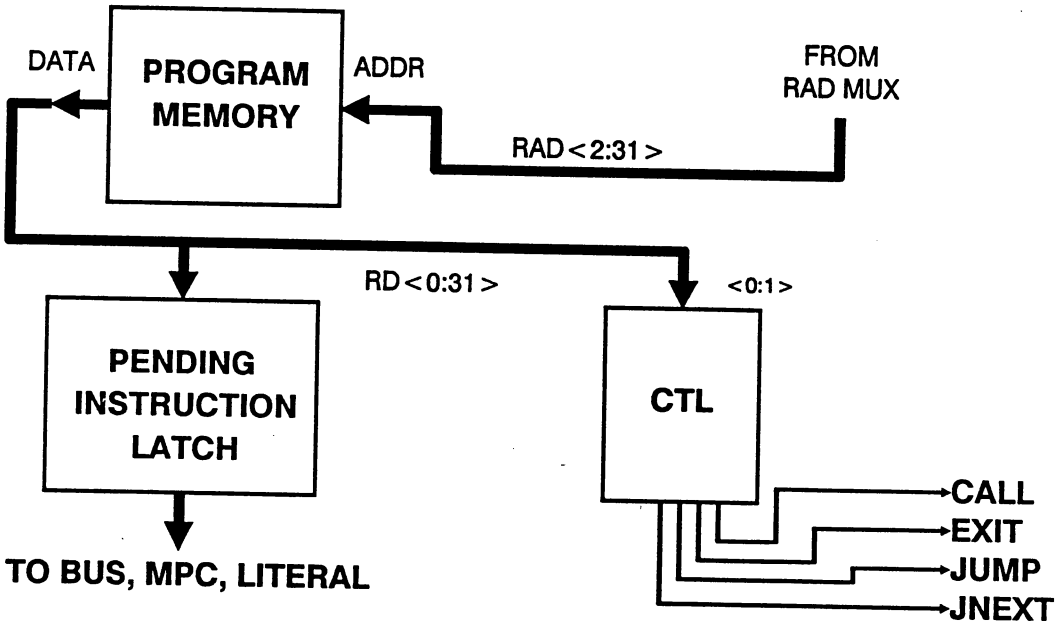
Instruction fetching involves a very short pipeline to "prefetch" the next instruction while the current instruction is executing. It is very simple from a hardware point of view, but there are various subtleties that make the "big picture" somewhat difficult to grasp. The way to understand this section is to re-read it several times during the course of a few days and work several hypothetical examples on paper, tracing the values of the registers involved during the execution of several instructions.

We will build an understanding of the instruction fetching and decoding logic by first examining the control of the IL, then the PAGE/NAR pair, then the Return Stack. For the purposes of the following discussions, we'll assume that single-cycle program memory is in use.

4.5.1 CONTROL OF THE INSTRUCTION LATCH

The Instruction Latch/Control Register (IL/CTL) pair holds the next instruction to be executed. The IL/CTL pair is loaded from the RAM Data Bus (RD) as shown in Figure 4-8,

Figure 4-8. Program Memory/Instruction Latch Path.



Program Memory/Instruction Latch Path. The 32-bit Instruction Latch is loaded from bits 0-31 of the RD bus, and the CTL register is loaded from bits 0-1. The Instruction Latch provides the opcode and literal information for the instruction, while the CTL register provides the instruction type information.

Bits 23-31 of the IL go directly to the MPC as discussed previously. The output of CTL goes through a 1-of-4 selector to activate one of the four signals: CALL, EXIT, 2OPS, and JNEXT for program flow control.

The IL/CTL register is always expected to have the next instruction ready whenever a DECODE micro-operation is executed, so we must ensure that the IL/CTL register always has an appropriate value. There are two ways to load the IL/CTL register. The first way is by executing a DECODE micro-operation. DECODE loads the current IL value into the MPC at the beginning of the clock cycle. Additionally, DECODE initiates a program memory read that gets a new value to be loaded into IL. When the new value is ready from memory, IL and CTL are loaded. How the address is generated for fetching the new value is covered later.

It may be the case that a DECODE micro-operation is executed on the clock cycle immediately following a loading of the IL/CTL register. This causes a problem since the first DECODE is providing information at the end of its clock cycle that is needed immediately at the beginning of the next clock cycle (which both correspond to the same rising edge of CLOCK.) This problem is solved by making IL a transparent latch instead of a register. Information loaded into IL/CTL near the end of a clock cycle is immediately available at the beginning of the next cycle.

If the only way of loading IL/CTL were with DECODE micro-operations, then operation of the IL/MPC pipeline would be simple. Each DECODE micro-operation would load the pending opcode from IL into MPC, then initiates a fetch of the next opcode from program memory into IL.

There is a complication, however. Performing conditional branches and some other operations requires re-loading the IL/CTL value with a new instruction, one not fetched by a DECODE. This is done using the LATCH-INSTRUCTION micro-operation discussed elsewhere. The LATCH-INSTRUCTION micro-operation loads the IL/CTL register from a value being fetched from the RAM bus just as if it were an instruction being fetched by DECODE. However, LATCH-INSTRUCTION does

not update the value in MPC. Thus, it may be thought of as a means for changing the pending opcode without affecting the currently executing opcode. When LATCH-INSTRUCTION is executed, the IL/CTL contents are over-written with a new value from RD, and it is as if this new value was the one that was actually fetched into IL/CTL by the previous DECODE.

Remember, IL/CTL acts as a pending instruction location, while MPC acts as the current instruction location.

4.5.2 CONTROL OF PAGE/NAR

Now, let's look at how the addresses are generated to program memory for the instruction fetching. Figure 4-9, PAGE/NAR Instruction Path, shows the previous IL/CTL Figure 4-8 updated to include the PAGE/NAR register pair used to generated program memory addresses. This discussion does not completely describe the operation for the two/word class but is accurate for other classes, see "BINAR Architectural and State Logic Requirements Imposed by the Two/Word Instruction Class". The PAGE/NAR register pair together form a 30-bit word-aligned address for fetching instructions. The bottom two bits of the address are wired to 0, since instructions are 32-bit values which must be word-aligned.

During every DECODE cycle, program memory is read to fetch the next instruction. What happens is that the RAM Address Mux (RAD MUX) passes a word-aligned address from PAGE/NAR to the RAD, which is the program memory address bus. The output of program memory is then read into the IL/CTL register as previously described. Now the machine has to update the PAGE/NAR register to point to the next instruction beyond the one just fetched.

In the following two paragraphs, the description of the use of the CTL field isn't quite accurate. It has been simplified for the moment. We'll explain the problem shortly.

If the CTL register holds the value 3, a Jump-To-Next (JNEXT) instruction is executed. This means that the PAGE/NAR registers are to act as a program counter. This is accomplished by loading the PAGE/NAR registers with the output of the incrementer attached to RAD<2:31>. This box adds 1 to RAD bits 2-31, which is the same as adding 4 to RAD<0:31>. The PAGE/NAR is also loaded with the incrementer output in the case of a subroutine EXIT function (more on that later.) Operation for a CTL register value of 2 (2OPS)

is similar in this respect, but with complications in the MPC manipulation.

If the CTL register holds 0, a subroutine call is being processed. In this case, the NAR is loaded with bits 2-22 of the instruction being read from memory. These bits correspond to the address field of the instruction for Jumps and subroutine CALLs. The new PAGE register value is not contained in the instruction since there is only a 21 bit address field in the instruction format. The PAGE register is loaded from the incrementer output. A Jump-To-Next instruction located in the very last word of an 8-MB page will jump to offset 0 of the next page.

