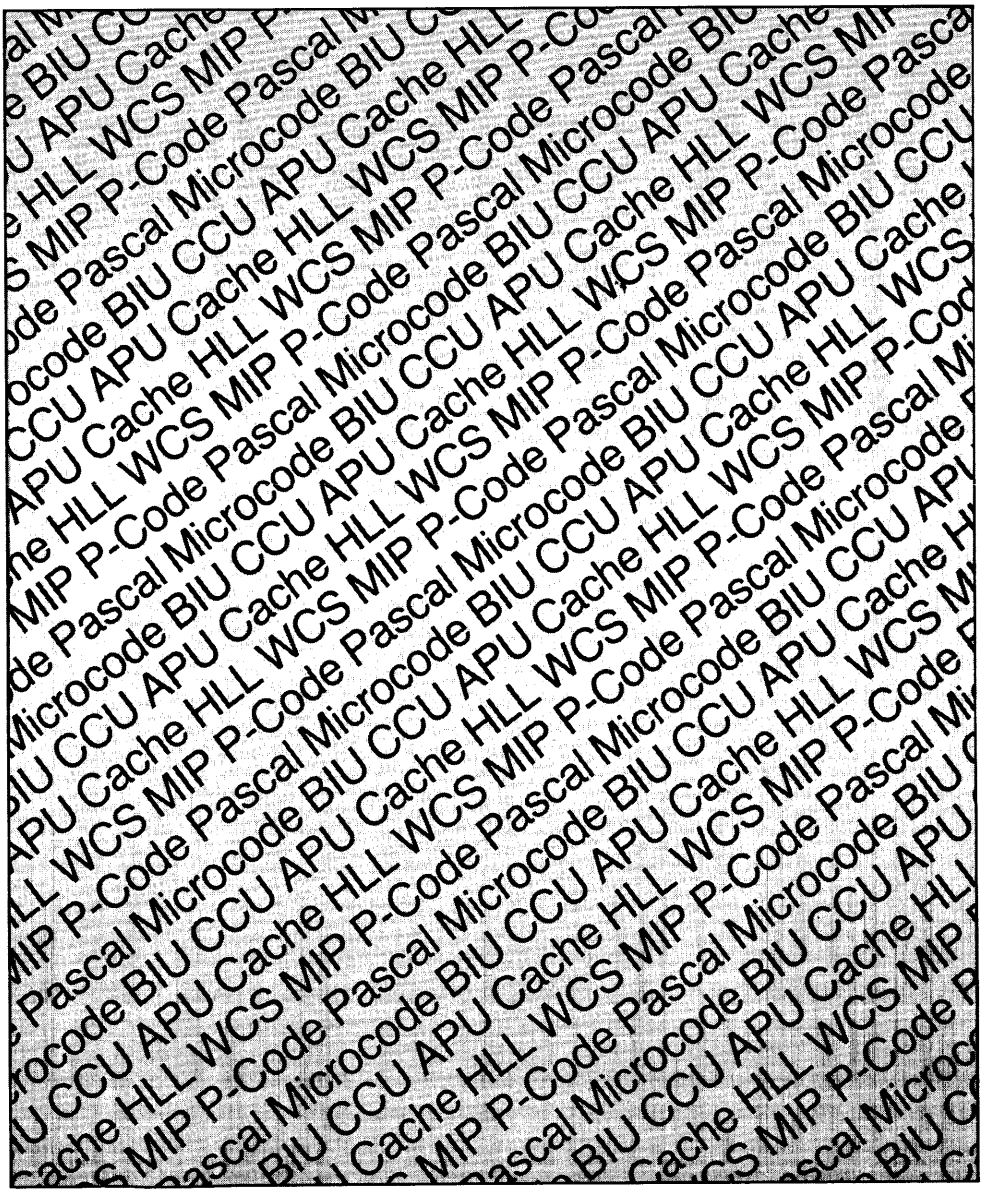
Am29116
A Microcoded Instruction Processor Based On The Am29116

Application Note
By Robert E. Anderson



ADVANCED MICRO DEVICES

The MIP Board: A Microcoded I-Code Processor Based on the Am29116

A. David Milton, Mitel
Robert E. Anderson, AMD
March, 1985.

Table Of Contents

INTRODUCTION

There are many general-purpose microprocessors available yet none shows significant advantages when used in high-level language and special purpose applications. A micro-programmed machine can be designed to take into account the general and specialized needs of a particular system. The resulting processor need not be much larger than a standard MOS-based micro-processor design; however, it will show significant improvement in performance.

This application note is the result of the authors' efforts in creating a high performance, 16-bit WCS (Writable Control Store) computer for practical as well as experimental use. Given the modifiable Control Store, along with considerable parallelism, such a computer is a perfect vehicle for high level language or protocol execution, or modifiable controller applications. The WCS shortens the development time needed to adapt the micro-programmed machine to different applications.

This project is not the result of any development activity at either AMD or MITEL, and neither company can be held responsible for the accuracy of this text or the design. The authors have tried to be as accurate as possible, and will update the text as discrepancies are noted. The information in this text is public property.

The Computer Itself

The main computer (sans I/O) resides on a board about 9 by 10 inches. An Am29116 is used as the processor. The computer has bulk RAM, a cache memory, a parallel multiplier, a pre-fetch buffer for instructions with its own program counter, a separate bus for I/O, and various registers and multiplexers to accommodate pipelined execution. Heavy use of VLSI and PAL devices allows it to compress all the afore mentioned, and a 4k × 32 WCS, onto one board. This compares favorably with standard MOS microprocessor designs.

Objectives

The name of this board, MIP, stands for Microcoded I-code Processor; it also alludes to the aim of one million high-level instructions per second of execution. The MIP board could be considered a 'working standard' microcoded instruction processor when comparing to novel architectures. The invitation is there to compare with board level computers done with MOS processors of conventional design.

Another objective is to create the ultimate personal workstation. UCSD Pascal has been ported so far, and it works well. Modula 2 for the next version of microcode is currently under consideration.

The Design

The design traces its ancestry back to similar processors built in the late seventies and early eighties, by several large companies. Some of these processors were used as minicomputers, and some as dedicated processors inside large machines such as telephone switches. The main advantage in using such a processor is that, given the flexible instruction set, performance can be optimized for a given application.

The Text

This application note should provide enough information for a design team to reproduce the MIP processor. In addition, some guidelines as to what tools to assemble, and what kind of effort should be required to complete the project are also included.

This application note is divided into three chapters. The first chapter covers hardware descriptions. This is divided into subsections describing the major functional blocks of the processor in detail. Units in block diagrams having numbers preceded by the letters A, B, C refer to IC's in the detailed schematic diagram. The first chapter should be used as a hardware reference manual for the MIP board.

The second chapter covers the software descriptions. This is divided into subsections describing both the low-level microcode software and the system-level software. The *Apple Pascal Reference Manual* should be referred to by readers new to this material. This chapter should be used as a software reference manual for the MIP board.

The third chapter covers performances. This covers the means used to measure performance, benchmarking, and how to tailor the software via intrinsic functions to enhance performances.

Chapter 3 concludes with a review of the design and a discussion of promising areas for future work. Appendix B provides brief descriptions of some of the AMD 29XXX family parts used in this application note. Detailed technical questions can be directed to the AMD Applications Department at (408) 982-6266.

Chapter 1

HARDWARE DESCRIPTION

1.0 OVERVIEW

To best describe the processor, the entire design is divided into a number of functional blocks which more or less operate autonomously. The functional blocks are treated separately and their inter-relationships are shown.

Figure 1.1 shows the major functional units of the MIP board as they are connected together by two main internal busses. The processor memory, including the cache, is accessed via the BIU; I/O access is provided by the I/O unit, which is essentially another BIU. The major functional units mentioned are the only units with direct connection to the YBUS.

Abbreviations

The abbreviations used in this application note are as follows:

APU	=	Arithmetic Processor Unit
CCU	=	Computer Control Unit
BIU	=	Bus Interface Unit
MSD	=	Microstore Data Bus
YBUS	=	Processor Data Bus
DBUS	=	Data Bus
ABUS	=	Address Bus

The ABUS runs between the BIU and the memory. There is another bus, named the S/D Bus (for Source/Destination), which is really just a collection of all the strobes and signals on board that did not get mapped otherwise, and which fulfill control functions.

In the remainder of this overview section, these units and busses are summarized.

The MSD Bus

The MSD Bus (Microstore Data Bus) is output from the CCU (Computer Control Unit). The MSD Bus controls the four major functional blocks on board: the APU, BIU, I/O UNIT, and the CCU itself. In turn, within the CCU, the MSD Bus is interpreted and various strobes and signals are sent out to control registers and buffers on board.

The YBUS

The YBUS is controlled by the CCU and provides a high-speed data path between the APU (Arithmetic Processor Unit), and other parts of the system. Data on the YBUS is gated to the other busses on board, in accordance with MSD directives and in synchronization with these other busses.

The CCU

The CCU (Computer Control Unit) is the source of the MSD Bus which provides instructions to the processor and the rest of the system. The CCU also handles all of the timings at the micro-instruction level, and produces the S/D Bus signals to synchronize and control gating onto all of the busses in the system.

The APU

The APU (Arithmetic Processor Unit) provides all of the arithmetic and logical functions of the processor. Special data-shift operations and multiply functions are also handled here.

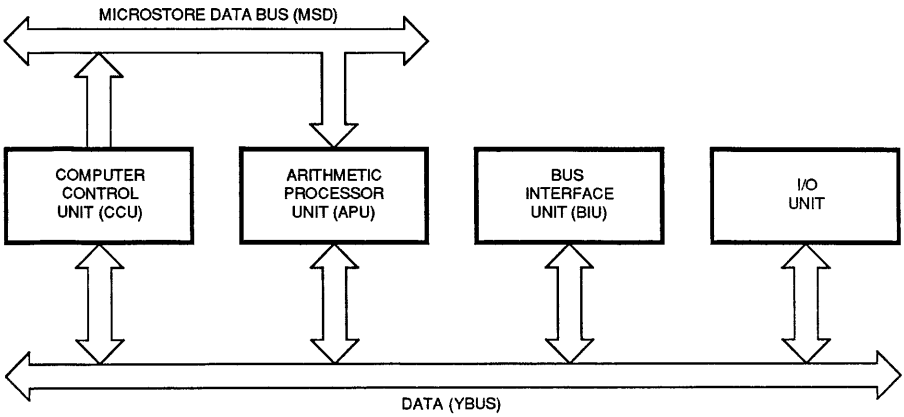


Figure 1.1 MIP Block Diagram

The BIU

The BIU (Bus Interface Unit) contains address and data registers to interface the MIP to the main memory. Some of these registers are general purpose, and some are used only for high-level language execution.

The I/O Unit

The I/O unit is essentially another BIU. It contains a bi-directional data register to interface to a high-speed I/O channel.

1.1 DETAILED DESCRIPTIONS

The Arithmetic Processing Unit (APU)

An APU is the part of a computer which performs arithmetic and logical operations on data. There are usually a number of general purpose registers within the APU which may be used for temporary storage of variables.

When dealing with high-level language concepts, a number of special purpose functions are often required; such as data shift and field isolation, bit operations, prioritize operations, and multiply and divide. These functions are also contained within the APU.

In the MIP computer, the APU consists of a Am29116, a Am29517, a Condition Code PAL device (CCPAL) and diagnostic registers for the YBUS. The Am29116 provides the bulk of the arithmetic functionality and the register file. All data shifting and field isolation capability are contained within the Am29116 instruction set, except for dynamic shifts, which are augmented by overlaying a field in the CCU. Certain bit-oriented and rotate instructions normally receive a count from the CCU (via the MSD Bus). These instructions may be modified by the CCU so that the count is dependent on the data on the YBUS.

All multiplications are done by the Am29517. Several formats are available with this device to suit different numerical algorithms. Divides are done using the Am29116 in a two-cycle-per-bit divide loop. To do faster divides, the Newton-Raphson method could be used.

The MSD Bus supplies instructions for the Am29116 and Condition Code PAL device circuit. The Source/Destination Control Bus supplies decoded control signals (i.e. strobes, clocks, etc.) to perform major sequencing.

The APU circuit is shown in detail in Appendix C. The instruction code going into the Am29116, I_9 to I_{12}, is from MSD bits 32-35. B20 creates these bits from MSD bits 9 to 12, or YBUS bits 0 to 3. This allows the YBUS to specify the count, or bit number, for certain operations in the Am29116. This is the family of dynamic bit-shift, rotate and field isolation instructions.

The Am29818 diagnostic pipeline registers are used to read or write to the YBUS when testing. During testing, if $\overline{OEY}$ is held High, any value can be placed on the YBUS to load any register. The Am29818 YBUS registers have also proved useful in some microcode sequences.

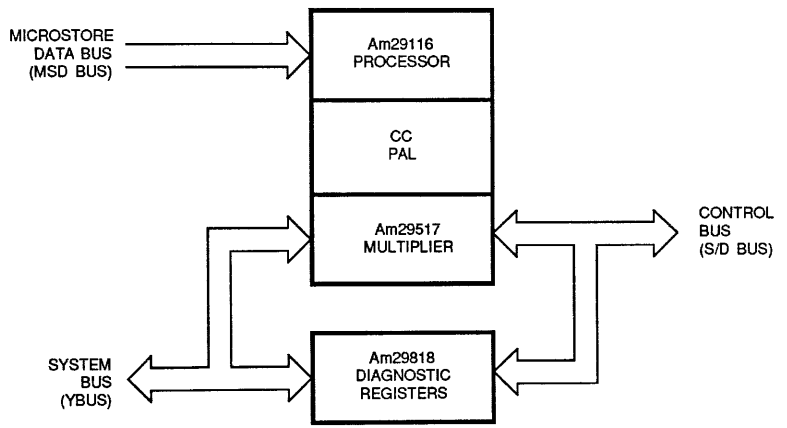


Figure 1.2 APU Block Diagram

The CCPAL accomplishes latching and decoding functions for the condition code ($\overline{CC}$). This is used by the CCU for conditional branches. Condition codes from the Am29116 are latched externally to give pipelined execution and improve the cycle time of the processor.

MSD bits 12–15 determine the polarity and selection criteria for the condition code ($\overline{CC}$) from the status bits.

T1 thru T4, and CT from the Am29116 are latched when PLCLK goes from Low to High at the start of each cycle, if $\overline{SRE}$ is Low. If $\overline{SRE}$ is High, the latched codes are retained (Figure 1.3). T1 thru T4 are the standard arithmetic flags: Zero, Sign, Carry and Overflow. CT is generated by a class of instructions in the Am29116 and is simply a delayed function of T1 thru T4. PLCLK has a 50% duty-cycle, except in the case of Wait States which extend the Low phase by integral cycles (62.5 ns each for an 8 MHz PLCLK).

The CCPAL equations show how the latching and decoding are done (Figure 1.4). The APU, as constructed, combines the Am29116, the Am29517, and Enhanced Condition Code handling.

The Computer Control Unit (CCU)

The CCU controls and synchronizes the various units of the computer. It provides an ordered set of instructions to the rest of the machine. The flow of these instructions may depend on the data.

An overview of the CCU is shown in Figure 1.5. It consists of 3 main sections. These sections are: the Microstore Control Section, the Exception (Interrupt) Control Section, and the Cache and Register Control Section.

The Microstore Control Section is the source of the microcode instruction for all parts of the machine. The Cache and Register Control Section 'cracks' microcode fields into timing strobes which form collectively the S/D Bus.

The Exception Control Unit (ECU) handles irregular control transfers (i.e. interrupts). In some cases the ECU may be simplified, or absent if the application does not require it.

Microstore Control Section

Instruction execution in the processor is controlled by a single clock, PLCLK. A micro- instruction cycle is defined by a period of PLCLK. This clock is a 50% duty-cycle signal with a nominal period of 125 ns The Low period of this clock may be extended by 62.5 ns increments to accommodate timing conflicts in the BIU, or may be held Low by the external HALT line.

The Microstore Control Section is shown in Figure 1.6. The Micro-Sequencer (Am2910A) creates a Microstore Address which accesses the Control Store. The Control Store data is latched at the beginning of each pipeline clock cycle and forms the MSD Bus. Most of the MSD bits go to other parts of the machine, but 4 bits (28–31) are used to control the Am2910A sequencer itself.

A portion of the MSD data may be used to form a branch address by gating the lower 16 bits onto the YBUS. This machine uses a compressed Micro Store Word (vertically coded). The micro store branch address and condition code fields are overlapped with the Am29116 control field. This means that a given instruction may cause conditional branching to occur or may be used for an APU operation.

