Design and Implementation of the Alewife Startup Module

by

William K. Chan

S.B., Electrical Science and Engineering
Massachusetts Institute of Technology

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements of the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 1995

Author ______________________________
Department of Electrical Engineering and Computer Science
May 25, 1995

Certified by [illegible]
Anant Agarwal
Thesis Supervisor

Accepted by [illegible]
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Theses

Design and Implementation of the Alewife Startup Module

by

William K. Chan

Submitted to the
Department of Electrical Engineering and Computer Science

May 25, 1995

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The Alewife prototype uses a host connected via a VME bus for its communications. This configuration is a bottleneck for operations which require external devices. A method to use SCSI has been developed, but it cannot fully replace the startup sequence generated by the VME-based host. The focus of this thesis is to design and implement the necessary hardware and software to perform the startup sequence, allowing the removal of the VME interface. In addition, the clock synthesizer from the VME transceiver board will be enhanced and incorporated into the startup module to consolidate hardware.

Thesis Supervisor: Anant Agarwal
Title: Jamieson Career Development Associate Professor of Computer Science
and Engineering

Acknowledgments

Thanks go to Professor Anant Agarwal for his guidance and support throughout this thesis. Also, Ken Mackenzie was extremely valuable by helping with all aspects of design and testing, from explaining the design tools to writing some test code of his own. He was a wonderful source of advice and information.

In addition, a special thanks go to my parents, Raymond and Lily Chan, for their continuing emotional and financial support.

Table of Contents

Chapter 1
Introduction

Background

The Alewife machine is a large-scale distributed-memory multiprocessor organized as a set of processing nodes connected in a mesh topology. Each node consists of a processor, a cache, a portion of globally-shared distributed memory, a cache-memory-network controller, a floating-point coprocessor, and a network switch. Free network ports on peripheral nodes of the mesh are used for I/O, monitor, and host connections. The Alewife prototype attaches to a host Sun 4/110 by interfacing a network switch to the VME bus. A block diagram of the Alewife prototype is shown in Figure 1-1 [1].

This configuration is a bottleneck for operations which require external devices since all external access is handled through the host computer. Thus, the next step in the development of Alewife is to improve its communications capabilities. One method involves the addition of new non-processing nodes capable of SCSI communication. This not only enhances the communications capabilities of the machine by increasing connections, but also provides a more common interface standard by using SCSI rather than VME.

The addition of the SCSI nodes provides an opportunity to enhance the Alewife environment. With the presence of the SCSI interface, there is very little use for the old VME interface. In fact, the only task that does not have a SCSI equivalent is the startup sequence since it depends on the direct link

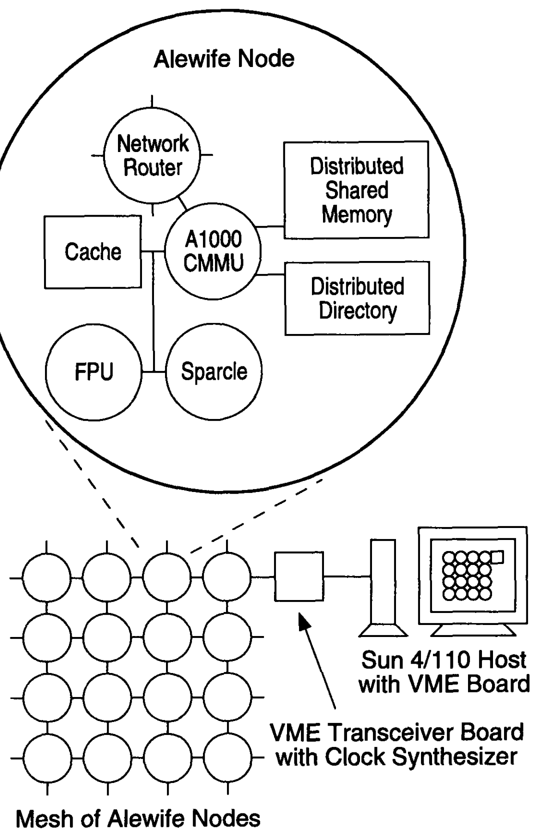
Alewife Node
Network Router
Distributed Shared Memory
Cache
A1000 CMMU
Distributed Directory
FPU
Sparcle
Sun 4/110 Host with VME Board
VME Transceiver Board with Clock Synthesizer
Mesh of Alewife Nodes

Figure 1-1: Block Diagram of Alewife Prototype

between the mesh network switch and VME bus. SCSI nodes can be considered to be specialized processing nodes and need to receive startup information themselves so they cannot initiate the startup sequence. However, it is unreasonable to maintain a VME-based host for the sole purpose of starting up the machine.