Now for the details of what's really going on with the CTL register. If you examine what happens to PAGE/NAR more closely, you will realize that there is a timing problem here. In order to know which way to route data through the mux on the NAR input, the CTL register must be loaded and decoded to produce CALL CALL/EXIT/2OPS/JNEXT. Note that it is the instruction being fetched that must determine the address of its own successor, which is the address loaded into PAGE/NAR. Therefore, whether to load the NAR with RAD+4 or the address field of the instruction being fetched depends on the CTL bits of the instruction being fetched, not the bits that were in CTL before the instruction fetching began. There would be a long path from program memory through the CTL register and its decoder to the NAR mux control if we did a straightforward implementation. However, there is a simpler way. The NAR input mux is simply controlled by RD<1>. If RD<1> is 0, then the incoming instruction is a CALL or EXIT and the NAR should be loaded from RD<2:22> (the value loaded into NAR for an EXIT is irrelevant, since it won't be used). Otherwise, NAR is loaded with the incrementer output. This scheme eliminates tricky timing problems with CTL and greatly reduces RAM chip speed requirements.

So, the final result of all this is that CALL instructions load their address fields into NAR as they are fetched. CALLs just load NAR with a value that points to the next instruction. JNEXT and 2OPS instructions have no address field, so whenever a JNEXT or 2OPS instruction is loaded, PAGE/NAR is loaded with the output of the RAD incrementer.

We're now juggling information on three instructions within the system. Between instruction fetches the MPC contains the currently executing opcode. IL contains the opcode of the pending instruction (the one to be executed next.)

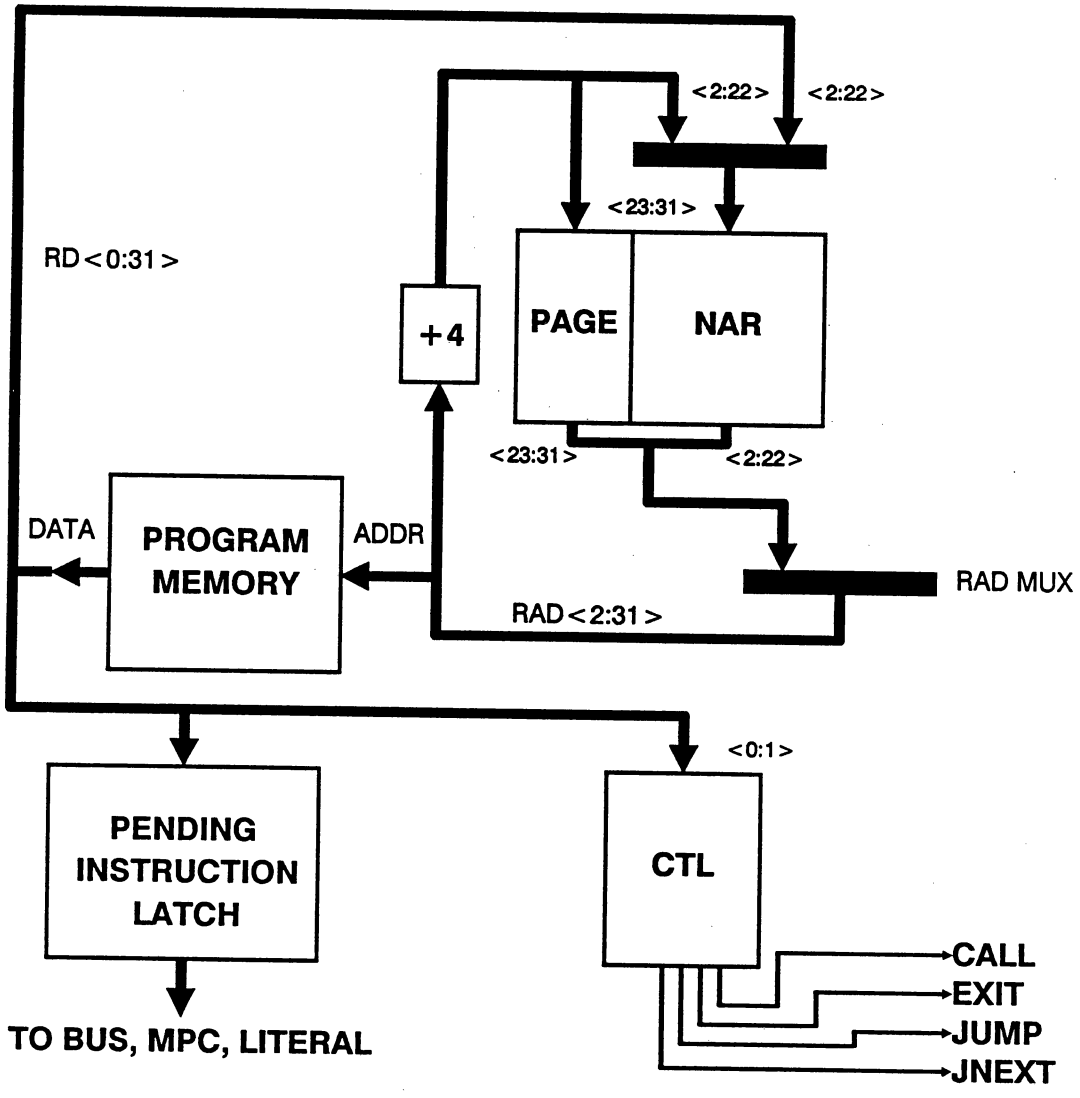
PAGE/NAR contains an address determined by the pending instruction which points to the instruction to be fetched next, beyond the pending instruction. Note that there is a big difference between this pipeline and the prefetch queues used by other processors: this pipeline automatically tracks subroutine CALLs and Jumps, so there are no pipeline breaks or flushes required.

The operation of the literal register, while not explicitly shown in Figure 4-9, PAGE/NAR Instruction Path, is tied to instruction decoding. Whenever an instruction is loaded, the literal fields, if any, are loaded into the Pending Instruction Latch with the rest of the pending instruction. At the end of the DECODE clock cycle, the SHORT LIT register is updated with the contents of the Pending Instruction Latch. This delay parallels the arrangement of the IL/MPC pipeline to separate the pending instruction from the currently executing instruction.

The LATCH-INSTRUCTION micro-operation loads PAGE/NAR and the Pending Instruction Latch exactly the same way that DECODE does, using the value coming in on the RD-REG as an instruction being fetched.

The operation of 2OPS instructions is very similar to the operation of JNEXT instructions. When a 2OPS instruction is transferred from the Pending Instruction Latch into execution (by executing the DECODE micro-operation of the previous opcode), both the OPA and OPB fields of the instruction are loaded into MPCA and MPCB (see Figure 3-7, Micro-Sequencer Block), respectively (previous references to MPC were implicitly to MPCA). Also, the two literal fields of the instruction are loaded into the LITA and LITB registers (see Figure 3-6, RAM Data Alignment Block]) as well. As the first opcode is executed, MPCA and LITA are selected for use, exactly as in the JNEXT case. This gives the logic at least one clock cycle to determine that a 2OPS instruction is in progress instead of a JNEXT instruction. When the DECODE micro-operation of the first 2OPS opcode is executed, instruction fetching is suppressed; instead, muxes simply shift from the MPCA/LITA select positions to the MPCB/LITB positions. When the next DECODE micro-operation is performed (from the OPB opcode), then instruction fetching is performed as if it were the DECODE from the only opcode in a JNEXT instruction. The suppression of the OPA DECODE micro-operation in terms of instruction fetches means that both the OPA and OPB opcodes may be only a single clock cycle long if desired within a 2OPS instruction without upsetting the two-clock-per-bus-cycle operation of the BINAR.

Figure 4-9. PAGE/NAR Instruction Path.