The WCS Section, shown in Figure 1.7, has eight 4K x 4 static RAMs (AMD9968 CMOS static RAMs) which are used for microcode data. Four Am29818 diagnostic pipeline registers are used for single-level pipelining of the microstore (RAM) data. Two Am29818 registers (Am29818 address access) can

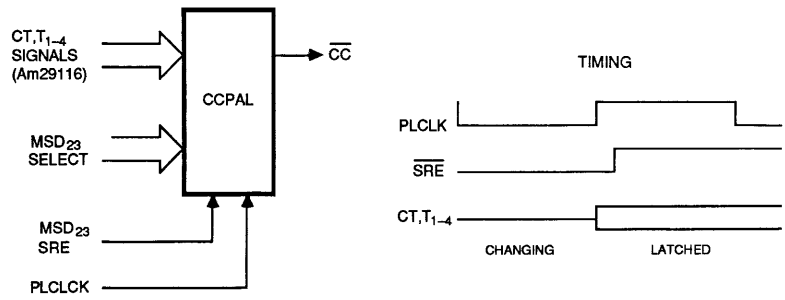


Figure 1.3 Condition Code PAL Device Circuit

```
PAL16L8
;
; Condition Code Pal
; ccl.pal.text
;
POL,C,B,A,CT,T1,T2,T3,/SRE,GND,PLCLK,
/CC0,/CTL,T4,/LT4,/LT3,/LT2,/LT1,/CC1,VCC
;
CC0 =/C*/B*/A* CTL      ; active low
conditions
    +/C*/B* A* LT1      ; nct,nz,nc,p,novr
    +/C* B*/A* LT2
    +/C* B* A* LT3
    + C*/B*/A* LT4
    @POL.
;
CC1 =/C*/B*/A*/CTL      ; active high
                          conditions
    +/C*/B* A*/LT1      ; ct,z,c,n,ovr
    +/C* B*/A*/LT2
    +/C* B* A*/LT3
    + C*/B*/A*/LT4
    + C* B* A           ; unc
    @/POL.
;
; CT latch
;
CTL = /CT*SRE*/PLCLK    ; sample if status
                          during last part
    + CTL*/SRE          ; keep if no status
                          update
    + CTL*PLCLK         ; hold while plclk
    + CTL*/CT.
;
; T1 latch
;
LT1 = /T1*SRE*/PLCLK    ; sample if status
                          during last part
    + LT1*/SRE          ; keep if no status
                          update
    + LT1*PLCLK         ; hold while plclk
    + LT1*/T1.
    ;
; T2 latch
;
LT2 = /T2*SRE*/PLCLK    ; sample if status
                          during last part
    + LT2*/SRE          ; keep if no status
                          update
    + LT2*PLCLK         ; hold while plclk
    + LT2*/T2.
;
; T3 latch
;
LT3 = /T3*SRE*/PLCLK    ; sample if status
                          during last part
    + LT3*/SRE          ; keep if no status
                          update
    + LT3*PLCLK         ; hold while plclk
    + LT3*/T3.
;
; T4 latch
;
LT4 = /T4*SRE*/PLCLK    ; sample if status
                          during last part
    + LT4*/SRE          ; keep if no status
                          update
    + LT4*PLCLK         ; hold while plclk
    + LT4*/T4.
;
END
```

Figure 1.4

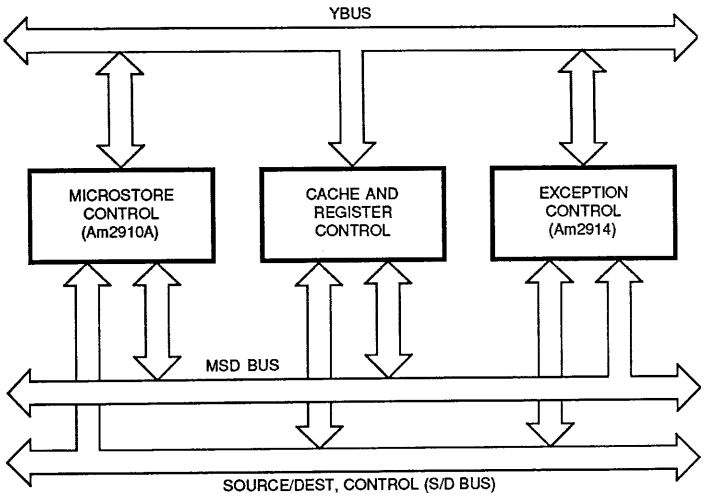


Figure 1.5 CCU Block Diagram

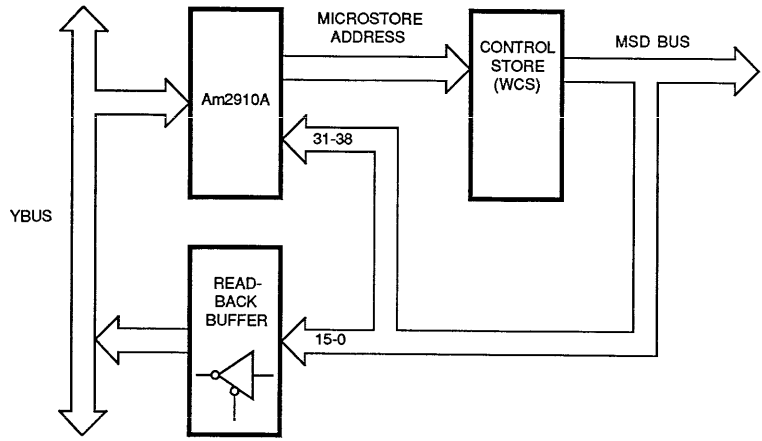


Figure 1.6 Microstore Control Section

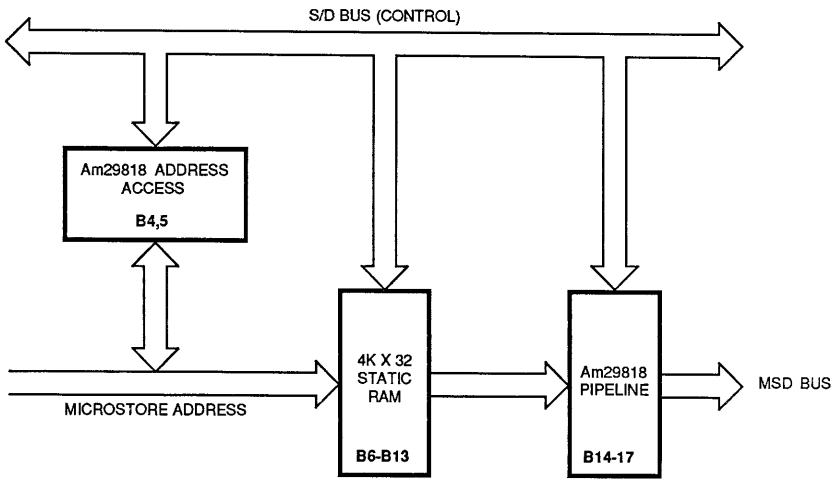


Figure 1.7 Writable Control Store (WCS)

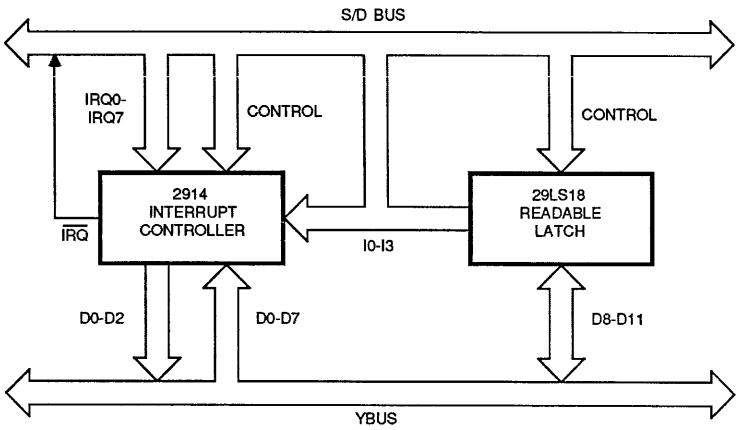


Figure 1.8 Exception Control Section

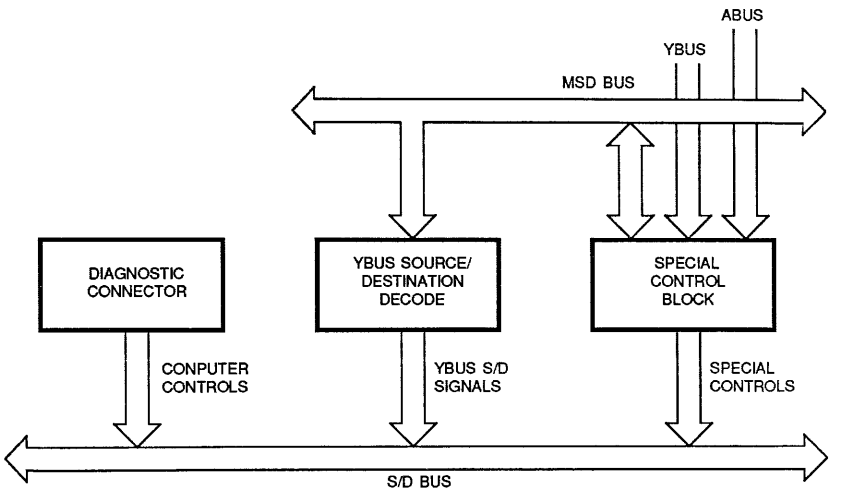


Figure 1.9 Cache and Bus Control Section

jam the Microstore Address Bus and thus allow the Microstore Memory to be stored from the pipeline registers. The address access registers also help, during diagnostics, by providing a convenient way to look at the address from the Am2910A.

Exception Control Section

The Exception Control Section (Figure 1.8) latches and regulates interrupt requests based on instructions from the MSD bus.

The Exception Control Section uses the Am2914 Interrupt Controller. Interrupts $\overline{IRQ0}$ thru $\overline{IRQ7}$ are prioritized and encoded to a 3-bit value. IRQ is generated and when the processor reads the Am2914, the 3 Low bits on the YBUS give the interrupt number.

The Am29LS18 outputs (I_0–I_3) are shared between the Am2914 and the Am29517. The interrupt enable signal, $\overline{IEN}$, is used to separately qualify instructions to the Am2914 (I_0–I_3). When the Am29517 is being used, $\overline{IEN}$ is held High.

Cache And Bus Control Section

The Cache and Bus Control Section (Figure 1.9) produces almost all of the signals for the S/D Bus. For clarity, this section is divided into three parts: the Diagnostic Connector, the YBUS Source/Destination Decode Block, and the Special Control Block.

The Diagnostic Connector provides access to the diagnostic pipeline registers and the timing logic of the processor. It is used to examine processor status, load the WCS, or change some data value in the processor or its bulk memory. With this facility, it is possible to debug hardware, micro-code or high-level software. The signals on this interface are not time critical so the diagnostic device can be kept simple and need not have super performance to be useful.

The YBUS Source/Destination Decode Block decodes two fields of the MSD Bus (16–19, 20–22) into the respective register and buffer control signals for devices connected to the YBUS.

The Special Control Block integrates a number of functions which are necessary for the BIU. Due to the critical timing nature of some of the signals and the overall control relationship to the processor, they are included in the CCU. This also leads to a compression in the amount of circuit real estate devoted to these functions.

The YBUS is the main data highway of the processor. All of the functional blocks attach to the YBUS. A set of fields in the micro-store word control activity on this bus. One data value can be moved from source to destination per pipeline cycle. The diagnostic registers (Am29818's) attached to the YBUS unconditionally capture the data at the end of each cycle.

YBUS Source and Destination Control Section

The YBUS Source and Destination Decode Block produces the Source and Destination control signals (S/D Bus) which are used to control register access to or from the YBUS (Figure 1.10). The signals are defined as follows:

Signal	Meaning
$\overline{OEY}$	Enable output on Am29116
$\overline{DREN}$	Enable incoming data register (BIU)
$\overline{YREG}$	Enable output of Am29818 registers on YBUS
$\overline{IODEN}$	Enable I/O unit incoming data
$\overline{IREN}$	Enable instruction register (This signal is further qualified in regctl PAL.)
$\overline{OEPM}$	Enable high product from Am29517 to YBUS
$\overline{OEPL}$	Enable low product from Am29517 to YBUS
$\overline{LDIOC}$	Move data into I/O control register
$\overline{LDDR}$	Move data into data output register (BIU)
$\overline{LDPC}$	Move data into program counter reg. (BIU)
$\overline{LDAR}$	Move data into address register (BIU)
$\overline{DLE}$	Move data to Am29116 (This signal is changed to DH by STEP PAL device in Diagnostics Section before feeding into the Am29116.)
$\overline{LDPG}$	Move data to high address reg. (BIU)
$\overline{LDPCPG}$	Move data to high PC-address reg. (BIU)
$\overline{LDIOA}$	Move data to I/O address reg.
$\overline{CVECT}$	Enable MSD bus to YBUS (Bits 15–0) enable branching
$\overline{ENY}$	Enable Y register load on Am29517
$\overline{ENX}$	Enable X register load on Am29517
$\overline{IEN}$	Enable instruction for Am2914
$\overline{ABEN}$	Enable readback on address bus (BIU)
$\overline{PGEN}$	Enable readback on high address bus (BIU)
$\overline{YSHIFT}$	Cause YBUS 3–0 to overlay MSD BUS 35–32, which are normally MSD BUS 12–9

Diagnostic Section

The Diagnostic Section (Figure 1.11) consists of the Diagnostic Connector and part of 'STEP' PAL device (B24). The Diagnostic Connector pin numbers are as shown in Figure 1.11. The 'STEP' PAL device, as the name suggests, is used to synchronize the incoming signals from the Diagnostic Connector and cause the XWAIT signal to be well-behaved with respect to the master clock of the processor.

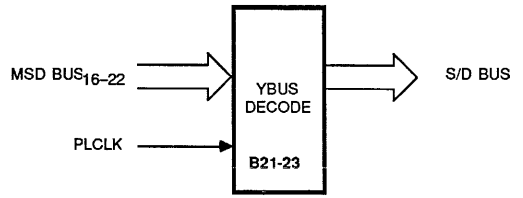


Figure 1.10 YBUS Decode Section

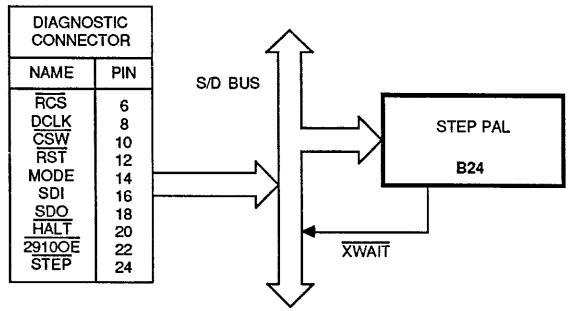


Figure 1.11 Diagnostics Section

```
PAL16R4
;
;
; MIP Processor Cache enables and step
control
; step.pal.text
;
CP16M,/PLCLK,/DLE,/RESET,/HALT,/STEP,A0,/BYTE,

A16,GND,/EN,/C1L,/C1H,/XWAIT,/ST2,/ST1,/DH,
/C0H,/C0L,VCC
;
DH := DLE*PLCLK*/RESET      ; hold data if
                              dle
   +  DH*/PLCLK*/RESET.     ; no change
                              while plclk high
;
ST1 := STEP*/RESET.         ; detect the
                              step pulse
;
ST2 := ST1*/RESET.          ; stage 2
;
XWAIT := HALT*/ST1*/ST2       ; normal halt
      + HALT*ST1
      + HALT*ST1*ST2.         ; do step when
                                /st1*st2
;
; Cache chip selects
;
C0L = /A16*BYTE
    + /A16*A0*/BYTE.
;
C0H = /A16*/A0*/BYTE
    + /A16*BYTE.
;
C1L =  A16*BYTE
    +  A16*A0*/BYTE.
;
C1H = A16*/A0*/BYTE
    + A16*BYTE.
;
END
```

Figure 1.12 Step PAL Equations

The meanings of the mnemonics in the Diagnostic Connector are given below:

Signal Name	Utility
$\overline{RCS}$	Read Control Store (Normally Asserted) causes MSD RAMs to assert data
DCLK	Data Clock (Normally Low) clocks data into Am29818 diagnostic register
$\overline{CSW}$	Control Store Write (Normally High) strobes data into MSD RAMs
$\overline{RST}$	General Reset signal (Normally High)
MODE	Am29818 control signal.
SDI, SDO	Serial Data for Am29818 devices
$\overline{HALT}$	Processor Halt signal
Am2910$\overline{OE}$	Output enable for Am2910A (Normally Asserted)
$\overline{STEP}$	Single Step control signal.

The PAL equations for 'STEP' PAL device are shown in Figure 1.12. The cache decode signals share this PAL device with the Diagnostics Section. Since the cache has four 2k × 8 RAMs, 4 cache enables are required to provide for byte access. The signals have the following functions:

$\overline{DH}$	A signal for the input data latch of the Am29116, follows DLE sampled by PLCLK (micro-program uses this to control input latch)
ST1, ST2	Registers used to detect edge of STEP for single step of the processor
$\overline{XWAIT}$	The control signal derived from HALT and ST1 and ST2 which is used to stop the processor from an external device
C0L,C0H C1L,C1H	Individual chip enables for the cache RAMs

Special Control Block

The Special Control Block consists of four distinguishable sections (Figure 1.13). There is a Cache Byte Decode Section to aid in decoding cache accesses, a Register Control Section, a Bus Sequence Control Section, and a MSD multiplexer (MUX).

The Cache and Byte Decode Section is shown in Figure 1.14. Part of the 'STEP' PAL device is used to demultiplex the address and byte-op information given. It is necessary to decode to the byte-level for the cache because the processor can do byte reads and writes. A16 is part of the Address Bus (A16 is '1' for C1L and C1H, '0' for C0L and C0H). The 4 individual byte-wide cache outputs are controlled from here.

Part of 'SHIFT' PAL device is used to control byte overlay for accesses over the DBUS. During memory byte operations, the byte being read or written is right justified in the data register of the BIU.

The Shift PAL equations are shown in Figure 1.17A, since most of the PAL device is used in the 'SHIFT' function.