The focus of this thesis is the design and implementation of hardware and software to support the SCSI-capable Alewife machine such that it can function without a VME interface. This includes performing the startup sequence and generating the system clock which are functions of the VME interface.

Overview

The startup module is a stand-alone hardware and software unit which enhances the Alewife machine by allowing it to startup without external resources. In particular, it eliminates the need for a VME-based host computer to execute the startup sequence. Instead, it contains a startup engine in hardware and interfaces directly to a peripheral node of the Alewife machine through an unused network switch port. In addition, an improved version of the clock synthesizer located on the VME transceiver board is included to consolidate hardware. A block diagram of the new Alewife system is shown in Figure 1-2.

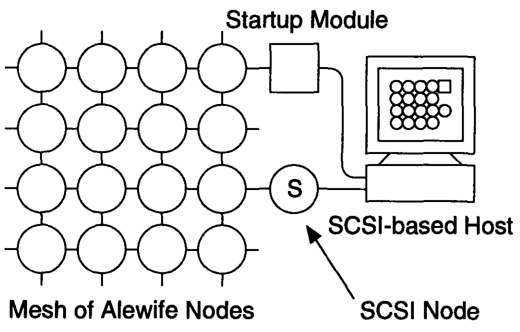

Figure 1-2: Block Diagram of New Alewife System

The startup engine is designed to be invisible to the user in its operation. It automatically sends startup information to the node whenever the Alewife machine is reset. Its major components are a microcontroller unit (MCU) and an electrically-erasable programmable read-only memory (EEPROM). The EEPROM contains both the software for the MCU and the startup information for the Alewife machine. The EEPROM can be changed easily, making it simple to add software features to the startup module and update the Alewife startup information.

The clock synthesizer in its default mode of operation is also transparent to the user. It is responsible for providing the system clock for the Alewife machine as a differential positive ECL (pECL) signal. The system actually consists of a MCU-controlled phase-locked loop (PLL) which generates a standard ECL signal at a default frequency of 40 MHz. The clock frequency can be adjusted within a range of 5 MHz to 80 MHz by the user. Control is accomplished through the clock synthesizer software in the MCU.

Chapter 2
System Requirements

Startup Engine

The purpose of the startup engine is to send startup information directly to an Alewife processing node. It consists mainly of a Motorola 68HC11 MCU and an EEPROM. A block diagram of the startup engine is shown in Figure 2-1. The 68HC11 operates in its expanded multiplexed mode in order to access the EEPROM. The EEPROM contains both the software for the 68HC11 and the Alewife startup information. In expanded multiplexed mode, the 68HC11 is capable of fetching its instructions directly from the EEPROM, making software updates to the 68HC11 very easy.

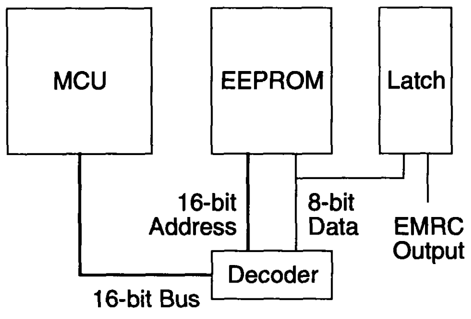

Figure 2-1: Startup Engine Block Diagram

The operation of the startup engine requires a direct connection to an unused network switch port. The network switch used for Alewife processing nodes is the Elko Mesh Routing Chip (EMRC) from the California Institute of Technology [2]. The EMRC uses eight-bit channels and operates asynchronously, requiring three additional bits for handshaking. In addition, the startup engine needs to be able to reset the Alewife machine, requiring one additional signal. The hardware interface from the 68HC11 cannot be accomplished by only using its built-in ports. Expanded multiplexed mode occupies 16 I/O pins, leaving only eight digital pins available and the interface requires at least ten outputs. In order to solve this, a latch connected to the data bus can be used to provide additional outputs. The latch can load values from the data bus to use as outputs. However, this method of generating outputs is not glitch free, but it is sufficient for the 8-bit EMRC data channel which is not continuously sampled.

The specifics of the network switch protocol are handled by the 68HC11 software. The primary concern in communicating with the EMRC is the handshaking protocol since the EMRC operates asynchronously. In particular, the protocol expects a data transfer at each transition of the handshake.

Clock Synthesizer

The clock synthesizer is responsible for providing the clock signal for the entire Alewife machine. The signal generated conforms to pECL voltage levels and is configurable from 5 MHz to 80 MHz. The synthesizer is built around a programmable PLL which interfaces directly with a 68HC11. The 68HC11 provides the user interface to adjust the frequency of the clock signal. The PLL generates a signal at standard ECL voltage levels and requires conversion to the pECL levels expected by the Alewife machine. A block diagram of the clock synthesizer is shown in Figure 2-2.