4.5.3 CONTROL OF THE RETURN STACK

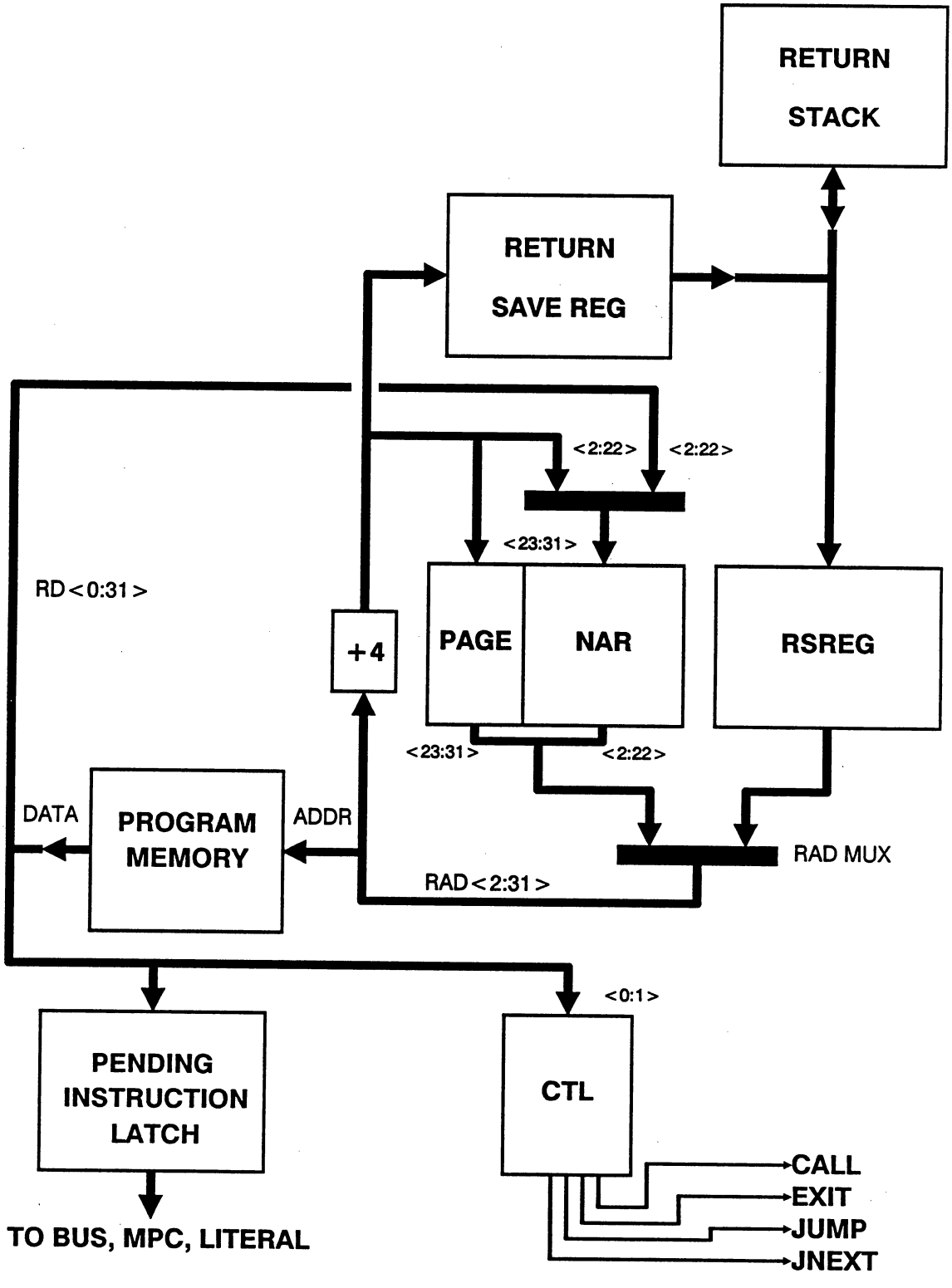
There is one aspect of instruction fetching that we have not discussed yet: return stack operation for subroutine CALLs and EXITs. Figure 4-10, Simplified Instruction Path, shows the "complete picture" for instruction fetching, including the Return Stack. There are two separate considerations for controlling the return stack: popping elements for subroutine EXITs, and pushing elements for subroutine calls.

Processing subroutine EXITs is straightforward. Whenever DECODE is used to fetch a new instruction from memory, the RAD MUX uses the value in the CTL register to select its input. If the EXIT signal is asserted, it uses the RSREG to address Program Memory. Otherwise, it uses PAGE/NAR as previously described. Since the return address is always at the top of the RS (and therefore in RSREG), subroutine EXITs work properly. As an additional consideration, an EXIT coupled with a DECODE cause the RP to be incremented, popping the RS element to allow multiple consecutive EXIT operations. Loading the PAGE register with the contents of RAD+4 for subroutine EXITs is the correct action, since this automatically restores the PAGE register when doing an return from an inter-page subroutine call. (The call itself must be handled by a special opcode, but the return is handled automatically.)

Processing subroutine calls is a little bit trickier. Getting the address to call to is simple: the PAGE/NAR registers tell you where to fetch the next instruction. Now what is needed is to push the return address onto the RS. Pushing an arbitrary value onto RS is also simple. The RP is decremented at the beginning of a clock cycle when a DECODE micro-operation with a CTL value of CALL is encountered, then the RS is written with the return address. The hard part is generating the return address.

The approach used to generate return addresses is to in a sense assume that every instruction is a subroutine call. The return address is then computed for every subroutine instruction. The return address is simply the output of the RAD incrementer, since it generates the address of the currently being fetched instruction plus 4. The problem is that it is too soon to push an element onto the RS, since the instruction might not have been a CALL after all. To solve this problem, the RETURN SAVE Register is used. Every instruction fetch updates RETURN SAVE with a potential subroutine return address. If the instruction that was

Figure 4-10. Simplified Instruction Path.



fetched happens to be a subroutine call, the return address is pushed onto RS when the instruction fetches its successor (i.e. fetches the first instruction of the subroutine.) If not, the RETURN SAVE value is over-written with the next return address. RETURN SAVE is updated whenever either DECODE or LATCH-INSTRUCTION are used.

Think of RETURN SAVE as the pending instruction's return address. It is used only if the pending instruction happens to be a subroutine call. It is written to RS when the pending instruction fetches the first instruction of the subroutine, which is at the same time as the opcode associate with the subroutine call is being transferred from the IL into the MPC by a DECODE.

4.5.4 MULTI-CLOCK CYCLE MEMORY INSTRUCTION DECODING

We have not really worried about the precise timing of events in the instruction fetching and decoding sequence thus far. The BINAR is constructed assuming that single clock cycle access memory cannot affordably be used. It is, instead, optimized for the case of two-cycle memory.

In the case of two-cycle memory being used for instruction fetching, two different microcycles are used to do the memory access and other manipulations. The first clock cycle is the DECODE microcycle, which is the last microinstruction to be executed of the "current opcode". We'll call this the START-DECODE cycle. During the START-DECODE cycle, all the actions associated with the beginning of a single-cycle decode take place: IL is copied to MPC at the beginning of the clock cycle, and the RS push or pop starts (with RP being incremented or decremented at the end of the clock cycle). Also, the IL is clocked into the user-visible SHORT-LIT register at the end of the START-DECODE cycle so that it can be used by the opcode beginning execution.

On a two-cycle instruction decode, IL/CTL, PAGE/NAR, RS, RSREG, and RETURN SAVE cannot be updated on START-DECODE. This is because they contain critical control information that is being used for a Program Memory access that is only half-completed. These actions must be postponed until the end of the cycle after START-DECODE, which we will call the END-DECODE cycle. The END-DECODE cycle corresponds to the first microinstruction of the next opcode (the one clocked into MPC during START-DECODE). At the end of the END-DECODE microinstruction, IL/CTL, PAGE/NAR, RS, RSREG, and RETURN SAVE are all updated. All these resources see the START-

DECODE/END-DECODE sequence as a sort of double-length clock cycle. In particular, RS/RSREG are not updated at the end of START-DECODE, even though the RP has been changed -- they are only updated by the END-DECODE clock. A microcoding constraint introduced by this is that RP may not be modified during the first clock cycle of any opcode, since this is the END-DECODE cycle for the previously executed DECODE micro-operation (this rule can be excepted when it is guaranteed that the opcode beginning execution is not compiled with a CALL or EXIT instruction).