The Register Control PAL device controls the instruction fetch, and takes care of interrupt vectoring and cycle stretching for store conflicts. The outputs have the following interpretations:

$\overline{IRLEN}$, $\overline{IRHEN}$	Byte wide enables for the instruction register
$\overline{PCINC}$	Program counter increment if no interrupt
$\overline{IRQV}$	Enable interrupt vector to YBUS if interrupt accepted
$\overline{COE}$	Enable cache if read and hit
$\overline{STREQ}$	Store request for cycle in progress means BIU about to be busy. If BIU is already busy, there will be a wait

The Bus Sequence Control Section controls accesses over the data and address bus to the cache and main memory (Figure 1.16). The BSEQ PAL device provides PLCLK, detects Data Acknowledge, requests an external memory cycle if the cache misses, and latches and holds the memory operation (MSD 27–25) until the cycle is completed. The microstore instruction will specify that a certain type of bus cycle be performed. This instruction is loaded into the Bus Sequencer and the cycle is started. The microprogram sequencer can continue onto other instructions while this cycle is in progress. If another bus cycle instruction (other than NOP) arrives while the Bus Sequencer is busy, the Wait line will be asserted until the current bus cycle completes.

There are also other conditions which can cause this Wait to occur. If the micro-instruction specifies a data or instruction register, while a bus cycle is in progress to fill that register, a Wait will occur until the register is filled.

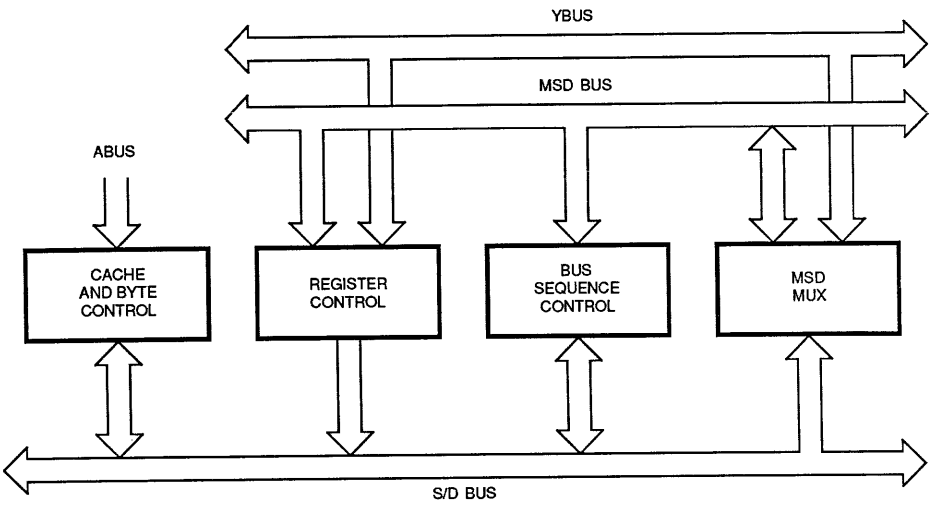


Figure 1.13 Special Control Block

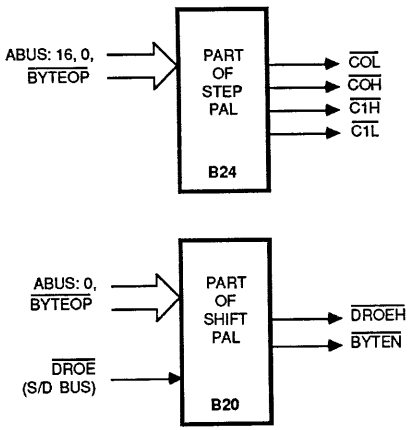


Figure 1.14 Cache and Byte Decode Section

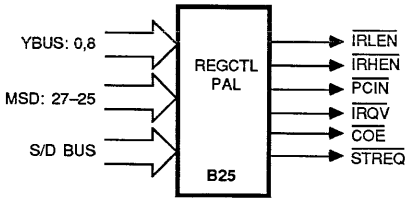


Figure 1.15 Register Control Section

```
PAL16L8
;
; MIP Processor Register Control Pal
; regctl.pal.text
;
R0,R1,R2,/IRQ,PC0,/DREN,/HIT,/IREN,/LDPC,GND,
YBUS0,/IRLEN,/IRHEN,/PCINC,/IRQV,/DRLD,/IRLD,/COE,/STREQ,VCC
;
; Instruction register enables if no vector interrupt
;
IRLEN = /PC0*IREN*/IRQ.          ; low byte if no int
;
IRHEN = PC0*IREN*/IRQ.           ; high byte if no int
;
; Program counter increment if no vector interrupt
;
PCINC = IREN*/IRQ                ; inc pc if no vector interupt
      + LDPC*YBUS0*/PC0          ; loadpc & new <> pc0 then inc
      + LDPC*/YBUS0*PC0.
;
; interrupt vector bit generation
;
IRQV = IREN*IRQ                  ; decode vector interrupt condition
      @IREN*IRQ.                 ; enable vectoring to ybus
;
; Cache output enable
;
COE  = IRLD*HIT                  ; cache enable if read and hit
     + DRLD*HIT.
;
; Store request for processor cycle in progress
;
STREQ = R2                        ; 4 - 7 data store ops and waits
      + /R2    * R0               ; 1 & 3
      + /R2* R1*/R0*PCINC*PC0.    ; 2     auto fetch program store
;
END
```

Figure 1.15A REGCTL PAL

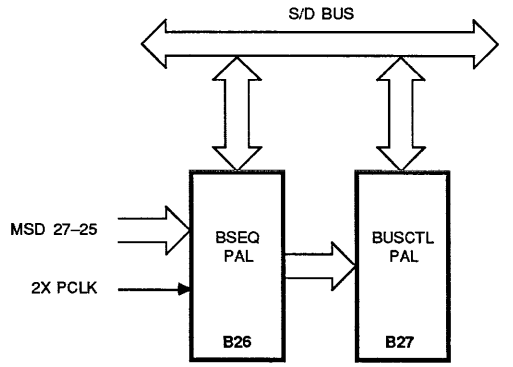


Figure 1.16 Bus Sequence Control Section

The types of bus cycles which may be performed are Read and Write Word, using AR (address register) and DR (Data Register), Read and Write Byte, using AR and the low-order half of DR, Read Word, using PC (Program Counter) and IR (Instruction Register), and Conditional Read Word, using PC and IR. This last cycle is used when fetching op-codes for high-level language execution. If the PC is odd and the IR is being accessed, a bus cycle will start to automatically fill the IR, using the value of the PC after it has been incremented by one.

A NOP instruction is also available for the Bus Sequencer. This causes no action to be done to memory, nor will it cause a Wait. There is a WIO (Wait on I/O) op-code for the Bus Sequencer control logic. It also causes no memory activity but does cause a Processor Wait if the memory is being used. The Bus Control Sequencer signals have the following functions:

PLCLK	Is the master instruction clock for the processor
$\overline{\text{DTAK}}$	The synchronized acknowledge from a memory device or cache
$\overline{\text{XRQ}}$	Active during a memory request when there is no HIT on the cache (always active during write cycles)
R0, R1, R2	The latched memory store operation in progress
$\overline{\text{ACTIVE}}$	Indicates a memory cycle in progress
PC0	A copy of the LSB of the program counter used to point to the proper IR byte and to determine when to do the next program store fetch.

The BUSCTL PAL device is controlled by the BSEQ PAL device. The BUSCTL PAL device creates $\overline{\text{BYTEOP}}$ and decodes signals $\overline{\text{DRLD}}$, $\overline{\text{DROE}}$, $\overline{\text{PCEN}}$, $\overline{\text{AREN}}$, and the Wait State criteria. The four decoded signals control the data flow on the main memory data bus,'DBUS', and determine which address register is to supply the address for the bus cycle.

This PAL device also does the cache write enable if one is allowed by the UPDATE signal ($\overline{\text{UPDT}}$) from the cache logic. Figure 1.16A gives the equations for the MIP Processor Bus Control PAL device. For the BUSCTL PAL device (Figure 1.16B) the signals have the following functions:

$\overline{\text{IRLD}}$	Loads the Instruction Register (IR) when data is available from a Program Store Read cycle
$\overline{\text{DRLD}}$	Loads the Data Register (DR) when data is available from a Data Store Read
$\overline{\text{BYTEOP}}$	Signifies that the current cycle is a byte operation
$\overline{\text{CWE}}$	Is the write enable for the Cache Data and Tag Memory, it occurs when valid data is written into the cache.
$\overline{\text{DROE}}$	Is the enable for data register write to memory
$\overline{\text{PCEN}}$	Is active when the Program Counter provides the memory address
$\overline{\text{AREN}}$	Is active when the Address Register provides the memory address
$\overline{\text{WAIT}}$	Causes the processor to wait when a conflict occurs within the BIU

```
PAL16R8
;
; Mip Processor Bus Control Sequencer Pal
; bseq.pal.text
;
CP16M,M25,M26,M27,/WAIT,/DACK,/HIT,/PCINC,/RESET,GND,
/EN,/PLCLK,PC0,/DTK,/ACTIV,/XRQ,R2,R1,R0,VCC
;
; main pipe clock for the processor 1 cycle = 1 instruction
;
PLCLK := /RESET*/PLCLK                          ; going low
       + WAIT*/RESET.                           ; wait request
;
; data acknowledge from either external memory or the cache
; only generate dtk for state 4 & 7 when not in external wait
;
DTK := XRQ*DACK*ACTIV*/RESET                   ; dack detect normal cycles
     +  /R2*/R1*R0*ACTIV*HIT*PLCLK*/RESET      ; 1 dtk for cache hit
     +  /R2*R1*    ACTIV*HIT*PLCLK*/RESET      ; 2 & 3 dtk for cache hit
     +   R2*/R1*R0*ACTIV*HIT*PLCLK*/RESET      ; 5 dtk for cache hit
;
; external memory request if data not in cache or a write cycle
;
XRQ := /R2*/R1* R0*/XRQ*/DTK*/RESET*ACTIV*/HIT*PLCLK  ; 1
     +  /R2* R1*    /XRQ*/DTK*/RESET*ACTIV*/HIT*PLCLK ; 2 & 3
     +   R2*/R1* R0*/XRQ*/DTK*/RESET*ACTIV*/HIT*PLCLK ; 5
     +   R2*    /R0*/XRQ*/DTK*/RESET*ACTIV*PLCLK      ; 4 & 6 write thru cache
     +   XRQ*/DTK*/RESET.
;
; store request codes r0 - r2 are transparent until an active cycle
; then the latch the code for that cycle until it is completed
;
   /R0 :=/M25*/ACTIV*/RESET   ; transparent while not active
        + /R0*ACTIV*/RESET.   ; then latch last state
   ;
   /R1 :=/M26*/ACTIV*/RESET   ; transparent while not active
        + /R1*ACTIV*/RESET.   ; then latch last state
   ;
   /R2 :=/M27*/ACTIV*/RESET   ; transparent while not active
        + /R2*ACTIV*/RESET.   ; then latch last state
   ;
   ; signifies that a an active store request is in progress
   ; until an acknowledge of some sort is generated
   ; may not go active until wait goes away
   ;
   ACTIV := ACTIV*/DTK*/RESET                          ; staying active
          + M27*/M26    */ACTIV*/DTK*/RESET*PLCLK*/WAIT  ; 4 & 5
          + M27*M26*/M25*/ACTIV*/DTK*/RESET*PLCLK*/WAIT  ; 6 data write
          +/M27     *M25*/ACTIV*/DTK*/RESET*PLCLK*/WAIT  ; 1 & 3
          +/M27*M26*/M25*PCINC*PC0*/ACTIV*/DTK*/RESET*PLCLK*/WAIT.
   ; 2 if pcinc
   ;
   ; a copy of the PC LSB for instruction register uses
   ; allows overlapped AR activity and IR use
   ;
   /PC0 := /PCINC*/PC0*/RESET              ; keep what got
         +  /PC0*WAIT*/RESET               ; hold while waiting
         +  PCINC*/PLCLK*/PC0*/RESET       ; hold while PLCLK high
         +  PLCLK*PCINC*PC0*/WAIT*/RESET.  ; active if no waits
   ;
   END
```

Figure 1.16A BSEQ PAL Device

```
PAL16L8
;
; Mip Processor Bus Control Pal
; busctl.pal.text
;
R0,R1,R2,/STREQ,/IREN,/DREN,PLCLK,/UPDT,/ACTIV,GND,
/DTAK,/IRLD,/BYTEOP,/DRLD,/CWE,/DROE,/PCEN,/AREN,/WAIT,VCC
;
; Store control loads instruction register
;
IRLD =/R2*/R1* R0*ACTIV*/DTAK          ; 1 RPS
     +/R2* R1*/R0*ACTIV*/DTAK.         ; 2 CRPS
;
; Store control loads data register
;
DRLD = /R2* R1 R0*ACTIV*/DTAK          ; 3 RDSB
     +  R2*/R1* R0*ACTIV*/DTAK.        ; 5 RDS
;
; Byte wide data op
;
BYTEOP = /R2* R1 * R0 * ACTIV          ; 3 RDSB
       +  R2*/R1 */R0 * ACTIV.         ; 4 WDSB
;
; Cache write control on current store control cycle
;
CWE =/R2*/R1* R0*ACTIV*UPDT*DTAK       ; 1 RPS
    +/R2* R1*/R0*ACTIV*UPDT*DTAK       ; 2 CRPS
    + R2*/R1*/R0*ACTIV*UPDT*DTAK       ; 4 WDSB
    + R2*/R1* R0*ACTIV*UPDT*DTAK       ; 5 RDS
    + R2* R1*/R0*ACTIV*UPDT*DTAK.      ; 6 WDS
;
; Store control enables data out register (also called WE)
;
DROE = R2 * /R1 * /R0 * ACTIV          ; 4 WDSB
     + R2* R1*/R0*ACTIV.               ; 6 WDS
;
; Store control using PC to supply address
;
PCEN =/R2*/R1* R0                      ; 1 RPS
     +/R2* R1*/R0                      ; 2 CRPS
     +/R2*/R1*/R0.                     ; 0 to read back PC
;
; Store control using AR to supply address
;
AREN = /R2* R1* R0                     ; 3 RDSB
     + R2.                             ; 4 - 7
;
; New store request conflicts with cycle in progress
;
WAIT =ACTIV*STREQ*/PLCLK               ; store active and pending request
     +/R2*/R1* R0*ACTIV*/PLCLK*IREN    ; 1 RPS  not done before IREN
     + R2* R1*/R0*ACTIV*/PLCLK*IREN    ; 2 CRPS not done before IREN
     +/R2* R1* R0*ACTIV*/PLCLK*DREN    ; 3 RDSB not done before DREN
     + R2*/R1* R0*ACTIV*/PLCLK*DREN    ; 5 RDS  not done before DREN
     + DTAK*STREQ*/PLCLK.              ; store active and pending request
;
END
```

Figure 1.16B BUSCTL PAL Device

The MSD MUX (Figure 1.17) provides a method whereby the rotation, or bit number can be passed from the YBUS back into the micro-instruction. When YSHIFT is asserted, the YBUS data overlays regular MSD data bits. This is caused by a particular YBUS store code in the MSD control word. The equations for 'SHIFT' PAL device are shown in Figure 1.17A.

The SHIFT PAL signals have the following functions:

I_9, I_{10} I_{11}, I_{12}	The multiplexed instruction to the Am29116.
$\overline{DROEH}$	Output enable for the upper byte of the data register for word writes to memory
$\overline{BYTEN}$	Enable for the transceiver connecting the upper and lower bytes of the data bus used to properly justify byte data being read/written to memory.

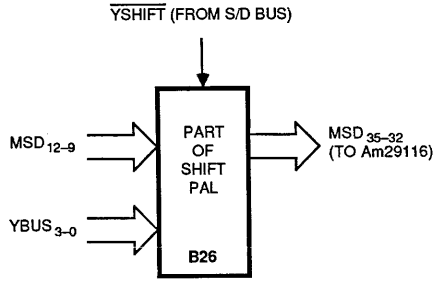


Figure 1.17 MSD Mux

```
PAL16L8
;
; MIP Processor Shift Control Pal
; shift.pal.text
;
MSD9,MSD10,MSD11,MSD12,Y0,Y1,Y2,Y3,/YSHIFT,GND,
/DROE,I9,I10,I11,I12,A0,/BYTE,/DROEH,/BYTEN,VCC
;
/I9 = /MSD9*/YSHIFT              ; shift value from microcode
    + /Y0*YSHIFT.                ; shift value from ybus
;
/I10 = /MSD10*/YSHIFT            ; shift value from microcode
     + /Y1*YSHIFT.               ; shift value from ybus
;
/I11 = /MSD11*/YSHIFT            ; shift value from microcode
     + /Y2*YSHIFT.               ; shift value from ybus
;
/I12 = /MSD12*/YSHIFT       ; shift value from microcode
     + /Y3*YSHIFT.          ; shift value from ybus
;
DROEH = DROE*/BYTE.         ; use upper byte if not byte write
;
BYTEN = BYTE*/A0.           ; enable mux to upper byte
;
END
```

Figure 1.17A SHIFT PAL Device

The Bus Interface Unit

The Bus Interface Unit, or BIU, consists of the registers and transceivers necessary to interface to the main memory and the cache (Figure 1.18). The Data Section (Figure 1.19) consists of two 16-bit bidirectional data registers, and a 16-bit instruction register, accessible one byte at a time (op-codes are one byte each). There is also a byte MUX to allow reading the high byte, or writing to the high byte (for data accesses) in a byte addressed fashion, with the resulting data being right-justified in the data register.

The Address Section (Figure 1.20) consists of a 24-bit address register and a 20-bit program counter. The address register is used with the data register for memory data transfers. The program counter is used with the instruction register to implement a pre-fetched program store. When data bytes from the instruction register are read, the Program Counter is auto-incremented.

Both the Address and Program counter may be read back to the YBUS by the CPU via readback buffers.

The Memory Section

The Memory Section (Figure 1.21) consists of a Cache Section (Figure 1.22), which runs at processor speed, and a Main Memory Section (Figure 1.23), which runs at about a 350 ns access.

A single set associative cache is used. The size is quite large: 4K words are available. A 75% hit ratio with this size cache is estimated.

The Main Memory Section consists of 128k of 16-bit words, two PAL devices for sequencing, and a Am2964B DRAM Controller.

The Cache Memory consists of 4k words of 70 ns static RAM for data, and a 4k × 4 RAM for address tags. The data RAMs are accessible by byte to support byte operations. Control circuitry is provided to supply the HIT/MISS acknowledge and allows for tag and data updating under control of the CCU. A write-thru-cache scheme is employed.

The Tag Memory is accessed for every memory operation. Four address bits (7, 9, 11, 14) form the tag data and the rest form the address to the tag and data RAMS. The tag data is compared with address bits (7, 9, 11, 14) and the HIT/MISS status is reported to the Cache Control Block. The upper address bits are used to check range validity.

If the memory cycle is a Read and there is a Hit on the cache (requested data is in the cache), then cache data is output on the DBUS and no bulk memory cycle is performed. If a Miss occurs, then a bulk memory cycle will happen. When data is available from the bulk memory, it will be written into the cache as well as being loaded into the proper data register.

A Hit on the cache memory during a Write cycle will cause the data in the cache, as well as that in the bulk memory, to be updated. If a Miss occurs, then only the bulk memory will be updated.

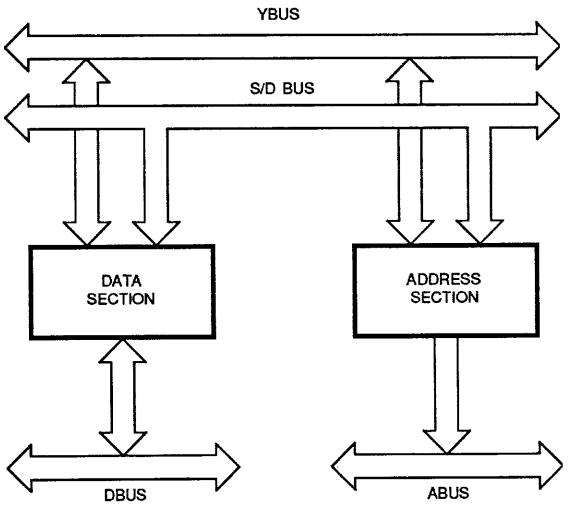


Figure 1.18 The Bus Interface Unit (BIU)

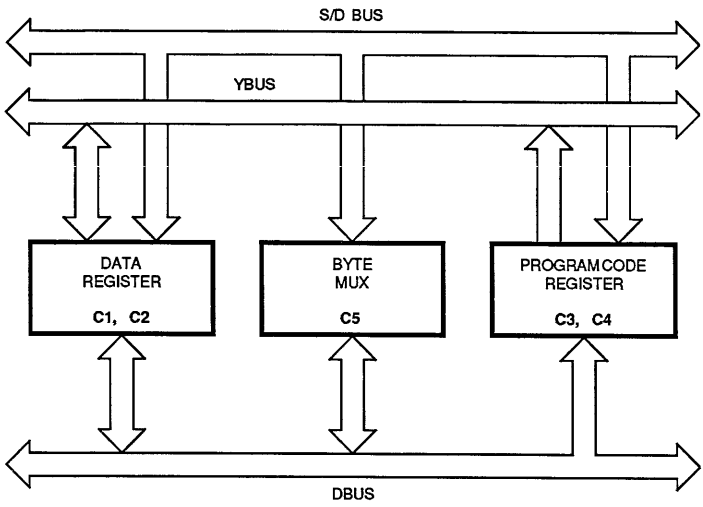


Figure 1.19 Data Section

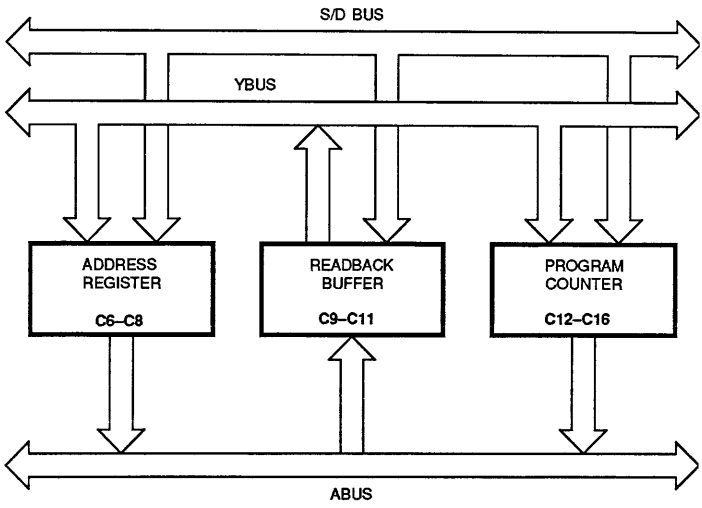


Figure 1.20 Address Section

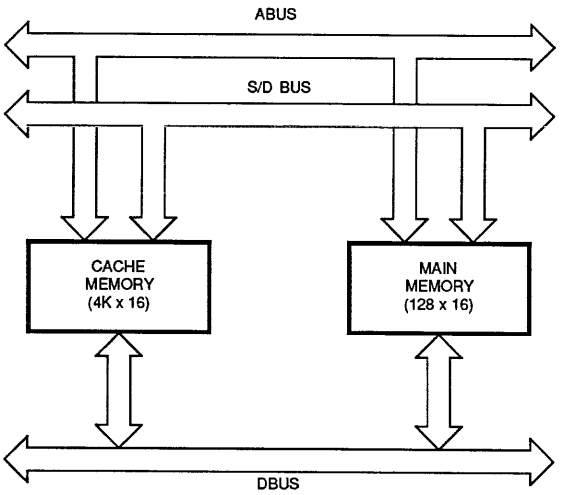


Figure 1.21 Memory Section

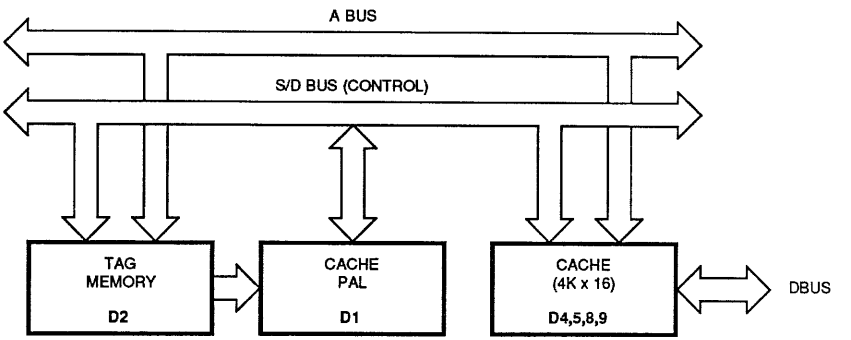


Figure 1.22 Cache Section

In order to keep cache consistency, a Read byte miss on the cache will not cause data to be written to the cache. This is due to the fact that when a byte is read from the bulk memory, only one byte is read. A byte write hit on the cache will update the correct byte as well as the main memory byte.

There is no special cache initialization logic. After power up, all that is required is to read the block of main memory which is covered by the cache memory. This makes the cache memory consistent with the main memory. Subsequent memory operations will not change this.

```
PAL16L8
;
; MIP Processor Cache Memory Control
; Cache.pal.text
;
A17,A18,A19,A20,A7,A9,A11,A14,/CWE,GND,
/DROE,/UPDT,OA11,OA9,OA7,/HIT1,OA14,/HIT,/LOCAL,VCC
;
LOCAL = /A20*/A19*/A18*/A17.              ; decode for local space
; HIT,HIT1  do the 4 bit = comparison for tag matches ( 16 terms )
;
HIT =/A7*/OA7*/A9*/OA9*/A11*/OA11* OA14* A14     */A20*/A19*/A18*/A17
    +/A7*/OA7*/A9*/OA9*/A11*/OA11*/OA14*/A14     */A20*/A19*/A18*/A17
    + HIT1*/OA14*/A14                            */A20*/A19*/A18*/A17
    + HIT1* OA14* A14                            */A20*/A19*/A18*/A17.
;
HIT1 =/A7*/OA7 */A9*/OA9 *  A11* OA11     ; do 7 of 8 terms
     + A7* OA7 */A9*/OA9 *  A11* OA11
     +/A7*/OA7 * A9* OA9 *  A11* OA11
     + A7* OA7 * A9* OA9 *  A11* OA11
     + A7* OA7 */A9*/OA9 * /A11*/OA11
     +/A7*/OA7 * A9* OA9 * /A11*/OA11
     + A7* OA7 * A9* OA9 * /A11*/OA11.
;
/OA14 = /A14                              ; when CWE enable address to output
     @CWE.
;
/OA11 = /A11                              ; when CWE enable address to output
     @CWE.
;
/OA9 = /A9                                ; when CWE enable address to output
     @CWE.
;
/OA7 = /A7                                ; when CWE enable address to output
     @CWE.
;
UPDT = /DROE*/HIT                         ; if read miss
     +  DROE* HIT                         ; if a write hit
     + CWE                                ; hold if start write
     @ /A20*/A19*/A18*/A17.               ; can update only if local access
;
; the following table indicates the cache update algorithm
; cache updates only occur on the local ram segment.
;
;                    HIT      MISS
;                  ----------------
;      Read word      nc      update
;
;      Read byte      nc       nc
;
;      Write word   update     nc
;
;      Write byte   update     nc
;
END
```

Figure 1.22A Cache PAL Device

The Cache PAL device signals have the following functions:

$\overline{\text{LOCAL}}$	Indicates that the on-board 256k of memory being acessed
$\overline{\text{HIT}}$	Indicates that a tag match has occurred
$\overline{\text{HIT1}}$	Creates some of the terms required for HIT
OA14, OA11 OA9, OA7	Data to/from the tag memory
$\overline{\text{UPDT}}$	Indicates if an update should be done to the cache memory

The Main Memory uses an Am2964B DRAM controller. Two PAL devices are required—one for sequencing (RAMSEQ) and one for decoding signals for the RAM array.

A synchronous system clock at 16 MHZ is used by the sequencer. Since this clock is synchronized to the processor clock, no exotic timing is required to get good memory response.

The RAMSEQ PAL device generates the necessary $\overline{\text{RAS}}$, $\overline{\text{CAS}}$, and $\overline{\text{MUX}}$ signals for the Am2964B DRAM controller. The refresh timer (74LS393) generates a 16 µsec. clock which is used for refresh control. The RAMSEQ PAL device does the necessary arbitration between refresh and regular memory cycles.

The RAMDCD PAL device generates some miscellaneous signals for the DRAM array and controls the Am29833 parity transceivers. Byte parity is checked in the DRAM array. A parity error will appear as an interrupt to the processor.

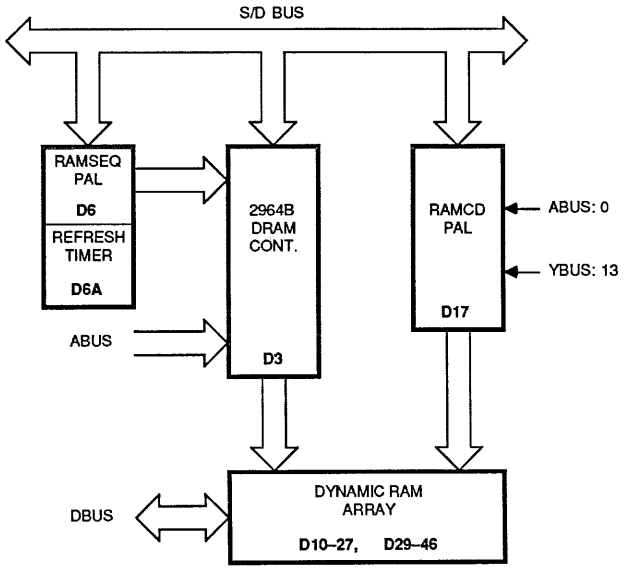


Figure 1.23 Main Memory Section

```
PAL16R6
;
; MIP Processor Ram Seq & Refresh Control
; ramseq.pal.text
;
CP16M,/XRQ,/REFCK,/RESET,NC,NC,NC,NC,NC,GND,
/EN,/DAK,NC,/REF_DNE,/RFSH,/TC,/MUX,/RAS,/CAS,VCC
;
RAS := /RAS*/MUX*/TC*/RFSH*/RESET*XRQ   ; RAS to memory controller
     +  RAS*/TC*/RESET
     +  RFSH*/RESET*/REF_DNE.
;
MUX := RAS*/MUX*/TC*/RFSH*/RESET        ; change address mux
     +  MUX*/TC*/RESET.
;
; tc ends RAS cycle and provides precharge delay between rfsh & active cycles
;
TC  := /TC*/RFSH*RAS*MUX*/RESET         ; complete ras cycle
     +  TC*/RFSH*XRQ*/RESET
     +  RFSH*RAS*/REF_DNE*/RESET        ; start tc for refresh cycle
     +  RFSH*RAS*TC*/RESET.             ; hold tc until refresh ras done
;
; rfsh active for duration of refresh cycle
;
RFSH := /RAS*/MUX*/TC*/RFSH*/REF_DNE*/RESET*/XRQ*REFCK  ; refresh in progress
      + RFSH*/REF_DNE*/RESET
      + RFSH*REF_DNE*RAS.               ; keep rfsh until ras done
;
; indicates that refresh cycle done for this refck cycle
;
REF_DNE := /REF_DNE*RFSH*TC*/RESET      ; refresh for this refck has been done
         +  REF_DNE*REFCK*/RESET        ; until refck goes low
         +  REF_DNE*RAS*/RESET          ; remainder of refresh cycle
         +  REF_DNE*TC*/RESET
         +  REF_DNE*RFSH*/RESET.
;
CAS = MUX*/RESET                        ; CAS for memory
    + CAS*XRQ*/RESET.
;
DAK = XRQ*TC*/RAS*CAS*/RESET            ; dtack for memory
    @ XRQ*TC*/RAS*CAS*/RESET.
;
```

Figure 1.23A RAMSEQ PAL Device

For RAMSEQ PAL, the signals have the following functions:

$\overline{RAS}$	Ras timing signal to the Am2964 controller
$\overline{MUX}$	Mux timing signal to the Am2964 controller
TC	Indicates that the RAM cycle is complete
$\overline{RFSH}$	Indicates a refresh cycle is in progress
REF DNE	Indicates that the requested refresh has been done
$\overline{CAS}$	CAS timing signal to the Am2964 controller
$\overline{DAK}$	Data acknowledge for the dynamic RAM cycle

The equations for 'RAMDCD' PAL device are shown in Figure 1.23B.

```
PAL16L8
;
; MIP Processor error control & misc decode
; ramdcd.pal.text for 29833's
;
/WE,NC,/XWAIT,/BYTE,/XRQ,NC,A0,/CAS,YBUS13,GND,
/LDIOA,/WEL,/REU,/WEH,/REL,NC,/CLRERR,/WAIT,/SWAIT,VCC
;
WEL = WE*XRQ*A0*BYTE              ; WE to lower ram bank
    + WE*XRQ*/BYTE.
;
WEH = WE*XRQ*/A0*BYTE             ; WE to upper ram bank
    + WE*XRQ*/BYTE.
;
REU  = CAS*XRQ*/A0*BYTE*/WE       ; for lower byte of mem
     + CAS*XRQ*/BYTE*/WE.
;
REL  = CAS*XRQ*A0*BYTE*/WE        ; upper byte of mem
     + CAS*XRQ*/BYTE*/WE.
;
CLRERR = LDIOA*YBUS13.            ; parity error clear
;
SWAIT = WAIT + XWAIT.             ; allow external waits
;
END
```

Figure 1.23B RAMDCD PAL Device

The RAMDCD PAL signals have the following functions:

$\overline{WEL}$, $\overline{WEH}$	Write enables for the upper and lower RAM bytes
$\overline{REH}$, $\overline{REL}$	Output enables for the Am29833 transceivers
$\overline{CLRERR}$	Clears the parity latch on the Am29833
$\overline{SWAIT}$	Combines a couple of WAIT signals

I/O Unit

The I/O unit is shown in Figure 1.24. All registers and buffers are currently 8 bits wide. The I/O unit interfaces with the peripheral processor which does all of the I/O work. There are buffers in one direction and registers in the other direction so the transfer path can be pipelined (i.e. the MIP processor does not have to wait for a message to be read).

There is also an 8-bit register for control signals and an 8-bit buffer for status signals from this interface. These are used for signalling and synchronization of the peripheral devices.