What about memory that is slower than 2 clock cycles? Memory control logic could stall the processor using the WAIT pin (during the END-DECODE phase) so that only 2 microinstructions are executed while memory is being accessed.

4.5.5 INSTRUCTION TRAPS

What happens if an illegal opcode (i.e. an opcode for which there is no corresponding MROM or MRAM memory available)? At first thought, one would be tempted to generate an interrupt or some other fancy control mechanism to handle this case. The BINAR takes a much simpler approach.

The addressing logic for the MROM and MRAM memory must select which of the two memories to address based on the MPC value in use. A small addition to this logic forces opcode 3 to be addressed from MROM whenever the address to the microcode memories falls into the gap between MROM and MRAM. This, in turn, causes opcode 3 (the TRAP opcode) to be executed whenever an illegal opcode is encountered at run time. Opcode 3 uses microcode to generate a jump to memory location $FFFFFFFC, behaving in a similar manner as an interrupt, but without generating a pulse on the INTA pin.

4.5.6 SUMMARY OF INSTRUCTION DECODING ACTIONS

The following is a summary of actions that take place on instruction decoding.

START-DECODE:

Actions at CLOCK rising edge when PENDING-DECODE asserted (i.e. actions at the starting CLOCK rising edge of the START-DECODE cycle):

> Sample interrupt request lines (clock into a D-REG)
> Clock MPCs with Pending Instruction Latch values

Initiate a bus cycle (default of read-RAM-word cycle)

Actions during the microcycle when START-DECODE is asserted (i.e. asynchronous actions during the microcycle):

```
IF interrupt
THEN
     set MPC to opcode 1
     set CTL to a CALL instruction
     change bus cycle to an INTA bus cycle
     IF        NMI
               THEN set PAGE/NAR for address $FFFFFFF8.
               ELSE (INTR)
                    set PAGE/NAR for address $FFFFFFF4.
               ENDIF
ENDIF
Configure RAD-MUX to assert RSREG if EXIT,
                                   else PAGE/NAR
(this configuration holds through END-DECODE)
```

Actions at the end of START-DECODE cycle (at rising CLOCK):

```
IF   interrupt
THEN (simulate hardware subroutine call, return
     to address of squashed instruction)
     (This saves restart address+4 on RS)
     (DEC[RP] -- performed by microcode at 0: of
     <INTERRUPT>)
ELSE
     Load SHORT-LIT register from
                         Pending Instruction Latch
     IF        CALL      THEN       DEC[RP]  ENDIF
     IF        EXIT      THEN       INC[RP]  ENDIF
(Note: this permits an INC/DEC of RP by microcode
     during the DECODE microinstruction, but forbids
     it during the first microinstruction of a new
     opcode unless the instruction having that opcode
     is guaranteed not to be a CALL or EXIT)
ENDIF
```

END-DECODE:

Actions during microcycle when END-DECODE is asserted (i.e., asynchronous actions during the microcycle):

```
IF CALL OR Interrupt
THEN enable write of RETURN-SAVE to RS-RAM
     (during second half of clock cycle)
ENDIF
```

Actions at the end of END-DECODE cycle (at rising CLOCK):

```
Clock RETURN-SAVE
Clock RDREG
Clock RS-REG

Latch Pending Instruction Latch

IF RD<1> = 0 (i.e. if an incoming CALL or EXIT) THEN
     clock NAR from RD bits
ELSE clock PAGE/NAR from RAD+4
ENDIF

Configure RAD-MUX to assert ADDR register
```

Actions Taken for a LATCH-INSTRUCTION micro-operation, at rising edge of CLOCK at end of microcycle:

```
Clock RETURN-SAVE
Clock RDREG
Latch Pending Instruction Latch
IF RD<1> = 0 (i.e. if an incoming CALL or EXIT) THEN
     clock NAR from RD bits
ELSE clock PAGE/NAR from RAD+4
ENDIF
```

Actions taken when RESET signal is asserted to perform cold-start:

PAGE/NAR are set so they produce a memory address of $FFFFFFF0. Pending Instruction Register is set to 0. MPC is set to 4.

4.6 MEMORY BUS OPERATION

The operation of the memory bus may be considered at two levels. In this section we shall consider the operation of memory accesses on a microinstruction-by-microinstruction basis. The timing of the operation of the memory interface control pins are discussed in Appendix C.

In general, the BINAR runs too fast to be able to use affordable single-cycle memory chips. This is not a significant problem, however, since the BINAR architecture is balanced to make best use of two-cycle memory, since this will be the fastest memory that can generally be installed. The commercial version of the processor will have on-chip wait state generation logic to allow the use of a variety of

memory chip speeds. However, the fastest memory that will be supported will be two-clock cycle access time memory.

In order to maximize the use of available memory bandwidth, the BINAR design goes to great lengths to have the address available to memory chips as soon as possible after the start of a memory cycle. In order to do this, the address used for accessing RAM is computed the microcycle before a RAM access is started, and stored in a group of registers that are very close to the RAM Address Pins (namely, PAGE/NAR, ADDR, and RSREG). This significantly decreases the delay from the rising edge of CLOCK on a memory cycle until the RAD bus has a valid RAM address. The values in PAGE/NAR, ADDR, and RSREG are available to the RAD bus in the clock cycle after they are written.

The BINAR also delays the time when data read from RAM must be valid until as late as possible in the clock cycle. This allows a minimum setup time from the RAM chips into the RD pins, allowing the use of the slowest possible RAM chips. To accomplish this, data read from RAM is held in a Ram Data Register (RDREG) that takes its inputs directly from the RDBUS. The RDREG is connected to the system Data Bus to allow RAM data to be routed throughout the system. However, the RDREG does not contain valid data until the clock cycle after the memory cycle is completed (since it is loaded with data at the very end of the last memory access microcycle).

The general principles used to increase memory bandwidth in the BINAR having been discussed, let's cover some specific examples to illustrate the techniques used.

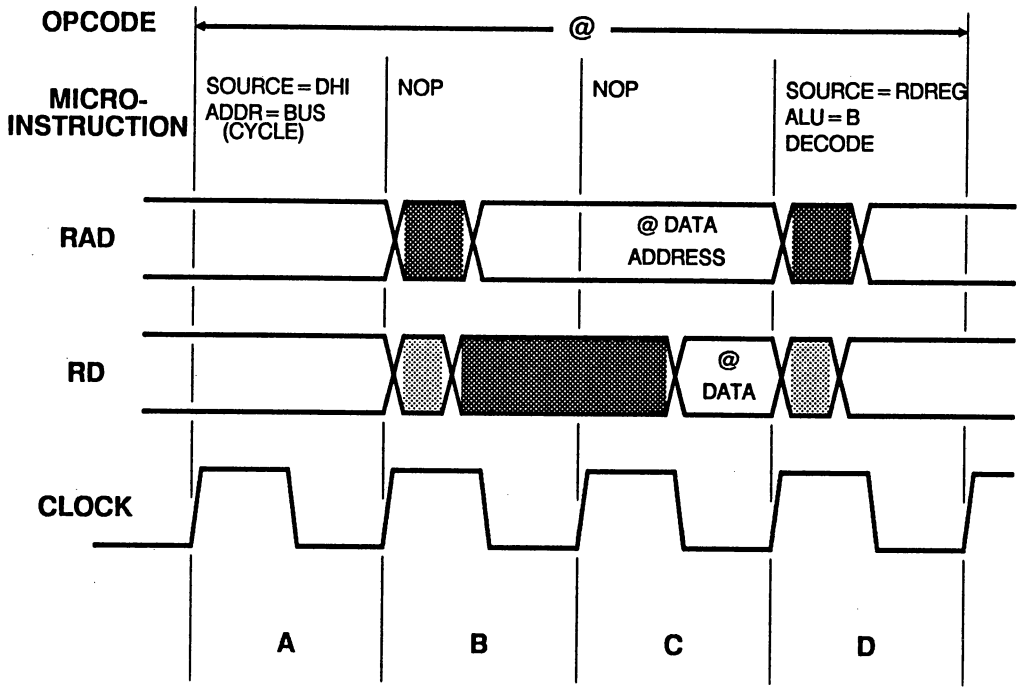
4.6.1 A SIMPLE DATA READ EXAMPLE

Figure 4-11, Data Read Example, shows the sequence used to read a word of data from memory using the Forth @ opcode. All memory bus activity not a part of the actual data fetch is ignored for the moment. The microinstructions shown are those microinstructions that are executing during the indicated clock cycle. The "A" through "D" letters are just arbitrary labels of clock cycles for use in the current discussion.

During clock cycle A, the ADDR register is written with the value on the top of the user's data stack (which is in the DHI register). This value actually becomes valid in the ADDR at the very beginning of clock cycle B.

Since clock cycle A specifies a RAM operation is to be performed (because of the "(CYCLE)" portion of the

Figure 4-11. Data Read Example.



destination micro-operation), a RAM cycle is started at the very beginning of cycle B. As quickly as possible, the contents of ADDR are driven onto the RAD bus. Simultaneously, the RD bus is turned around by first turning off any previous device asserted on the bus, then selecting the memory device addressed by the current RAD value. By the end of cycle B, the address and RAM control pins are stable, and the processor is waiting for the RAM chips to respond with valid data.

Since this is a two-cycle memory example, cycle C is needed to wait for the RAM chips to return valid data. By the end of cycle C, the RD bus has the data read from RAM. Note that the data need only be valid a short setup time before the beginning of clock cycle D.

At the very start of clock cycle D, the data is captured from the RD bus into the RDREG. Simultaneously, the RAD bus is turned over to the next memory bus operation (if any) and the RAM devices used by the data read are turned off, relinquishing the RD bus.

In this example, @ requires four clock cycles to perform its function. However, only two clock cycles of actual memory bandwidth are consumed. The other two clock cycles (cycles A and D) are put to use fetching instructions.

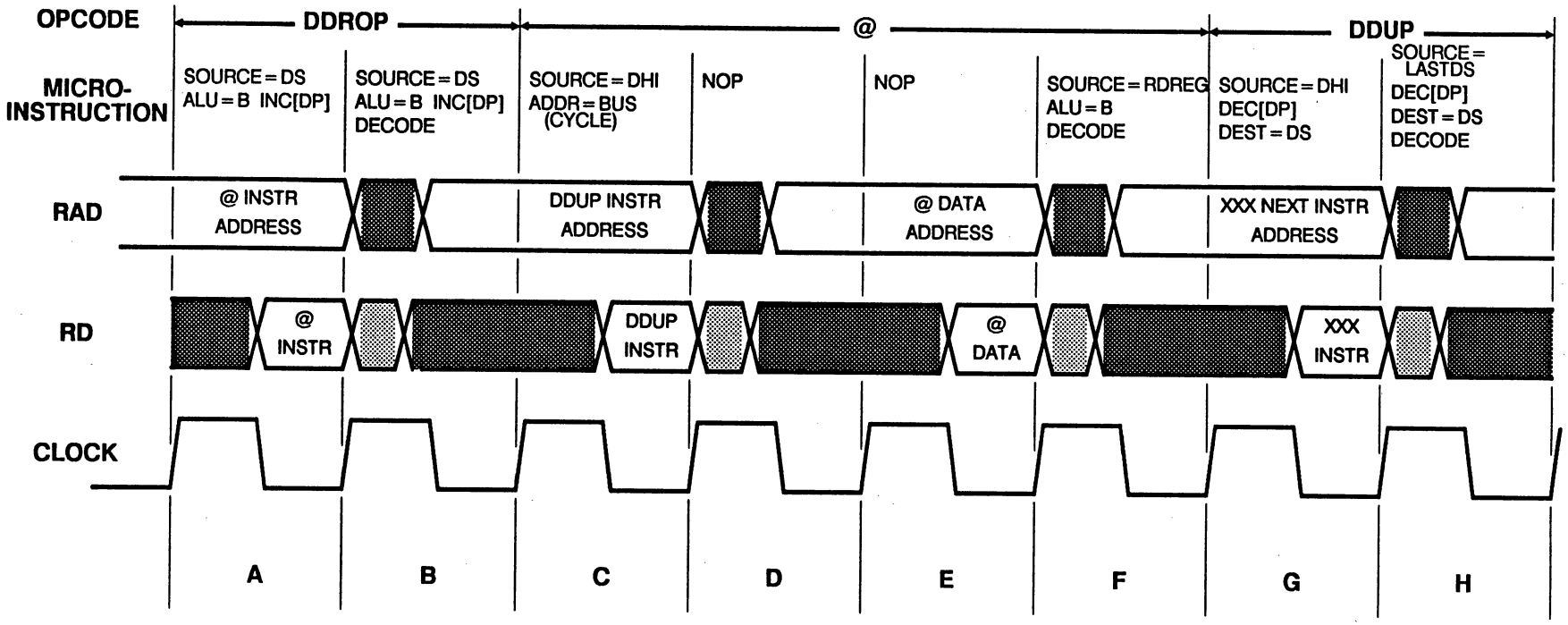
Although not shown in this example, cycle B can be used to override the default RAM word read operation by specifying a micro-operation such as RAM-BYTE-READ, RAM-HALF-READ, etc. Since the RAM cycle is just beginning in cycle B, these micro-operations can provide appropriate information to the I/O control pin logic in plenty of time to accomplish the correct memory transfer operation. The default bus cycle is a RAM word read.

4.6.2 A SEQUENCE OF INSTRUCTIONS INCLUDING A DATA READ

We have seen an example of a data read bus transaction. The other kind of reads that are possible are instruction reads. Instruction reads are identical to data reads as far as the RAD and RD busses are concerned, taking two clock cycles to access memory. The only unusual thing about instruction access is that their two clock cycles straddle an instruction boundary.

Figure 4-12, Data Read Instruction Sequence Example shows the following sequence of instructions as they are executed:

Figure 4-12. Data Read Instruction Sequence Example.



DDROP @ DDUP

During clock cycles A and B, the microcode to accomplish a DDROP instruction is executed. Note that in cycle A, some previous action (namely, the DECODE of the opcode executed before DDROP) has already started a fetch of the @ instruction, and the instruction is available for reading on the RD bus at the end of cycle A. At the end of cycle A, this new instruction is clocked into the pending instruction register (IR and PAGE/NAR) and is ready for execution.

During cycle B, DDROP executes a DECODE micro-operation. This transfers the IL into the MPC, and uses the PAGE/NAR value to initiate the fetch of the next instruction (the one after @, since the @ has already been loaded). Therefore, during cycles B and C, we see the RAD bus supplying the address of the DDUP instruction, and the RD bus returning the DDUP instruction. Also note that during cycle B, the microcode memory is being addressed with the new MPC value (which is the page number of the @ opcode) so as to provide the first microinstruction for @ at the beginning of clock cycle C.

Cycle C executes the microinstruction to start the actual data fetch for the @ opcode. Note that the operation to load the ADDR register (at the end of clock cycle C) is overlapped with the fetching of the DDUP instruction. This means that while @ uses a microcycle to load ADDR, no time is actually wasted, since RAD and RD are busy fetching the DDUP instruction.

During cycles D and E, the @ data is fetched using RAD and RD just as in Figure 4-11, Data Read Example.