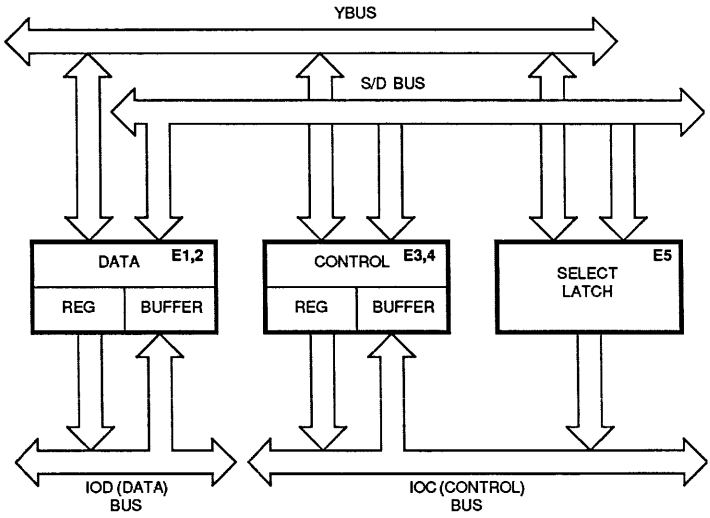


Figure 1-24 I/O Unit

Chapter 2

SOFTWARE DESCRIPTION

2.0 OVERVIEW

The MIP processor is designed to be used in several different environments requiring high-speed processing capability. The main advantage of the processor is that the instruction set can be tailored to the target system, thus achieving near ideal execution efficiency.

The first implementation given here is an instruction processor for PASCAL. The Pascal compiler emits intermediate code called P-code. This code is based on the concept of a stack processor and is designed to be compact and is reasonably easy to generate from a high-level language.

The P-code is executed by an interpreter written in micro-code. This intermediate code thus becomes the instruction set of the processor; a significant speed advantage is obtained over other types of processors. Intrinsics and other special purpose routines can be written in micro-code and linked with the intermediate code. Changes or upgrades in the machine hardware only mean minor modifications to the micro-code interpreter, not wholesale changes to a compiler. It is also very easy to port such a system to a number of different processors.

The task of writing an interpreter for such a machine is not difficult but does require a good set of tools, such as assemblers, and trace and debug utilities. The authors have developed a set of tools which, when used in conjunction with the diagnostic pipeline registers, give a unified approach to micro-code development. These tools will work with various personal computers which have Pascal capability.

As this processor is also designed to occupy minimum real estate, a number of trade-offs were employed so as to fit the processor in a small space yet retain performance.

The micro-code word is designed to be 32 bits in length. This is generally regarded as being short, but good performance is still possible. By using a short word and eliminating some of the hardware surrounding micro-programmed machines, it is possible to make a processor which is competitive with traditional NMOS/CMOS processors, when compared on the basis of processing power vs. board real estate. Given twice the NMOS equivalent of real estate, one can end up with a machine having 8 to 10 times the performance of an NMOS machine. Interestingly enough, the comparison is also relevant when comparing processing power vs. cost (i.e., 10 times the performance at twice the cost).

The use of overlapped fields for the Am29116 and the jump address field allow the micro-code to fit in 32 bits. The cost is about 3–4% in total performance because not all op-codes use Jump instructions and those that do are usually limited by the memory bandwidth.

The traditional instruction mapping PROMs were replaced by a table mapping technique which keeps the vector table in the same code space as the main micro-code. Such a techinque costs a bit of execution time (approx 5–6%) but saves considerable power and space. It is also useful where the micro-code is constantly being modified as the instruction mapping table is now created and loaded with the micro-code.

Although the processor is compact, a number of pipe-line stages, which can be used by the micro-code, exist within the processor. These increase the utilization of each component of the processor and, in turn, increase performance.

The first major pipe-line stage is the memory interface. Here, either an instruction fetch or data store operation can be performed in parallel with Am29116 instruction execution. As long as there is no conflict (i.e., using a value from a Read Data operation before the read completes), the micro-code will not experience any Wait cycles. If there is a conflict, the processor will pause automatically until the conflict is resolved. This is the most frequently used pipe-line stage, as there is often a number of micro-instructions to be executed while data is being accessed. The instruction pre-fetch mechanism is also part of the same pipe-line. While the processor is using the last data value from the instruction register, new data is being read using the program counter.

The next important stage is between the Am29116 and the YBUS. Data may be operated on, inside the Am29116, while other data is being moved from source to destination on the YBUS. A temporary holding register on the YBUS may be used to delay or duplicate a data transfer without involving the Am29116.

A third pipe-line stage exists with the Micro-program Sequencer. It is possible to be utilizing all of the previous pipe-line stages while performing certain types of micro-program loop or subroutine returns.

All of these stages are used to allow efficient micro-program execution and help to offset some of the trade-offs that were made to compact the micro-code into 32 bits.

A mechanism is included to allow data values, which

appear on the YBUS, to be used in the bit-oriented instructions of the Am29116 as part of the instruction. This allows dynamic bit-oriented instructions to be created. Packed field, Set operations and Floating-point routines use this class of instruction frequently.

Computed Jumps and Calls are also possible by placing the desired data on the YBUS and doing a Jump or Call with the Micro-program Sequencer. These have the effect of reducing the overall code size and also help to improve performance by eliminating costly chains of Test and Jump instructions.

2.1 MICRO-CODE DEVELOPMENT

Several micro-code assemblers are available for code development. These are usually more difficult to use than the assemblers that are used for standard micro-processor work. Several "meta" assemblers are available to do microcode development. These are more difficult to work with than normal assemblers because they do not reflect field relationships in the target machine. Complexity in the assembler syntax due to architectural parallelism is forgivable. Less forgivable is the complexity that arises when each field of the machine language must be independently specified.

As this project required a lot of micro-code development, some effort was made to streamline the code creation process. Inspection of the published Am29116 instruction set shows some redundant information which is handled by the improved assembler. This makes the resulting source code much easier to write and debug. Features which are unique to the machine may be included as optional parameters, separated by commas, after the main instruction code. Examples of this micro-code format are shown in Appendix A. Streamlining of the micro-instructions also make it easier to upgrade the machine to larger word sizes and add new features when required.

The micro-code word for this processor is 32 bits long, which is relatively short for a micro-coded machine. There is one overlapped 16-bit instruction field, shared between the Am29116 instruction and the Jump address field. This means that there are two instruction formats. The first is a data type instruction and will involve some action by the Am29116 and optional operations by other data elements of the processor. The second instruction type is a Control instruction. Here, micro-instruction control will conditionally change (eg. JUMP, CALL, RTS). Each micro-instruction has 6 other fields which occupy the remaining 16 bits micro-store width. They are used to specify data path (YBUS) source and destination, memory control, sequencer control, and flag register updates. All of the fields must be defined for each word. To make the task of writing micro-code easier, the micro-assembler uses a default micro-instruction word. This instruction is initialized at the beginning of each source line and then modified by operands on the source line. If written out, the default control word would appear as:

NOOP CONT, SRE, IE, NYS, NYD, NSR

This sequence specifies a NOP instruction to the Am29116, a continue to the Am2910 sequencer, status flag update, Am29116 instruction enable, no YBUS destination, no YBUS source, and no store control operation.

Any name which effects control over the YBUS, store control, micro-sequencer, or status flag updates, may be entered in free field form after all the required operands for the main instruction. These will be denoted optional parameters in the following descriptions.

Any Am29116 instruction mnemonic which contains an 'I' requires an immediate data operand following the Am29116 instruction descriptor.

The micro-assembler performs a number of syntax checks on the generated code to detect invalid instruction combinations, illegal instruction sequences, and missing operands.

2.2 OP-CODE EXECUTION

Creation of an interpreter for a micro-coded intermediate code machine is quite straightforward as there are several good descriptions of the intermediate object code in print.

The basic concept of a intermediate code processor is that of a stack machine. A number of registers point to various constructs of this stack machine such as local and global data frames, the heap and the stack. The various op-codes move data values between these frames and the stack, and operate on stack elements.

All the registers in the hypothetical stack machine are contained within the register file of the Am29116, leaving 24 work registers for op-codes and intrinsic procedures to use.

This processor has no dedicated hardware to maintain the stack. This may appear to be a short-coming of the processor, but it is not serious. The presence of the cache memory means that most of the stack elements are available without a Wait, as they are usually the most frequently referenced items. To reduce the number of memory references to the top of stack element, it is kept within the register file. This also allows a form of pipe-lining to be done during op-code execution. A side effect is that stack reads and writes now get done during a different portion of the execution of the op-code. This reduces the peak rate of demand on the memory system.

The instruction stream for the intermediate code is byte-oriented. Within the processor is a two-byte instruction prefetch queue, filled on demand by the Bus Sequencer. The micro-code is arranged so that there is an instruction jump table starting at F00 (hex). As there are 256 op-codes, this table occupies the last 256 bytes of memory. During an op-code fetch, the instruction byte pointed to by the program counter is enabled onto the lower byte of the YBUS. The high byte is forced to 1's by pull-up resistors. The microprogram sequencer is told to do an unconditional jump to the address on the YBUS and so ends up in the instruction vector table. This table is comprised of jumps to the individual op-code routines. A high percentage of instructions only use one or two bytes of instruction so the pre-fetch queue works quite well.

In the event of an interrupt, the next byte of the instruction queue is not enabled onto the YBUS. Instead, bit 8 of the YBUS is forced Low. The resulting Jump is to FEF (hex) which contains a Jump to the interrupt sevice routine. (YBUS bit 8 is tied to $\overline{\text{IVECT}}$ in REGCTL PAL.)

Several examples of micro-code op-codes are shown in Appendix A. The total amount of micro-code for all op-codes and intrinsics is approximately 2.5 k words. With a 4 k micro-store, this leaves sufficient room for new op-codes and intrinsic procedures.

On average each op-codes takes about 8 micro-code instructions, one of which is the jump to next instruction. Some subroutines are used to keep the code compact.

2.3 MICRO-CODE DEBUG TOOLS

Normally, to debug the hardware and micro-code of a machine such as this would require a special development station. The use of diagnostic registers on critical parts of the MIP processor allows the use of much simpler hardware. A small personal computer such as an APPLE II or a PC, with Pascal language capability can be programmed to act as a development station or debug tool. Currently a 68000 based system is being used.

Access to the MIP processor diagnostic connector requires a TTL level I/O port with 9 output pins and 1 input pin. The output bits could just be registered, although faster port operation would occur if 7 were registered and 2 were pulse outputs (DCLK and STEP).

A Pascal program executing on the workstation provides access to all of the processor registers, the writeable control store, and the main memory. This same workstation is used to edit and assemble the micro-code for the processor under development.

The debug tool is menu driven. The display normally shows all the registers of the processor. When a command is entered, the action is performed and the display updated. This is suitable for single-stepping through micro-code or changing register values. Branches to new sections of code can also be done.

Loading of the writeable control store is done by specifying an object file (which is located on the workstation disk) to be loaded into the writable control store.

Code and data files may also be loaded into the main memory in a similar manner. Commands exist to display a block of 128 main memory bytes at once, as well as change single bytes or words.

Breakpoints may be specified for the micro-code. Execution of the code will progress until the breakpoint is encountered and the display will be updated. This mode of execution is not done in real time, however, as the workstation checks the micro-address after each instruction. This mode is quite useful for debugging most codes. Other techniques such as scope loops and computation loops can be easily implemented to isolate timing problems.

2.4 MICROCODE DETAILS

Microcode field definitions for SMIP

The 32-bit microword is divided up into fields as shown below:

MSD31–28	Am2910A sequencer control
MSD27–25	memory control
MSD24	Am29116 $\overline{\text{IEN}}$
MSD23	Am29116 $\overline{\text{SRE}}$
MSD22–20	YBUS source
MSD19–16	YBUS destination
MSD15–0	Am29116 instruction or immediate data

```
;
; single operand format
;
MOVE    SORA     ; ram to acc       Ram Name<,optional parms>
COMP    SORY     ; ram to y
INC     SORS     ; ram to status
NEG     SOAR     ; acc to ram
        SODR     ; d to ram
        SOIR     ; id to ram
        SOZR     ; 0 to ram
        SOZER    ; d(oe) to ram
        SOSER    ; d(se) to ram
        SORR     ; ram to ram
        SOA      ; acc -> ?         NRY   ; ybus        <,optional parms>
        SOD      ; d   -> ?         NRA   ; acc
        SOI      ; i   -> ?         NRS   ; status
        SOZ      ; 0   -> ?         NRAS  ; acc,status
        SOZE     ; d(oe)
        SOSE     ; d(se) ;
; two operand instructions          R     S     D
;
SUBR    ; s - r         TORAA      ; ram acc   acc     Ram Name<,optional parms>
SUBRC   ; s - r - c     TORIA      ; ram  i    acc
SUBS    ; r - s         TODRA      ; d    ram  acc
SUBSC   ; r - s - c     TORAY      ; ram  acc  y
ADD     ; r + s         TORIY      ; ram  i    y
ADDC    ; r + s + c     TODRY      ; d    ram  y
AND     ; r and s       TORAR      ; ram  acc  ram
NAND    ; r nand s      TORIR      ; ram  i    ram
EXOR    ; r xor s       TODRR      ; d    ram  ram
NOR     ; r nor s       TODAR      ; d    acc  ram
OR      ; r or s        TOAIR      ; acc  i    ram
EXNOR   ; r xnor s      TODIR      ; d    i    ram
                        TODA       ; d    acc  ?
                        TOAI       ; acc  i    ?
                        TODI       ; d    i    ?
;
; single bit shifts                 u     d
;
SHUPZ   ; up 0          SHRR       ; ram  ram          Ram Name<,optionalparms>
SHUP1   ; up 1          SHDR       ; d    ram                  "
SHUPL   ; up qlink
SHDNZ   ; down 0
SHDN1   ; down 1        SHA        ; acc               NRY or NRA<,optional parms>
SHDNL   ; down qlink    SHD        ; d                          "
SHDNC   ; down qc
SHDNOV  ; down qn xor qovr
;
; bit oriented instructions
;
SETNR   ; set ram bit n (n*512)                     Bit#,Ram Name<,optional parms>
RSTNR   ; reset ram bit n
TSTNR   ; test ram bit n
LD2NR   ; 2**n -> ram (n*512)
LDC2NR  ; comp(2**n) -> ram
A2NR    ; ram + 2**n -> ram
S2NR    ; ram - 2**n -> ram
TSTNA   ; test acc bit n
RSTNA   ; reset acc bit n
SETNA   ; set acc bit n
A2NA    ; acc + 2**n -> acc
```

```
S2NA    ; acc - 2**n -> acc
LD2NA   ; 2**n -> acc
LDC2NA  ; comp(2**n) -> acc
TSTND   ; test d bit n
RSTND   ; reset d bit n
SETND   ; set d bit n
;
A2NDY   ; d + 2**n -> y                                  Bit#,<optional parms>
S2NDY   ; d - 2**n -> y                                          "
LD2NY   ; 2**n -> y                                              "
LDC2NY  ; comp(2**n) -> y                                        "
;
; rotate by n bits
;       u         d
RTRA    ; ram     acc                                    Bit#,Ram Name<,optional parms>
RTRY    ; ram     y                                              "
RTRR    ; ram     ram                                            "
RTAR    ; acc     ram                                            "
RTDR    ; d       ram                                            "
;
RTDY    ; d       y                                      Bit#<,optional parms>
RTDA    ; d       acc                                            "
RTAY    ; acc     y                                              "
RTAA    ; acc     acc                                            "
;
; rotate and merge
;       u         r/d   s
MDAI    ; d       acc   i          Bit#,Immediate Data<,optional parms>
MDAR    ; d       acc ram          Bit#,Register name<,optional parms>
MDRI    ; d       ram   i          Bit#,Register name,Immediate Data<,optional parms>
MDRA    ; d       ram   acc        Bit#,Register name,<,optional parms>
MARI    ; acc     ram   i          Bit#,Register name,Immediate Data<,optional parms>
MRAI    ; ram     acc   i          Bit#,Register name,Immediate Data<,optional parms>
;
; rotate and compare
;       u         r     s
CDAI    ; d       acc   i          Bit#,Immediate Data<,optional parms>
CDRI    ; d       ram   i          Bit#,Register name,Immediate Data<,optional parms>
CDRA    ; d       ram   acc        Bit#,Register name<,optional parms>
CRAI    ; ram     acc   i          Bit#,Register name,Immediate Data<,optional parms>
;
; crc instruction
;
CRCF    ; crc forward              Register Name<,optional parms>
CRCR    ; crc reverse              Register Name<,optional parms>
;
; status bit instructions
;
SETST   ; set bit       ONCZ       ; OVR,N,C,Z<,optional parms>
RSTST   ; reset bit     L          ; link
                        F1         ; f1
                        F2         ; f2
                        F3         ; f3
;
SVSTR   ; save status in ram       Register name<,optional parms>
SVSTNR  ; save status in NRY       <optional parms>
;
; conditional jumps and calls
;
JUMP    ; cond jump    Condition,address<,optional parms>
CALL    ; cond call    Condition,address<,optional parms>
CRET    ; cond return  Condition<,optional parms>
;
; condition codes for Jump,Call & CRET
```

```
;
CT      ; latched CT from 29116
Z       ; latched Zero flag
C       ; latched Carry flag
N       ; latched Sign
OVR     ; latched Overflow
UNC     ; unconditional
NCT     ; latched /CT
NZ      ; latched not Zero
NC      ; latched not Carry
P       ; latched Positive
NOVR    ; latched not Overflow
;
NOOP    ; nop                          <optional parms>
;
; Test condition instructions
;
TNOZ    ; (N xor OVR) + Z              <optional parms>
TNO     ; (N xor OVR)
TZ      ; Zero
TOVR    ; OVR
TLOW    ; low
TC      ; C
TZC     ; Z + /C
TN      ; N
TL      ; link
TF1     ; f1
TF2     ; f2
TF3     ; f3
;
; prioritize
;
; PRTXYZ <ram name><,Immediate data><,optional parameters>
; where x = source
;       y = mask
;       z = destination
;
;       R = ram
;       A = accumulator
;       D = d inputs
;       I = immediate data
;       Z = zero
;       NR = no ram destination
;
PRTARA
PRTARY
PRTARR
PRTRAA
PRTRZA
PRTRIA
PRTRAY
PRTRZY
PRTRIY
PRTRAR
PRTRZR
PRTRIR
PRTAAR
PRTAZR
PRTAIR
PRTDAR
PRTDZR
PRTDIR

PRTNRAA
```

```
PRTNRAZ
PRTNRAI
PRTNRDA
PRTNRDZ
PRTNRDI
;
; 29116 Internal Register names     { internal RAM on 29116 }
;
D0      ; 0       - D0
D1      ; 1       - D1
D2      ; 2       - D2
D3      ; 3       - D3
D4      ; 4       - D4
D5      ; 5       - D5
D6      ; 6       - D6
D7      ; 7       - D7
MP      ; 8       - MP  local pointer
BP      ; 9       - BP  base pointer
NP      ; 10      - NP  heap pointer
SP      ; 11      - SP  stack pointer
IPC     ; 12      - IPC temporary program counter
SEGP    ; 13      - SEGP segment pointer
JTAB    ; 14      - JTAB proc pointer
TOS     ; 15      - top of stack element
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15
;
; The following field names are optional in any micro-insruction
;       and are denoted by <optional parms> in the above list
;
;       Sequencer control sc[0..3]
;
JZ      ; jump to address 0
CJS     ; conditional jsr via PL
JMAP    ; jump to address via MAP
CJP     ; jump to address via PL
PUSH    ; push stack and cond load counter
JSRP    ; jsr via R or PL
CJV     ; cond jump to VECT
JRP     ; jump to R or PL
RFCT    ; repeat loop if CT <> 0
RPCT    ; repeat PL if CT <> 0
CRTS    ; conditional return
CJPP    ; conditional jump to PL and pop stack
LDCT    ; load CT
LOOP    ; test end of loop
CONT    ; continue
TWB     ; three way branch
;
; YBUS sources
```

```
;
OEY     ; 29116 output enable Y
DRS     ; data register
YREG    ; YBUS data capture reg
IOD     ; I/O data bus
IR      ; instruction reg
OEPM    ; 29517 msp product enable
OEPL    ; 29517 lsp product enable
NYS     ; no Ybus source
;
; YBUS destinations
;
IOC     ; I/O control
DR      ; data register
PC      ; program counter
AR      ; address register
DLE     ; 29116 data latch
PG      ; address page
PCPG    ; program status word
FMT     ; formats reg
CTEN    ; condition code enable
ENY     ; multiplier y input
ENX     ; multiplier x input
ICR     ; interrupt control dest
ABEN    ; address reg readback
PGEN    ; page reg readback
REG     ; 2910 register/counter
YSHFT   ; enable dynamic bit operations
NYD     ; no Ybus destination
;
; Memory Control codes
;
NSR     ; 0 - no store request
RPS     ; read program store
CRPS    ; cond read program store
WIO     ; wait for memory io to finish
RDS     ; read data store
WDS     ; write data store
RDSB    ; read data store byte
WDSB    ; write data store byte
;
; Status & Instruction control
;
NSE     ; status load disable
NIE     ; 29116 instruction disable
SRE     ; status update enable
```