During cycle F, the @ opcode executes a DECODE micro-operation, initiating a read of the next instruction past DDUP, which is called XXX for simplicity. The transfer of control between @ and DDUP is accomplished in the same manner as it was between DDROP and @.

The key point of this example is to note that the memory bus is never idle. @ Uses the middle two clock cycles (cycles D and E) to perform its data fetches, and uses its first and last cycles (C and F) for portions of instruction fetch cycles. If @ were a longer opcode that did other operation, the memory bus would go idle after the @ data read operation, and become active with the XXX instruction fetching operation only during the last (i.e. the DECODE micro-operation) cycle of the opcode.

All instruction fetched is initiated during the DECODE clock cycle of an opcode. The instruction that is fetched is always the one to be executed after the pending instruction.

4.6.1.3 A SIMPLE DATA WRITE EXAMPLE

Figure 4-13, Data Write Example, shows the sequence used to write a word of data to memory using the Forth ! opcode. All memory bus activity not a part of the actual data store is ignored for the moment.

During clock cycle A, the ADDR register is written with the value on the top of the user's data stack (which is in the DHI register). This value actually becomes valid in the ADDR at the very beginning of clock cycle B. This is identical to the operation of a read bus cycle.

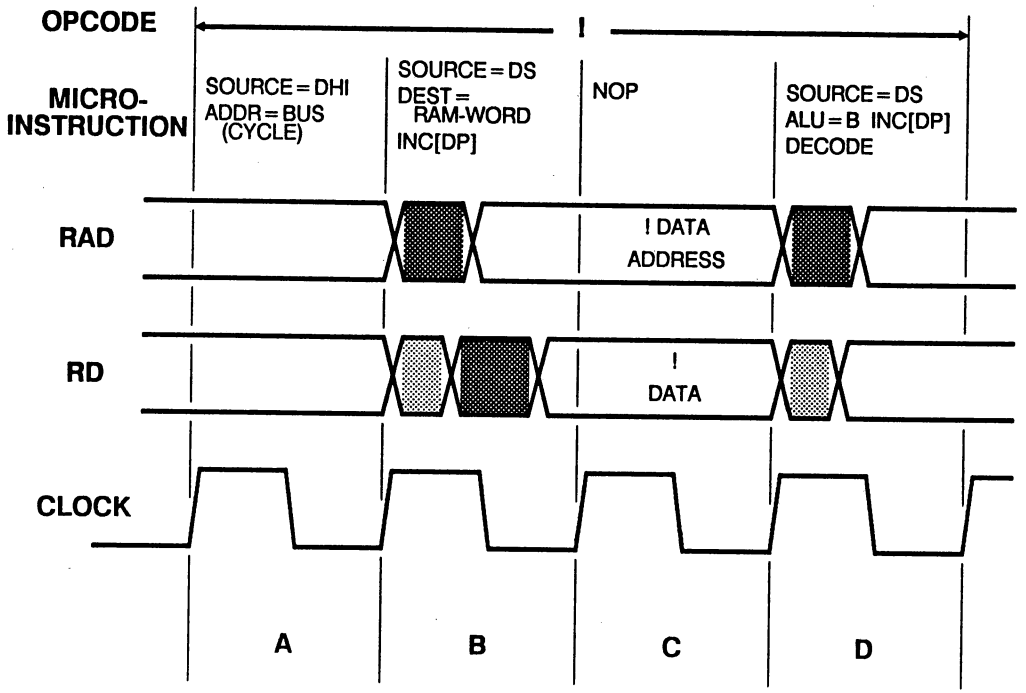
Since clock cycle A specifies a RAM operation is to be performed (because of the "(CYCLE)" portion of the destination micro-operation), a RAM cycle is started at the very beginning of cycle B. As quickly as possible, the contents of ADDR are driven onto the RAD bus. Simultaneously, the RD bus is turned around by first turning off any previous device asserted on the bus, then asserting the data to be written to the memory device addressed by the current RAD value. By the end of cycle B, the data, address and RAM control pins are stable, and the processor is waiting for the RAM chips to capture the data. Since the RD bus must have data to provide to the RAM chips by the end of the B cycle, the microinstruction executed performs a DEST=RAM-WORD micro-operation to provide the data. This data is held in a latch at the RD pins throughout the write cycle, so that other micro-operations could be performed in cycle C if desired.

Since the BINAR uses two-cycle memory, cycle C is needed to wait for the RAM chips to capture the data. By the end of cycle C, the RD bus has written the data into RAM.

At the very start of clock cycle D, the RAD bus is turned over to the next memory bus operation (if any) and the RD pins on the BINAR are turned off, relinquishing the RD bus.

In this example, ! requires four clock cycles to perform its function. However, only two clock cycles of actual memory bandwidth are consumed. The other two clock cycles (cycles A and D) are put to use fetching instructions.

Figure 4-13. Data Write Example.



The selection of the correct RAM writing micro-operation (DEST=RAM-WORD) during cycle B is important. Not only does this supply data to be driven onto the RD bus, but also specifies the type of memory cycle to be executed (word, byte, half-word, read/write).

4.6.1.4 A SEQUENCE OF INSTRUCTIONS INCLUDING A DATA WRITE

Figure 4-14, Data Write Instruction Sequence Example, shows the following sequence of instructions as they are executed:

```
DDROP ! DDUP
```

This example is almost identical to Figure 4-12, Data Read Instruction Sequence Example, with the exception of the memory write signals indicated in cycles D and E. These signals are taken directly from the data write example of Figure 4-13.

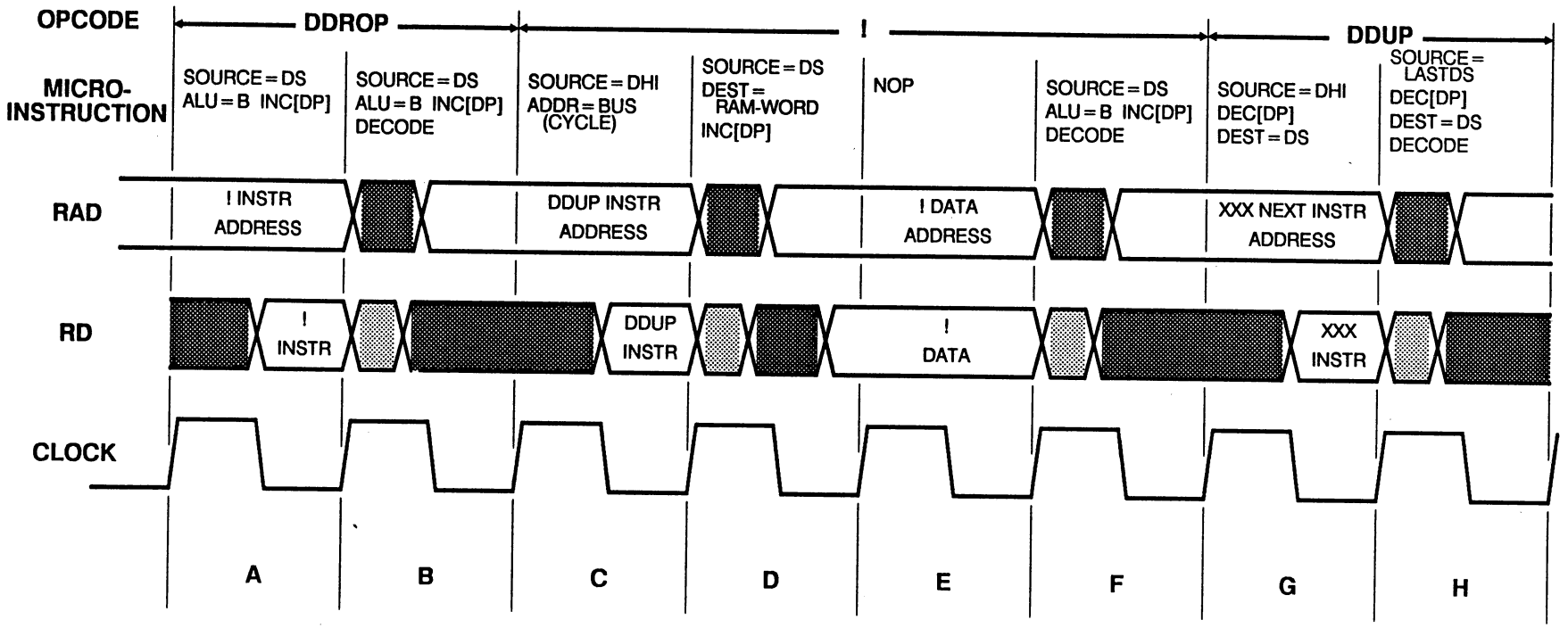
4.7 SYSTEM INITIALIZATION

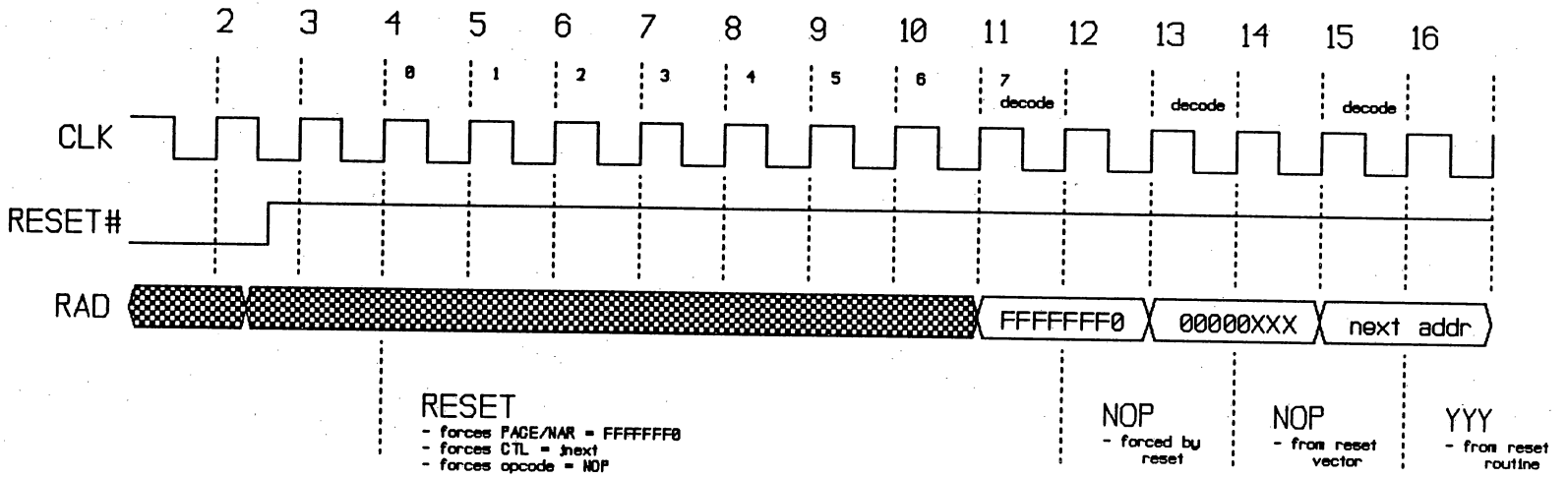
The RESET pin of BINAR is used to perform system initialization. When the RESET pin is asserted (typically by using an RC network with a Schmitt-triggered gate or by a command from the host processor), it must be held active for several (10 or more?) clock cycles while the OSC input is being cycled for proper operation. During the reset operation, all registers in the processor are initialized to zero with the following exceptions:

PAGE is set to 7FF hex, resulting in a page value of FF800000 being added to all addresses.
NAR is set to 007FFFF0, resulting (together with page) in a cold-start execution routine starting execution at address FFFFFFF0.
MPC is set to 4, resulting in a cold-start opcode of 4 being executed.
DP is set to 3E hex, DPTOP is set to 3F hex, and DPBOT is set to 00 hex, initializing the data stack pointer and limit logic.
RP is set to 3E hex, RPTOP is set to 3F hex, and RPBOT is set to 00 hex, initializing the return stack pointer and limit logic.
CONFIG is set to 1, masking interrupts.

All on-chip RAM powers up in an uninitialized state, and must be assumed to contain garbage values until written. Figure 4-15 illustrates resetting the chip.

Figure 4-14. Data Write Instruction Sequence Example.





FFFFFFF0 - CALL XXX

00000XXX - OPCODE YYY

notes: first real clock edge at t= 3
first mir latched from mrom at t= 4
mrom 4:0 executed from t= 4 to t= 5

Figure 4-15. Reset Operations.

4.8 UNUSUAL HARDWARE FEATURES

That's right -- this is the section with known "features" (sometimes called bugs, quirks, and other names) that can cause problems on the BINAR chip.

1) Interrupts.
The interrupt processing opcode does not properly save the contents of the CONFIG register. Therefore it is impossible to tell from the status word captured by the interrupt opcode whether interrupts were masked or not before the interrupt was processed. If an INTR was processed, then obviously they could not have been masked. If an NMI was processed, then it is not easily discernible what the state of the INTR mask bit was before the interrupt. This could be worked around by using a semaphore to notify the NMI service routine about the status of the INTR mask bit. The easiest work around is to assume that an NMI is a catastrophic system event which causes a warm start. There may be other more subtle effects of this problem that have not been investigated. This problem will of course be corrected in the commercial part.

2) +LOOP is broken.
The +LOOP opcode placed in the microcode ROM is broken. The instruction set reference includes correct microcode source. The Forth compiler uses a microcoded RAM opcode to implement +LOOP. This problem will be corrected in the commercial part.

3) Return stack operations with CALLS/EXITS.
a) No return stack operator (R>, DR>, >R, D>R) may be used in the same instruction as a CALL or EXIT. Therefore, the sequence R> CALL must be compiled with the R> in one instruction and a NOP opcode with the CALL in the next instruction. The opcodes that have this restriction are clearly marked in the instruction set reference. This is an inherent characteristic of the design, and can not be changed. The Forth compiler automatically compiles correct code sequences for this case.
b) There is an additional restriction before an EXIT instruction. an R> or DR> must not be used as the opcode immediately preceding the execution of the EXIT instruction. This means that R> or DR> cannot be the second opcode of a 2OPS before the EXIT, nor may it be the opcode of a JNEXT before the EXIT. It may, however, be the first opcode of a 2OPS before the EXIT. It is not clear whether this characteristic of the design can be changed, but it may well

continue into the commercial product. The Forth compiler automatically compiles correct code sequences for this case.

Appendix A

Typical Characteristics of BINAR™

Typical Characteristics of the BINAR Chip

Phil Koopman
Robert Bryant

This document reports the "typical" operating characteristics of the BINAR chips in operation as of this date. It is not a formal characterization, but rather a document to give a general idea of operational characteristics. All measurements were made on actual hardware using chips from a typical wafer with an L(eff) of 1.05 microns. Operating conditions were using the BINAR prototype board on an extender card on an IBM PC at room temperature and approximately +5 volts.

AC Characteristics

Most operational chips run at maximum speeds of 17 to 20 MHz. BINAR boards are being populated with 32 MHz crystals, giving a 16 MHz operating speed. All references to clock rates are to the micro-cycle rate, which is one-half the crystal rate. The memory bus runs at one-half the micro-cycle rate (which is one-fourth the crystal rate), meaning that a 16 MHz system performs 8 MHz bus cycles.

Interfacing Delays

The main issue in interfacing to the BINAR is the required RAM chip speed for full-speed operation. Figure 1 shows typical delays for memory cycles.

The OSC signal is the input 2x oscillator. CLK is the clock signal output from the chip. The 13 ns delay from OSC to CLK is not important in practice, since external interface circuitry should use CLK as a primary reference.