Chapter 3

PERFORMANCE

The primary objective, when building the MIP Processor, was to achieve an order of magnitude of increase, in performance, over a standard workstation (68000 based) in the execution of HLL (High Level Language) benchmarks; this is all based on the intermediate code concept of PASCAL.

So far (July/85), a performance advantage of 8.25 in a linear weighting of op-codes has been observed. Most op-codes see a ten-fold, or better, advantage, therefore, with some fine-tuning of certain op-codes, an advantage of 10 is expected.

A cache hit rate of 75% was estimated and measurements showed a hit rate of 71% on the linear weighting test.

3.0 PRELIMINARY SURVEY

A number of benchmarks are available for performance evaluation.

The most notable one is the BYTE Sieve benchmark. A large body of data has been collected for this test. It mainly tests array accessing and logic test capability of the machine. It is included here because it is easy to do and every one else does it.

A linear weighting of op-codes is quite useful for comparing the performances of two machines using the same upper level software. It allows for a quantitative measure of specific features of the new machine. When combined with the op-code run time frequency of occurrence, the specific benefit of a new feature can be evaluated.

Because the machine is intended to be used in a workstation environment, compilation speed is also an important bench mark.

From time to time a workstation processor will perform a number of operations which are based on numerical algorithms. Of a large class of signal processing and statistical routines, the FFT is representative.

Another often quoted benchmark is the Whetstone benchmark. It uses floating point arithmetic quite heavily. At this point the floating point micro-code has yet to be completed, therefore, this benchmark test has not been performed.

In the following discussions of the benchmarks, all the timings were measured by using the self timing capabilities of the machines. The machines all have access to a real time clock with a resolution of 16.6 ms. Sufficient loops of each test were ran to bring the timing resolution to within 1 ms.

3.1 COMPILATION SPEED

The compilation speed of the 68000 used is in the range of 900 to 1500 lines per minute. A typical value for compiling the compiler is 1150 lines per minute; the MIP processor can do this 6.5 times faster (7500 lpm).

3.2 SIEVE OF EROSTHANES

The 68000 bench ran at 78 seconds, which reduced to 64 seconds, at best if Wait States are removed. The MIP sieve ran at 5.6 seconds. If Wait States are removed from the MIP, better performance can be expected.

Recently, 'streamlining' the Sieve bench has been done to take advantage of certain particular environments. There was one 'improved' version written in C that takes advantage of register coercion. The authors have no objection to that, but readers should consider the following: If the Sieve is rewritten as a microcode routine in the MIP, then the following program is the benchmark:

```
Begin
   Sieve
end;
```

This program runs in approximately .14 seconds.

Based on information in the BYTE article, an 68000 assembly language version had a performance time of 1.12 seconds (which was the fastest time quoted of any example). A microcoded sieve on the MIP still runs 8 times faster.

This illustrates the power of a microcoded approach. In practice, program bottlenecks are moved into microcode as they are encountered. Quite often significant system performance enhancements can be made by the addition of a few small intrinsic functions.

3.3 FFT

A micro-coded FFT routine was also created. It shows a dramatic increase over a similar program written for the 68000. There is a tendency to compare the micro-coded machine with special purpose FFT processors. This machine does not have the dual ALU's, dual memory banks, and dedicated address and coefficient ROMs normally associated with such a machine, so, as expected, it is about 6 to 8 times slower than an FFT

processor. As a general purpose processor, it does do a 1K complex FFT in 35 ms. This is a factor of 15 better than the 68000.

3.4 ANALYSIS AND CONCLUSIONS

The MIP processor has proven to be an effective vehicle for demonstrating the power and flexibility of a micro-coded machine. The P-code system has an overall performance improvement of 6 to 7, over a 68000 based system. While this is not as great as originally hoped, it remains a significant amount.

Micro-coded intrinsic functions do experience a greater performance improvement factor than a 68000 assembly intrinsics. This comparative speed-up is due to the instruction set of the Am29116 and the pipeline stages of the processor.

The small single set cache memory used is responsible for a 23% performance improvement when executing P-code programs. The hit rate averages 71%. These data are consistent with other single set cache memories that have been used on other processors and reported in the literature.

The processor is executing P-code with an effective processor utilization of 73%. Memory bus utilization is approximately 44%. Obviously a good system improvement could be made if these figures were closer to 100%. There are a couple of ways to do this. As each P-code is 'tuned', the number of processor cycles is reduced. Careful inspection of a P-code often allows the memory references to be arranged so as not to cause Wait States to occur; this improves processor utilization. Reducing the number of processor cycles per op-code increases memory utilization. This is because the number of memory references per P-code is constant.

Minor changes to the definition of the P-codes can also increase system performance. This was not done because that part of the system was kept constant to compare with other machines.

Probably the single largest improvement on the processor would be a more complex BIU. Such a unit would have a dedicated stack address and data register. It was not feasible to include this in the original design due to a limitation of board real estate. There are a number of multi-port register files (5 or 6 port) available now which would allow the entire BIU to be reduced in size with increased functionality. This could be done without increasing the micro-code width. Some machines have added a dedicated stack area. This limits the stack to a fixed size and location which can cause problems. With a stack address and data register working into a cache based memory system, the delays due to a quantity not being available in the stack register are minimal. The processor would have access to the top two items of the stack with no delay at all (TOS inside the Am29116, TOS-1 in the stack data register).

The 2 byte pre-fetch mechanism appears to work well. The average P-code is less than 2 bytes. If the bus utilization were very high (>90%), it may be necessary to have more than two bytes pre-fetched so as to minimize op-code waits within a P-code.

As a single board processor, this design is very effective from a performance point of view. The standard functionality offered by a 68000, or other such processor, is available, with the added ability to have micro-coded intrinsic functions. The effort to create a micro-coded intrinsic function is the same as writing assembly level routines for a 68000 (in fact, it is often easier due to the diverse nature of the Am29116 micro-code instructions).

The 4k size of the micro-store is adequate to allow the coding of a high-level intermediate code such as P-code and allows ample room (1.5k) for intrinsic functions.

The technique of using the 29818 diagnostic registers for trace and debug of the processor is effective. A program written in Pascal performs all functions required to initialize, load, and test the processor. This diagnostic program can be transported to virtually any workstation. A simple port gives access to the processor under test.

Appendix A

Sample Micro-Code

```
0017|                    SLDLX  STACKW  TOS                          ; load ith local word to stack
0017| EE81 D850         #      MOVE    SORY,TOS,OEY,DR,NSE,WIO
0018| EC83 C3EB         #      S2NR    1,SP,OEY,AR,WDS,NSE
0019| E07F E481 E07F FF2E      ADD     TOAI,NRA,<<MSLCL/2>-215> ; base value + offset
001B| E00F EC01               SHUPZ   SHA,NRA,OEY              ; * 2
001C| EA03 9088               ADD     TORAY,MP,OEY,AR,RDS      ; add mp and do read
001D| E01F D8D0               MOVE    SODR,TOS,DRS             ; wait for read to complete
001E|                         IFETCH
001E| 244F F921         #      MOVE    SOZE,NRA,IR,CRPS,JMAP

004F|                     LDO   STACKW  TOS                          ; load local with offset B
004F| EE81 D850         #      MOVE    SORY,TOS,OEY,DR,NSE,WIO
0050| EC83 C3EB         #      S2NR    1,SP,OEY,AR,WDS,NSE
0051|                         GBIG
0051| E44F D946         #      MOVE    SOSER,D6,IR,CRPS
0052| 61F8 ****         #      JUMP    P,$1
0053| E07F 91E6         #      RTRR    8,D6
0054| E07F FFC6         #      RSTNR   15,D6
0055| E44F 58C6         #      MOVE.B  SODR,D6,IR,CRPS
0056|                   #$1
0056| E07F 8486 E07F 0005      ADDTORIA,D6,<MSLCL/2>
0058| E07F EC01               SHUPZ   SHA,NRA                  ; * 2
0059| EA03 9089               ADD     TORAY,BP,OEY,AR,RDS      ; add bp and do read
005A| E01F D8D0               MOVE    SODR,TOS,DRS
005B|                         IFETCH
005B| 244F F921         #      MOVE    SOZE,NRA,IR,CRPS,JMAP

005C|                     LAO   STACKW  TOS                          ; load address of B'th local
005C| EE81 D850         #      MOVE    SORY,TOS,OEY,DR,NSE,WIO
005D| EC83 C3EB         #      S2NR    1,SP,OEY,AR,WDS,NSE
005E| GBIG
005E| E44F D946         #      MOVE    SOSER,D6,IR,CRPS
005F| 61F8 ****         #      JUMP    P,$1
0060| E07F 91E6         #      RTRR    8,D6
0061| E07F FFC6         #      RSTNR   15,D6
0062| E44F 58C6         #      MOVE.B  SODR,D6,IR,CRPS
0063|                   #$1
0063| E07F 8486 E07F 0005      ADD     TORIA,D6,<MSLCL/2>
0065| E07F EC01               SHUPZ   SHA,NRA                  ; * 2
0066| E00F 8089               ADD     TORAA,BP,OEY
0067| E00F D890               MOVE    SOAR,TOS,OEY
0068|                         IFETCH
0068| 244F F921         #      MOVE    SOZE,NRA,IR,CRPS,JMAP

009C|                     LDC   STACKW  TOS
009C| EE81 D850         #      MOVE    SORY,TOS,OEY,DR,NSE,WIO
009D| EC83 C3EB         #      S2NR    1,SP,OEY,AR,WDS,NSE
009E| E44F D931               MOVE    SOZER,R1,IR,CRPS         ; get length of block
009F| EE7F 7140               NOOP    WIO
00A0| E07C 7140               NOOP    ABEN
00A1| E07C E190               TSTND   0,ABEN                   ; test LSB of PC
00A2| 61F8 ****               JUMP    Z,$1                     ; if word aligned
00A3| E44F 7140               NOOP    IR,CRPS                  ; dump odd byte
00A4| E44F D920          $1   MOVE    SOZER,D0,IR,CRPS         ; may want flip
00A5| E07F 91E0               RTRR    8,D0
00A6| E44F 58C0               MOVE.B  SODR,D0,IR,CRPS
00A7|                         STACKW  D0                       ; move 1 word
```

```
00A7| EE81 D840          #       MOVE    SORY,D0,OEY,DR,NSE,WIO
00A8| EC83 C3EB          #       S2NR    1,SP,OEY,AR,WDS,NSE
00A9| E07F C1F1                  S2NR    0,R1
00AA| 61F8 9F5B                  JUMP    NZ,$1                   ; loop for all
00AB|                            STACKR
00AB| EA83 D96B          #       MOVE    SORR,SP,OEY,AR,NSE,RDS
00AC| E0FF C3CB          #       A2NR    1,SP,NSE
00AD| E01F D8D0                  MOVE    SODR,TOS,DRS
00AE|                            IFETCH
00AE| 244F F921          #       MOVE    SOZE,NRA,IR,CRPS,JMAP

00CB|                     STO    STACKR  ; read address
00CB| EA83 D96B          #       MOVE    SORR,SP,OEY,AR,NSE,RDS
00CC| E0FF C3CB          #       A2NR    1,SP,NSE
00CD| EE01 D970                  MOVE    SORR,TOS,OEY,DR         ; data
00CE| EC13 7140                  NOOP    DRS,DR,AR,WDS           ; write it
00CF|                            STACKR
00CF| EA83 D96B          #       MOVE    SORR,SP,OEY,AR,NSE,RDS
00D0| E0FF C3CB          #       A2NR    1,SP,NSE
00D1| E01F D8D0                  MOVE    SODR,TOS,DRS
00D2|                            IFETCH
00D2| 244F F921          #       MOVE    SOZE,NRA,IR,CRPS,JMAP

00D3| E00F E441 E00F 00F8 SINDX  SUBS    TOAI,NRA,248,OEY        ; adjust offset
00D5| E00F EC01                  SHUPZ   SHA,NRA,OEY
00D6| EA03 9090                  ADD     TORAY,TOS,OEY,AR,RDS    ; get the data
00D7| E01F D8D0                  MOVE    SODR,TOS,DRS
00D8|                            IFETCH
00D8| 244F F921          #       MOVE    SOZE,NRA,IR,CRPS,JMAP

00F1|                     IXA    STACKR  ; get base
00F1| EA83 D96B          #       MOVE    SORR,SP,OEY,AR,NSE,RDS
00F2| E0FF C3CB          #       A2NR    1,SP,NSE
00F3| 11F8 7FED                  CALL    UNC,GETBIG              ; get element size
00F4| E00A CC06                  SHUPZ   SHRR,D6,OEY,ENX         ; * 2 to mult
00F5| E009 D970                  MOVE    SORR,TOS,OEY,ENY        ; * element size
00F6| E00F 7140                  NOOP    OEY
00F7| E06F F8C1                  MOVE    SOD,NRA,OEPL
00F8| E01F C290                  ADD     TODAR,TOS,DRS           ; + base => tos
00F9|                            IFETCH
00F9| 244F F921          #       MOVE    SOZE,NRA,IR,CRPS,JMAP

018D|                     ADI    STACKR                          ; add TOS-1 and TOS
018D| EA83 D96B          #       MOVE    SORR,SP,OEY,AR,NSE,RDS
018E| E0FF C3CB          #       A2NR    1,SP,NSE
018F| E01F 9E90                  ADD     TODRR,TOS,DRS
0190| IFETCH
0190| 244F F921          #       MOVE    SOZE,NRA,IR,CRPS,JMAP

01F9|                     LESI   STACKR
01F9| EA83 D96B          #       MOVE    SORR,SP,OEY,AR,NSE,RDS
01FA| E0FF C3CB          #       A2NR    1,SP,NSE
01FB| E01F 9E50                  SUBS    TODRR,TOS,DRS
01FC| E00F 7342                  TNO     OEY                     ; test for LT
01FD| E08F D910                  MOVE    SOZR,TOS,OEY,NSE
01FE| 61F8 ****                  JUMP    NCT,$1
01FF| E00F C1F0                  S2NR    0,TOS,OEY
0200|                     $1     IFETCH
0200| 244F F921          #       MOVE    SOZE,NRA,IR,CRPS,JMAP

044F| E44F D946           UJP    MOVE    SOSER,D6,IR,CRPS        ; get jump code
0450|                     ;
0450| 61F8 ****           JMP    JUMP    N,JTABJMP               ; if neg use jtab
0451|                            READ_PC
```