The MEM-CYCLE signal is asserted during the first micro-cycle of a memory access. It is asserted 7 ns after the rising edge of CLK. This signal was used as the primary reference signal for other timings to make 'scope readings more accurate.

The delay from MEM-CYCLE to the time when the RAD pins are valid is 27 ns. This allows RAM chips to begin access before read/write information is available from the BINAR.

OE is asserted 7 ns after the first falling edge of CLK, approximately half-way through the time when MEM-CYCLE is asserted for RAM reads. It remains asserted until 17 ns after the rising edge of CLK at the end the memory cycle (which is 10 ns after the falling edge of MEM-CYCLE in the case of back-to-back memory cycles).

The WRx pins are asserted 10 ns after the falling edge of CLK, approximately half-way through the time when MEM-CYCLE is asserted for RAM writes. It remains asserted until 9 ns after the falling of CLK during the second micro-cycle of the memory cycle.

FRAM, which is intended as a chip enable signal for RAM chips, is changed 2 ns before the falling edge of MEM-CYCLE for each memory cycle. In I/O cycles, FRAM is first asserted, then deasserted at about the time when OE is asserted, since it takes a while before the processor can decide that an I/O cycle is in progress. ASIC is asserted with similar characteristics to OE.

For memory reads, RD must be valid from the memory chips with a setup time of 27 ns before the falling edge of MEM-CYCLE for the next memory cycle (20 ns setup time measured from the rising CLK edge if the processor is not performing back-to-back memory cycles.) While this may seem like a very long setup time, remember that this is only 7 ns before the rising edge of OSC that

triggers internal actions capturing the RD input value. The hold time for RD could not be measured, but is probably near 0 ns because of internal BINAR chip delays.

For memory writes, RD is asserted from the processor 9 ns after the second rising edge of CLK, approximately half-way through the micro-cycle when MEM-CYCLE is asserted. The asserted data remains valid until 18 ns after the next falling edge of MEM-CYCLE (which is 25 ns after the rising CLK edge associated with the end of the memory cycle).

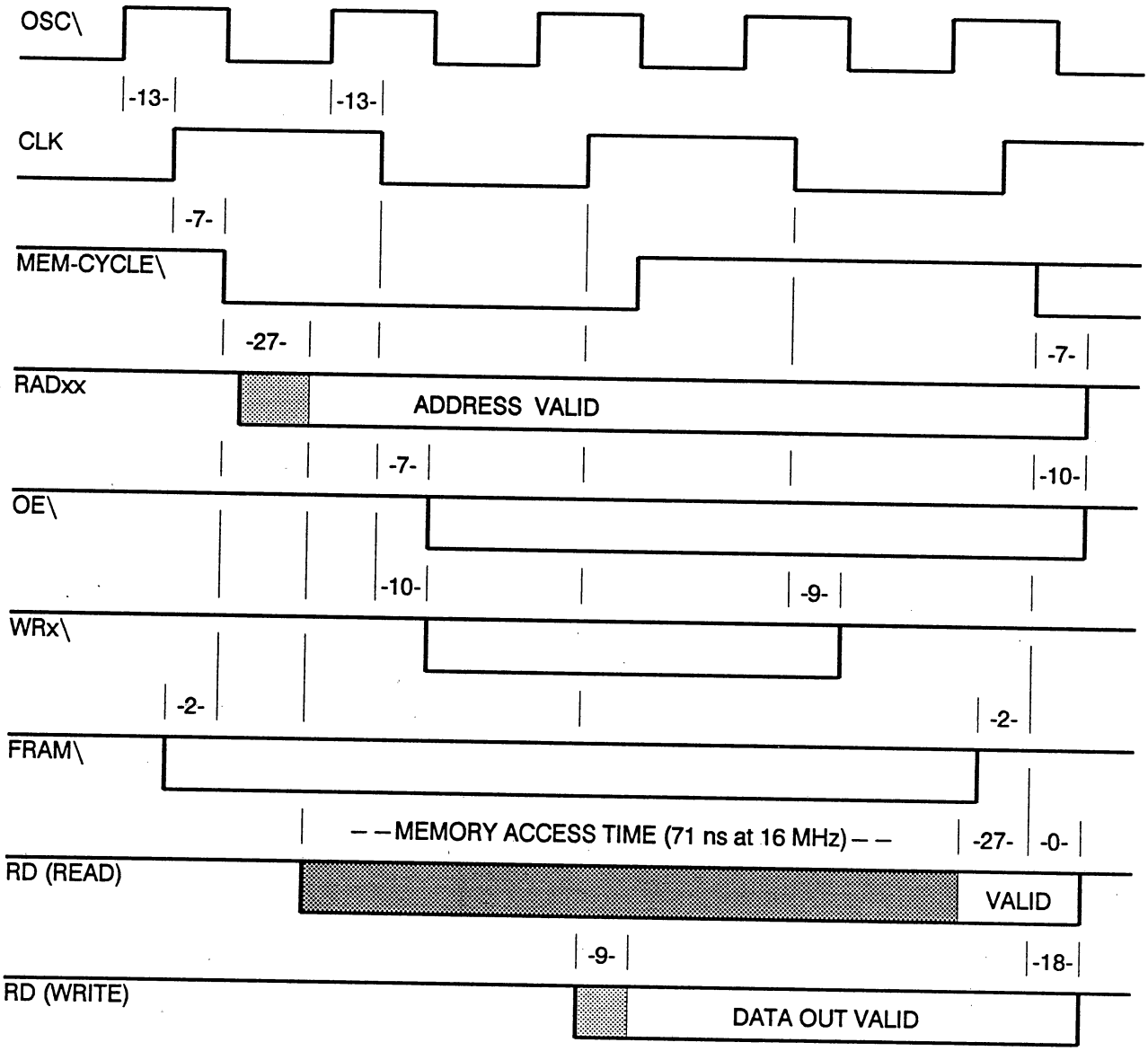


Figure 1. BINAR typical timing diagram.

Analysis

BINAR boards are currently populated by 35 ns SRAM chips, since they were on hand within Harris and it is difficult to find slower chips in 300 mil packages. The operating speed therefore appears to be limited by an internal chip critical path. This path is probably a delay through the bus mux, ALU, ALU carry logic, and zero detect logic. However, further studies using simulators will be conducted to identify and reduce delays in important paths.

At 16 MHz, memory writes require an SRAM chip which can accept a WE signal approximately 7 ns after the address has stabilized. The chips we are using have a WE address setup time of 0 ns, so this is no problem. The width of the write pulse at approximately 62 ns is sufficient to meet SRAM chip requirements of 25 ns for a 35 ns SRAM chip. This parameter should be watched carefully for higher operational speeds with slow RAM chips. Delaying WE by a few gate delays might help meeting write pulse requirements. Also, careful attention must be paid to the assertion of ASIC, FRAM, and WE to make sure no runt WE pulses are accepted by SRAM chips during an ASIC write cycle. The RD write data out should be moved back to the first falling edge of CLK to meet data valid times at higher operational speeds in later versions of the chip.

At 16 MHz, memory reads require an SRAM subsystem with an access time of 71 ns. Assuming a 6 ns address buffer chip, this mandates using 65 ns or faster SRAM chips.

The biggest room for improvement in the memory interface is in RAD valid time. There are several hardware optimizations that are being investigated to reduce the amount of time taken to drive a valid address from MEM-CYCLE. Also, the setup time could be reduced somewhat, but not too much since the indicated setup time is with reference to external signals that are significantly delayed beyond the internal signals used to actually latch data from the bus.

Currently, 57% of the bus cycle is actually available for memory subsystem access (the time between address valid and data setup requirements is 71 ns on a 125 ns bus cycle). This is a good start toward combining high operating speed with slow memory requirements. Further design efforts should be made to improve this ratio to better than 70% on the next silicon iteration.