FFT ROUTINE

```
092A|                ;
092A|                ; micro-coded fft routine for mip processor
092A|                ;
092A|                ; register assignments
092A|                ;
092A|                ; D0      - s.r
092A|                ; D1      - s.i
092A|                ; D2      - temp
092A|                ; D3      - scale check
092A|                ; D4      - w.r
092A|                ; D5      - w.i
092A|                ; D6      - minor loop
092A|                ; D7      - major loop
092A|                ;
092A|                ; R1      - original data array pointer
092A|                ; R2      - sin/cos table
092A|                ; R3      - offset
092A|                ; R4      - bigstep
092A|                ; R5      - scale count
092A|                ; R6      - big limit
092A|                ; R7      - pass count
092A|                ; R8      - number of points
092A|                ; R9      - working data pointer to A component
092A|                ; R10     - working data pointer to B component
092A|                ; R11     - shuffle array
092A|                ; R12     - check pointer
092A|                ;
092A|                          .MACRO  RCHEK                ; macro to do range check on data
092A|                          MOVE    SODR,D3,YREG
092A|                          JUMP    P,$1
092A|                          NEG     SODR,D3
092A|                $1        S2NR    14,D3,OEY
092A|                          JUMP    NC,$2
092A|                          SETST   F3                   ; set flag 3 for scaling
092A|                $2
092A|                          .ENDM
092A|                ;
092A| EA03 C5CB      FLUTERBY  A2NR    2,SP,OEY,AR,RDS
092B| E01F D8DB                MOVE    SODR,R11,DRS         ; shuffle pointer
092C| EA03 C3CB                A2NR    1,SP,OEY,AR,RDS
092D| E01F D8D2                MOVE    SODR,R2,DRS          ; sin/cos table
092E| EA03 C3CB                A2NR    1,SP,OEY,AR,RDS
092F| E01F D8D1                MOVE    SODR,R1,DRS          ; data pointer
0930| EA03 C3CB                A2NR    1,SP,OEY,AR,RDS
0931| E01F D8D7                MOVE    SODR,R7,DRS          ; pass count
0932|
0932| E07F D918                MOVE    SOZR,R8
0933| E01E E1B8                SETNR   0,R8,DRS,YSHFT
0934| E00F CC18                SHUPZ   SHRR,R8,OEY          ; number of points
0935| E02F D8D3                MOVE    SODR,R3,YREG         ; offset = 1/2 byte count
0936| E00F D915                MOVE    SOZR,R5,OEY          ; zero scale count
0937| E02F DCD6                INC     SODR,R6,YREG         ; big limit =1 to start
0938| 11F8 ****                CALL    UNC,RNGCHEK
0939| E07F F904                MOVE    SOZ,NRS

093A|
093A| E00F 7356      FFTL      TF3     OEY
093B| 11F8 ****                CALL    CT,SCALE             ; scale the data
093C|
093C| E07F D914                MOVE    SOZR,R4              ; bigstep
093D| E07F D816                MOVE    SORA,R6
093E| E07F D887                MOVE    SOAR,D7              ; outer loop
093F| E07F D811                MOVE    SORA,R1
```

```
0451| EE7F 7140        #       NOOP    WIO
0452| E07C 7140        #       NOOP    ABEN
0453| E07C F8C1        #       MOVE    SOD,NRA,ABEN
0454| EE02 8086                ADD     TORAA,D6,OEY,PC          ; re-load pc & fetch
0455| E206 F900                MOVE    SOZ,NRY,OEY,PCPG,RPS
0456|                          IFETCH
0456| 244F F921        #       MOVE    SOZE,NRA,IR,CRPS,JMAP
0457|                  ;
0457| E00F D80E        JTABJMP MOVE    SORA,JTAB,OEY
0458| EA03 8086                ADD     TORAA,D6,OEY,AR,RDS
0459| E01F E201                SUBR    TODA,NRA,DRS             ; self-relative
045A| EE02 F880                MOVE    SOA,NRY,OEY,PC           ; load PC and fetch
045B| E206 F900                MOVE    SOZ,NRY,OEY,PCPG,RPS
045C|                          IFETCH
045C| 244F F921        #       MOVE    SOZE,NRA,IR,CRPS,JMAP
045D|                   ;
045D|                   ; FJP S;       JUMP IF TOS IS FALSE.
045D|                   ;
045D|                   FJP     STACKR
045D| EA83 D96B        #       MOVE    SORR,SP,OEY,AR,NSE,RDS
045E| E0FF C3CB        #       A2NR    1,SP,NSE
045F| E07F E1F0                TSTNR   0,TOS
0460| E09F D8D0                MOVE    SODR,TOS,DRS,NSE         ; refresh TOS
0461| 61F8 1BB0                JUMP    Z,UJP
0462| E44F D926        NOJ     MOVE    SOZER,D6,IR,CRPS         ; dump jump byte
0463|                          IFETCH
0463| 244F F921        #       MOVE    SOZE,NRA,IR,CRPS,JMAP
0474| EE7F 7140         XJP    NOOP    WIO
0475| E07C 7140                NOOP    ABEN
0476| E07C D8CC                MOVE    SODR,IPC,ABEN            ; copy of PC
0477| E00F E1EC                TSTNR   0,IPC,OEY                ; align PC
0478| 61F8 ****                JUMP    Z,$1
0479| E07F C1CC                A2NR    0,IPC                    ; make it even
047A| EA03 D96C         $1     MOVE    SORR,IPC,OEY,AR,RDS      ; lower index
047B| E01F 8610                SUBR    TODRA,TOS,DRS            ; (TOS - lower) => case index
047C| EA83 C3CC                A2NR    1,IPC,OEY,AR,RDS,NSE     ; upper index
047D| E0FF C3CC                A2NR    1,IPC,NSE                ; inc to branch out
047E| 61F8 ****                JUMP    N,$2
047F| E01F 9650                SUBS    TODRY,TOS,DRS            ; (upper - TOS)
0480| 61F8 ****                JUMP    N,$2
0481| E07F EC01                        SHA,NRA                  ; make byte index
0482| E00F C3CC                        1,IPC,OEY                ; inc to table base
0483| EA03 988C                ADD     TORAR,IPC,OEY,AR,RDS     ; index into case table & read
0484| E01F 9E0C                SUBR    TODRR,IPC,DRS            ; self-relative
0485| EE02 D96C         $2     MOVE    SORR,IPC,OEY,PC          ; update PC & fetch
0486| E206 F900                MOVE    SOZ,NRY,OEY,PCPG,RPS
0487|                          STACKR
0487| EA83 D96B        #       MOVE    SORR,SP,OEY,AR,NSE,RDS
0488| E0FF C3CB        #       A2NR    1,SP,NSE
0489| E01F D8D0                MOVE    SODR,TOS,DRS             ; new TOS
048A|                          IFETCH
048A| 244F F921        #       MOVE    SOZE,NRA,IR,CRPS,JMAP
```

```
0940| E07F D89A              MOVE    SOAR,R10            ; pre-load with base address
0941|                 ;
0941| E07F D81A      MAJ     MOVE    SORA,R10
0942| E07F D899              MOVE    SOAR,R9             ; new A pointer
0943| E07F D813              MOVE    SORA,R3
0944| E07F D886              MOVE    SOAR,D6             ; minor loop count
0945| E07F 989A              ADD     TORAR,R10           ; new B pointer
0946| E07F D814              MOVE    SORA,R4             ; bigstep
0947| EA03 909B              ADD     TORAY,R11,OEY,AR,RDS ; index into shuffl
0948| E01F E599              RTDA    2,DRS               ; shuffl * 4
0949| EA03 8092              ADD     TORAA,R2,OEY,AR,RDS ; w.r
094A| E01F D8C4              MOVE    SODR,D4,DRS
094B| EA03 E384              A2NA    1,OEY,AR,RDS
094C| E01F D8C5              MOVE    SODR,D5,DRS         ; w.i
094D|                 ;
094D| EA03 D81A      BFLY    MOVE    SORA,R10,OEY,AR,RDS ; fetch b.r
094E| E00A D844              MOVE    SORY,D4,OEY,ENX     ; w.r => x
094F| E019 D8C2              MOVE    SODR,D2,DRS,ENY     ; b.r => y
0950| EA03 C3DA              A2NR    1,R10,OEY,AR,RDS    ; fetch b.i
0951| E05F F8C1              MOVE    SOD,NRA,OEPM        ; b.r * w.r
0952| E00A D845              MOVE    SORY,D5,OEY,ENX     ; w.i => x
0953| E019 D8C1              MOVE    SODR,D1,DRS,ENY     ; b.i => y
0954| EA03 D859              MOVE    SORY,R9,OEY,AR,RDS  ; fetch a.r
0955| E05F C280              ADD      TODAR,D0,OEPM      ; b.r*w.r + b.i*w.i => s.r
0956|
0956| E009 D842              MOVE    SORY,D2,OEY,ENY     ; b.r => y
0957| E01F D8C2              MOVE    SODR,D2,DRS         ; a.r => D2
0958| E05F F8C1              MOVE    SOD,NRA,OEPM        ; b.r*w.i
0959| E00A D844              MOVE    SORY,D4,OEY,ENX     ; w.r => x
095A| E009 D841              MOVE    SORY,D1,OEY,ENY     ; b.i => y
095B| EA03 C3D9              A2NR    1,R9,OEY,AR,RDS     ; a.i
095C| E05F C201              SUBR    TODAR,D1,OEPM       ; b.i*w.r - b.r*w.i => s.i
095D|
095D| E07F D802              MOVE    SORA,D2             ; a.r
095E| EE03 C3FA              S2NR    1,R10,OEY,AR
095F| EC01 9000              SUBR    TORAY,D0,OEY,DR,WDS ; a.r - s.r => b.r
0960|                        RCHEK
0960| E02F D8C3      #       MOVE    SODR,D3,YREG
0961| 61F8 ****      #       JUMP    P,$1
0962| E07F DEC3      #       NEG     SODR,D3
0963| E00F DDE3      #$1     S2NR    14,D3,OEY
0964| 61F8 ****      #       JUMP    NC,$2
0965| E07F 774A      #       SETST   F3
0966|                #$2
0966| EE03 C3F9              S2NR    1,R9,OEY,AR
0967| EC01 9080              ADD     TORAY,D0,OEY,DR,WDS ; a.r + s.r => a.r
0968|                        RCHEK

0968| E02F D8C3      #       MOVE    SODR,D3,YREG
0969| 61F8 ****      #       JUMP    P,$1
096A| E07F DEC3      #       NEG     SODR,D3
096B| E00F DDE3      #$1     S2NR    14,D3,OEY
096C| 61F8 ****      #       JUMP    NC,$2
096D| E07F 774A      #       SETST   F3
096E|                #$2
096E| E01F F8C1              MOVE    SOD,NRA,DRS         ; a.i
096F| EE03 C3DA              A2NR    1,R10,OEY,AR
0970| EC01 9001              SUBR    TORAY,D1,OEY,DR,WDS ; a.i - s.i => b.i
0971|                        RCHEK
0971| E02F D8C3      #       MOVE    SODR,D3,YREG
0972| 61F8 ****      #       JUMP    P,$1
0973| E07F DEC3      #       NEG     SODR,D3
0974| E00F DDE3      #$1     S2NR    14,D3,OEY
0975| 61F8 ****      #       JUMP    NC,$2
```

```
0976| E07F 774A      #         SETST   F3
0977|                #$2
0977| E07F C3DA                A2NR    1,R10                ; for auto-inc
0978|
0978| EE03 C3D9                A2NR    1,R9,OEY,AR
0979| EC01 9081                ADD     TORAY,D1,OEY,DR,WDS  ; a.i + s.i => a.i
097A|                          RCHEK
097A| E02F D8C3      #         MOVE    SODR,D3,YREG
097B| 61F8 ****      #         JUMP    P,$1
097C| E07F DEC3      #         NEG     SODR,D3
097D| E00F DDE3      #$1       S2NR    14,D3,OEY
097E| 61F8 ****      #         JUMP    NC,$2
097F| E07F 774A      #         SETST   F3
0980|                #$2
0980| E07F C3D9                A2NR    1,R9                 ; for auto-inc
0981|
0981| E07F C5E6                S2NR    2,D6
0982| 61F8 96B2                JUMP    NZ,BFLY              ; loop in bfly
0983|
0983| E07F C5D4                A2NR    2,R4                 ; for indxng shufl array (bigstep)
0984|
0984| E07F C1E7                S2NR    0,D7
0985| 61F8 96BE                JUMP    NZ,MAJ               ; major loop
0986|
0986| E07F CC16                SHUPZ   SHRR,R6              ; big limit
0987| E07F CC93                SHDNZ   SHRR,R3              ; offset
0988|
0988| E07F C1F7                S2NR    0,R7
0989| 61F8 96C5                JUMP    NZ,FFTL              ; passes
098A| E07F D815                MOVE    SORA,R5
098B| A07F D890                MOVE    SOAR,TOS,CRTS        ; return scale count in TOS
098C|                ;
098C| EA03 D811      RNGCHEK   MOVE    SORA,R1,OEY,AR,RDS
098D| E07F D89C                MOVE    SOAR,R12             ; data pointer
098E| E07F D818                MOVE    SORA,R8
098F| E07F D886                MOVE    SOAR,D6
0990| E07F D887                MOVE    SOAR,D7              ; # of points
0991| E07F D8E0 E07F 3000      MOVE    SOIR,D0,3000H        ; data limit
0993|                ;

0993| E01F F8C1      $1        MOVE    SOD,NRA,DRS          ; get data
0994| EA83 C3DC                A2NR    1,R12,OEY,AR,RDS,NSE
0995| 61F8 ****                JUMP    P,$2
0996| E07F FE81                NEG     SOA,NRA
0997| E00F 9000      $2        SUBR    TORAY,D0,OEY         ; do the compare
0998| 61F8 ****                JUMP    C,$3
0999| E07F C1E6                S2NR    0,D6
099A| 61F8 966C                JUMP    NZ,$1
099B| A1F8 7FFF                CRET    UNC
099C|                ;
099C| E07F 7543      $3        RSTST   ONCZ                 ; clear status bits
099D| E07F C1D5                A2NR    0,R5                 ; inc scale count
099E| E07F D811                MOVE    SORA,R1
099F| E07F D89C                MOVE    SOAR,R12             ; init data pointer
09A0| E07F C3FC                S2NR    1,R12
09A1|
09A1| EA03 C3DC      $4        A2NR    1,R12,OEY,AR,RDS     ; get data
09A2| E01F D8C0                MOVE    SODR,D0,DRS
09A3| EC01 CD00                SHDNOV  SHRR,D0,OEY,DR,WDS   ; write back shifted data
09A4| E07F C1E7                S2NR    0,D7
09A5| 61F8 965E                JUMP    NZ,$4
09A6| A1F8 7FFF                CRET    UNC
09A7|                ;
09A7| E00F D811      SCALE     MOVE    SORA,R1,OEY
```

```
09A8| E07F D89C              MOVE    SOAR,R12            ; data pointer
09A9| E07F C3FC              S2NR    1,R12
09AA| E07F D818              MOVE    SORA,R8
09AB| E07F D887              MOVE    SOAR,D7
09AC| E07F C1D5              A2NR    0,R5                ; inc scale count
09AD| E07F F904              MOVE    SOZ,NRS
09AE| E07F 7543              RSTST   ONCZ                ; clear status bits
09AF|
09AF| EA03 C3DC      $4      A2NR    1,R12,OEY,AR,RDS    ; get data
09B0| E01F D8C0              MOVE    SODR,D0,DRS
09B1| EC01 CD00              SHDNOV  SHRR,D0,OEY,DR,WDS  ; write back shifted data
09B2| E07F C1E7              S2NR    0,D7
09B3| 61F8 9650              JUMP    NZ,$4
09B4| A1F8 7FFF              CRET    UNC
09B5|               ;
```

Am29116 Processor

This circuit is a micro-programmable 16-bit processor. In addition to its complete arithmetic and logic functions, it contains functions that are particularly useful in controller-applications; Bit Set, Bit Reset, Bit Test, Rotate and Merge, Rotate and Compare, and Cyclic-Redundancy-Check (CRC) Generation. The divice consists of the following functional blocks (Figure B-1):

1) The 32-word by 16-bit RAM is a single-port RAM with a latch at its output. With the use of an external multiplexer, it is possible to select separate read and write addresses for the same instruction.

2) The accumulator is an edge-triggered register.

3) The data latch is able to hold data when DLE is Low.

4) The barrel shifter rotates data up to 15 positions.

5) The ALU has full carry lookahead across all 16 bits in the arthmetic mode. It has the ability to execute all conventional one- and two- operand operations. In addition, it can also execute three-operand instructions such as Rotate and Merge, and Rotate and Compare with masks. It provides 3 status outputs, C (carry), N (negative) and OVR (overflow).

6) The priority encoder produces a binary-weighted code to indicate the location of the highest order ONE in the data.

7) The status register holds 8 status-bits.

Flag3	Flag2	Flag1	Link	OVR	N	C	Z

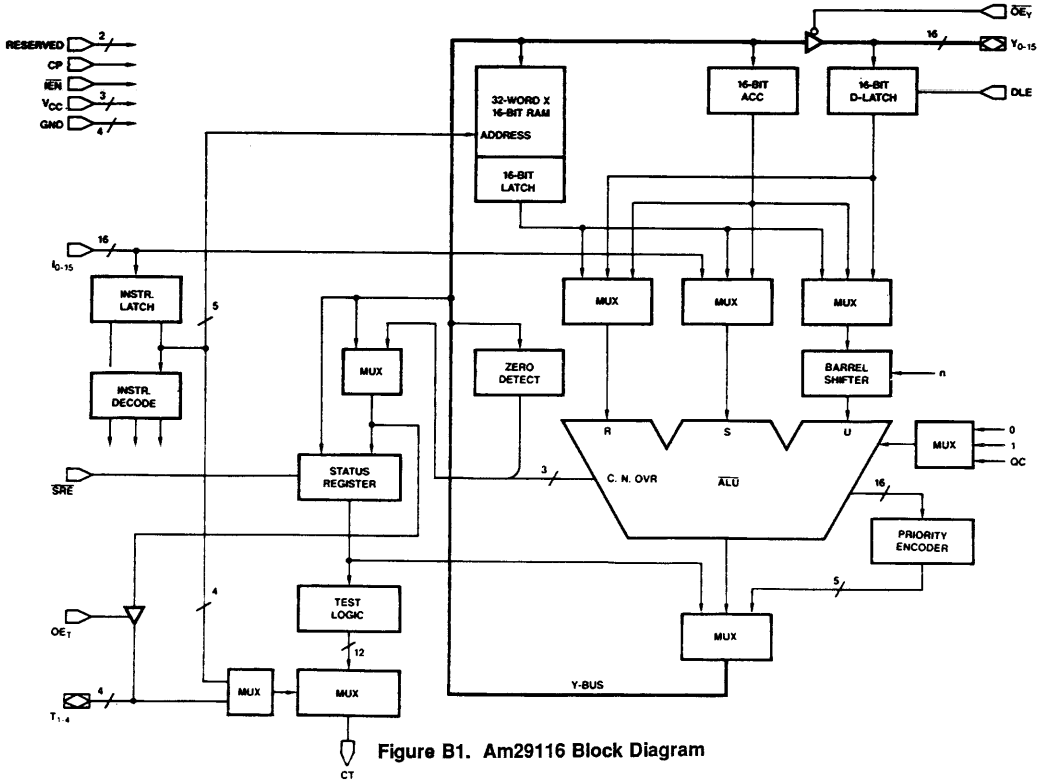


Figure B1. Am29116 Block Diagram

BD001960

8) The Condition-Code Generator/Multiplexer contains the logic necessary to develop the 12 condition-code test signals.

9) The 16-bit instruction latch is normally transparent to allow decoding of the instruction inputs by the decoder. All instruction, except immediate instruction, are executed in a single clock cycle. Immediate instruction requires 2 clock cycles for execution.

Am29517 Multiplier

This circuit performs the parallel multiplication of two 16-bit number, X and Y (see Figure B2). The product P is generated in the form of two 16-bit words that can be read out one after the other, on bus P or both together; the more significant bits on bus P, the less significant bits on bus Y. Control signals allow the part:

1) To accept the numbers X and Y, after an enable bit (ENX, WNY). The data is then stored in an input register simultaneously with a flag XM, YM, specifying whether the numbers are unsigned or in two's complement.

2) To define the output format as 32 or 31 bits. The 31-bit configuration is used if the data are two's complement fractions.

3) To use a transparent or pipelined output structure (FT). For a pipeline structure, an enable bit is necessary (ENPD). This configuration is the fastest, with a 65 ns maximum cycle time.

4) To switch some buses (OEP, OEL) to high impedence.

5) To round the 16 most significant bits when the 16 less significant bits are not used (RND).

Figure B2 shows the internal diagram of this circuit.

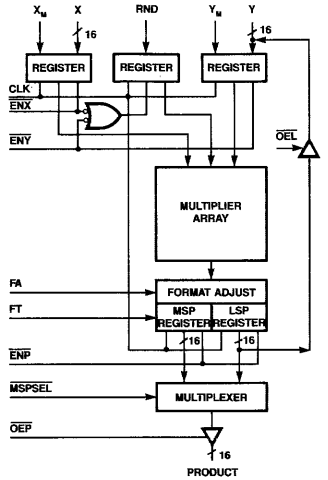


Figure B2. Am29517 Block Diagram

Diagnostics–WCS Pipeline Register

This circuit is an 8-bit pipeline register with an on-board shadow register.

1) The pipeline register can load parallel data to or from the shadow register; input data from the D-port, and output data to the Y-port.

2) The shadow register can load parallel data to or from the pipeline register and can output data through the D input port (as in WCS loading). It can also input serial data from the SDI input and output serial data through the SDO output.

Figure B4 shows the internal diagram of this circuit.

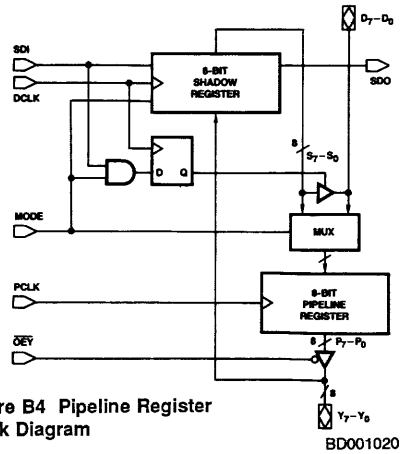


Figure B4 Pipeline Register Block Diagram

BD001020

Am2910A Microprogram Sequencer

This is a 12-bit address sequencer intended for controlling the execution sequence of micro-instructions stored in the microprogram memory. it consists of the following 5 functional blocks (Figure B3).

1) The four input multiplexer selects one of the following four sources:

µPC	Microprogram Counter
D	Direct Input
R	Register/Counter
F	Stack

2) The microprogram counter is composed of an incrementer followed by a register.

3) The internal loop counter is a pre-setable down-counter for repeating instructions and continuing loop itrations.

4) The 9-word deep stack provides return address linkage when executing micro-subroutines or loops.

5) The built-in decoder enables one of the following three direct input sources:

PL	Pipeline Register
MAP	MAP PROM
VECT	Interrupt Vector

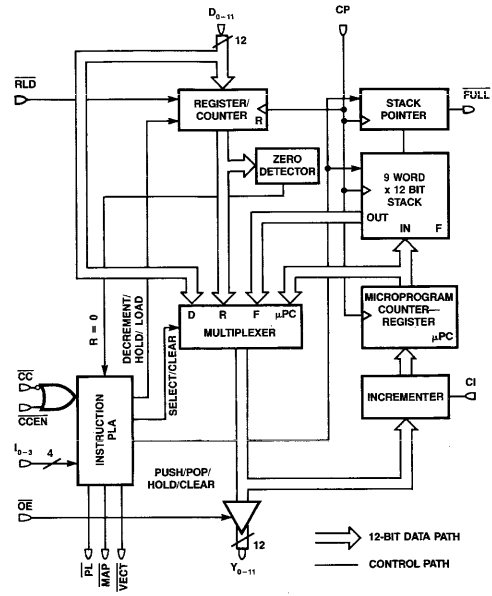


Figure B3 Am2910A Microprogram Sequencer

Am2964 Dynamic Memory Controller

This circuit provides address-multiplexing, refresh address-generation, and RAS/CAS control for the dynamic RAM memories. It can address up to 256 K and provide both 128 and 256 line refresh capability.

1) Two 8-bit address latches and an 8-bit refresh address generator feed into a multiplexer for output to the dynamic RAM address lines.

2) The RAS decoder allows 2 upper addresses to select one-of-four banks of RAMs.

Figure B5 shows the internal diagram of this circuit.

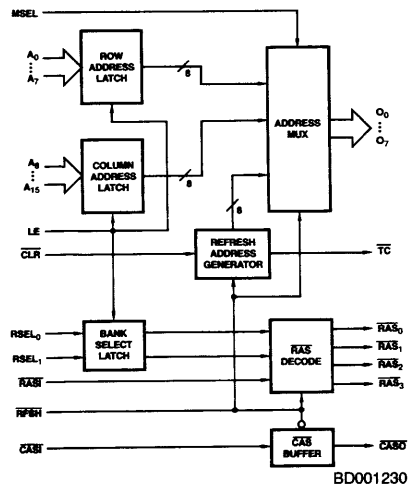
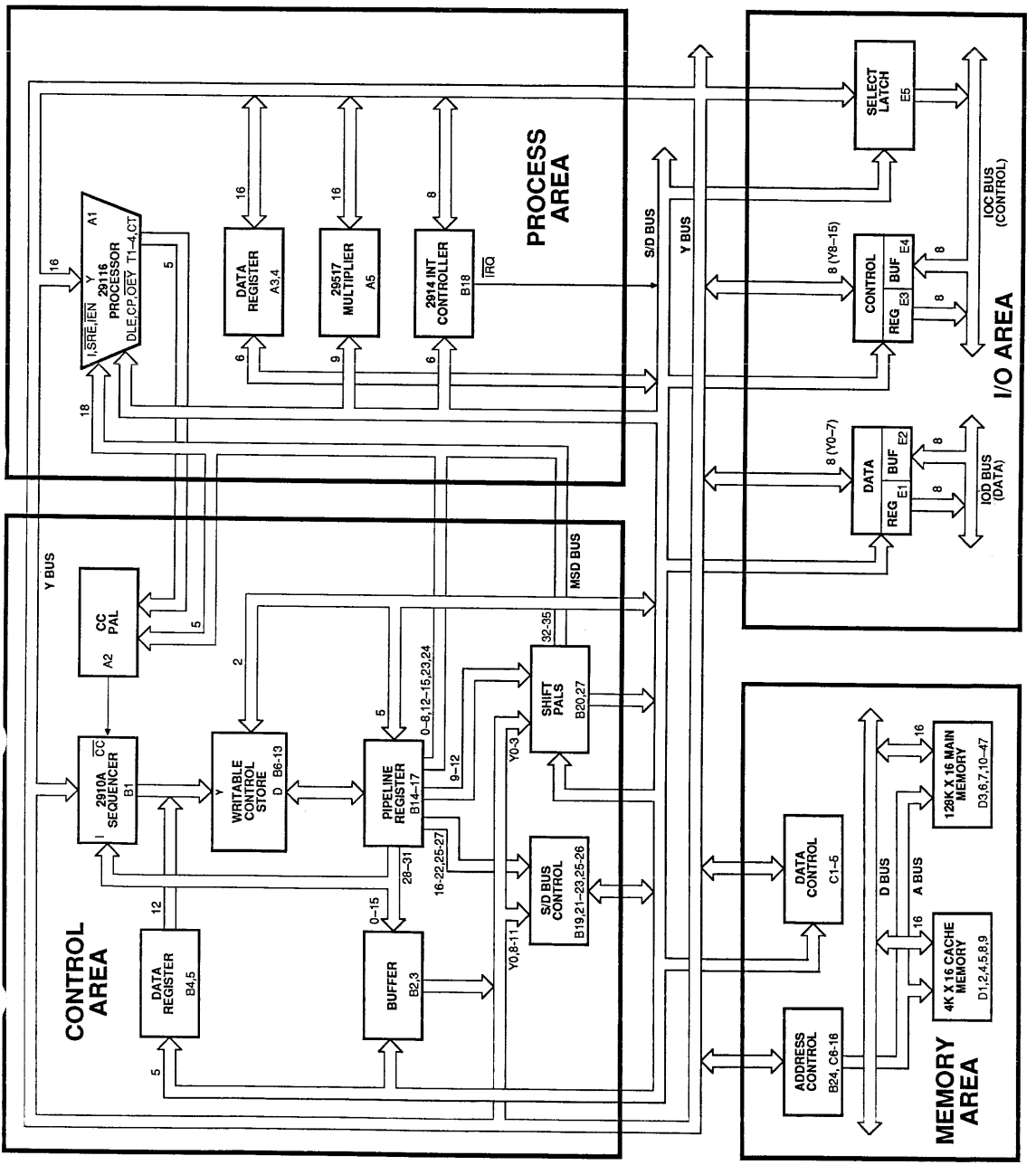


Figure B5 Am2964 Block Diagram

Appendix C

MIP Board Block Diagram



ADVANCED MICRO DEVICES
DOMESTIC SALES OFFICES

ALABAMA	(205) 882-9122
ARIZONA,	
Tempe	(602) 242-4400
Tucson	(602) 792-1200
CALIFORNIA,	
El Segundo	(213) 640-3210
Newport Beach	(714) 752-6262
San Diego	(619) 560-7030
Sunnyvale	(408) 720-8811
Woodland Hills	(818) 992-4155
COLORADO	(303) 691-5100
CONNECTICUT,	
Southbury	(203) 264-7800
FLORIDA,	
Altamonte Springs	(305) 339-5022
Clearwater	(813) 530-9971
Ft Lauderdale	(305) 484-8600
Melbourne	(305) 254-2915
GEORGIA	(404) 449-7920
ILLINOIS	(312) 773-4422
INDIANA	(317) 244-7207
KANSAS	(913) 451-3115
MARYLAND	(301) 796-9310
MASSACHUSETTS	(617) 273-3970
MINNESOTA	(612) 938-0001
NEW JERSEY	(201) 299-0002
NEW YORK,	
Liverpool	(315) 457-5400
Poughkeepsie	(914) 471-8180
Woodbury	(516) 364-8020
NORTH CAROLINA,	
Charlotte	(704) 525-1875
Raleigh	(919) 847-8471
OREGON	(503) 245-0080
OHIO,	
Columbus	(614) 891-6455
PENNSYLVANIA,	
Allentown	(215) 398-8006
Willow Grove	(215) 657-3101
TEXAS,	
Austin	(512) 346-7830
Dallas	(214) 934-9099
Houston	(713) 785-9001
WASHINGTON	(206) 455-3600
WISCONSIN	(414) 782-7748

INTERNATIONAL SALES OFFICES

BELGIUM,		
Bruxelles	TEL:	(02) 771 99 93
	FAX:	762-3716
	TLX:	61028
CANADA, Ontario,		
Kanata	TEL:	(613) 592-0090
Willowdale	TEL:	(416) 224-5193
	FAX:	(416) 224-0056
FRANCE,		
Paris	TEL:	(01) 687.36.66
	FAX:	6862185
	TLX:	20253
GERMANY,		
Hannover area	TEL:	(05143) 50 55
	FAX:	5553
	TLX:	925287
München	TEL:	(089) 41 14-0
	FAX:	406490
	TLX:	523883
Stuttgart	TEL:	(0711) 62 33 77
	FAX:	625187
	TLX:	721882
HONG KONG,		
Kowloon	TEL:	3-695377
	FAX:	1234276
	TLX:	50426
ITALY, Milano	TEL:	(02) 3390541
	FAX:	3498000
	TLX:	315286
JAPAN, Tokyo	TEL:	(03) 345-8241
	FAX:	3425196
	TLX:	J24064 AMDTKOJ
LATIN AMERICA,		
Ft. Lauderdale	TEL:	(305) 484-8600
	FAX:	(305) 485-9736
SWEDEN, Stockholm	TEL:	(08) 733 03 50
	FAX:	7332285
	TLX:	11602
UNITED KINGDOM,		
Manchester area	TEL:	(0925) 828008
	FAX:	827693
	TLX:	628524
London area	TEL:	(04862) 22121
	FAX:	22179
	TLX:	859103

NORTH AMERICAN REPRESENTATIVES

CALIFORNIA	
I^2 INC	OEM (408) 988-3400 DISTI (408) 498-6868
CONNECTICUT	
SCIENTIFIC COMPONENTS	(203) 272-2963
IDAHO	
INTERMOUNTAIN TECH MKGT	(208) 322-5022
INDIANA	
SAI MARKETING CORP	(317) 241-9276
IOWA	
LORENZ SALES	(319) 377-4666
MICHIGAN	
SAI MARKETING CORP	(313) 227-1786
NEBRASKA	
LORENZ SALES	(402) 475-4660
NEW JERSEY	
TAI CORPORATION	(609) 933-2600
NEW MEXICO	
THORSON DESERT STATES	(505) 293-8555
NEW YORK	
NYCOM, INC	(315) 437-8343
OHIO	
Dayton	
DOLFUSS ROOT & CO	(513) 433-6776
Strongsville	
DOLFUSS ROOT & CO	(216) 238-0300
PENNSYLVANIA	
DOLFUSS ROOT & CO	(412) 221-4420
UTAH	
R^2 MARKETING	(801) 595-0631

ADVANCED MICRO DEVICES 901 Thompson Pl., P.O. Box 3453, Sunnyvale, CA 94088, USA
TEL: (408) 732-2400 • TWX: 910-339-9280 • TELEX: 34-6306 • TOLL FREE: (800) 5388450

Printed in U.S.A. IH-WCP-5M-12/85-0

Order #07339A