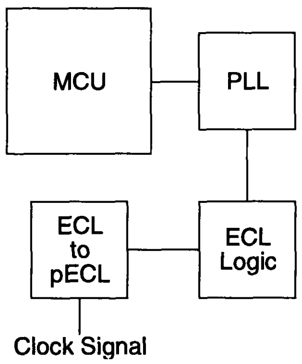

Figure 2-2: Clock Synthesizer Block Diagram

The clock synthesizer is based on the one located on the VME transceiver board [3]. Its design centers around the Motorola 145170 PLL Frequency Synthesizer with Serial Interface. The 145170, along with a VCO and a low pass filter, form the PLL which is interfaced serially to the 68HC11. The 68HC11 adjusts the frequency of the PLL by changing the value of a control register in the 145170.

The PLL is not capable of generating the entire frequency range desired due to the limits of the VCO. In order to ensure a valid signal throughout the desired range, the PLL is only designed to generate signals from 20 MHz to 80 MHz. The frequencies from 5 MHz to 20 MHz are then generated by dividing the PLL output by four using an ECL divider. The desired signal is then selected with a multiplexer. The accompanying software makes the entire frequency range appear continuous to both the Alewife machine and the user.

The clock synthesizer also has voltage requirements unique to the Alewife environment. The CMOS and TTL parts use +5 V and ground, but the ECL parts require -5 V. Although the power supply for the startup module does not generate -5 V, it does provide -12 V. A voltage regulator is used to generate the appropriate voltage level for the ECL parts.

System Issues

Both the design of the startup engine and the clock synthesizer utilize the 68HC11. Therefore, it is logical to try using the same part for both systems. From the software perspective, this is not a problem since each one uses minimal 68HC11 resources. However, there is hardware limitation in the number of output pins available. This can be alleviated by adding another latch to capture values from the data bus, similar to the latch used for the EMRC data in the startup engine. This provides eight additional outputs for general use. These outputs can only be used for signals which can tolerate glitches. Otherwise, a direct interface to the 68HC11 is required.

In addition to the startup engine and the clock synthesizer, several supporting applications can be developed for the startup module. In particular, a power supply controller requires only six I/O signals and which can be controlled through software by the 68HC11. Also, the Port E of the 68HC11 is available for the development of analog applications.

Chapter 3
Hardware Design and Implementation

Startup Engine

The startup engine is comprised mainly of the 68HC11, an EEPROM, and various parts for use in inputs and outputs. Specific parts used are the 28F512 EEPROM, 22V10 programmable array logic (PAL), LS244 buffer, and LS373 latch. All of the general logic necessary in the startup engine can be implemented within the PAL, saving many general logic parts. The schematic for the startup engine is shown in Figure 3-1.

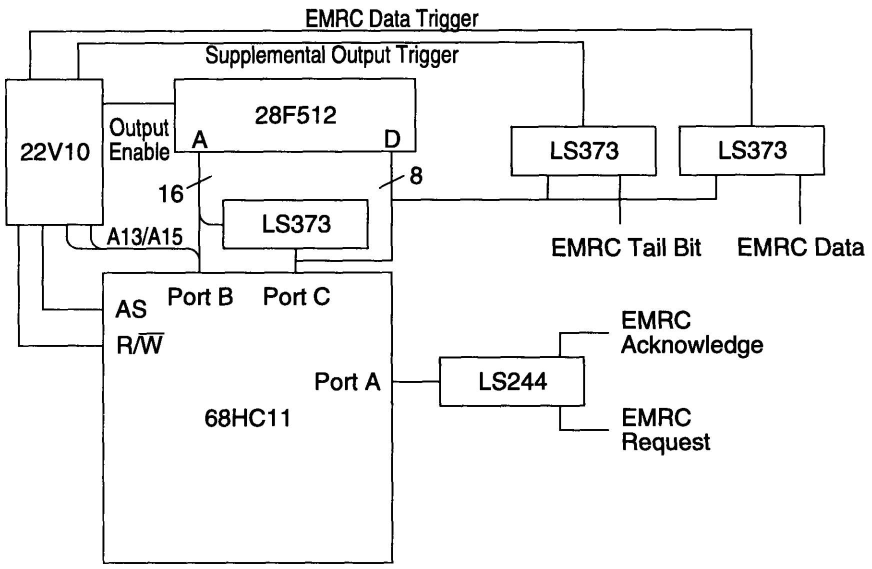

Figure 3-1: Startup Engine Schematic

In the normal operation of the startup engine, the 68HC11 is in its expanded multiplexed mode [4]. This allows it to access the EEPROM by using only eighteen bits for address, data, and bus control. The address is 16 bits and the 8-bit data is multiplexed with the lower eight bits of the address. The address, R/$\overline{W}$, and AS are active and valid for all bus cycles including accesses of internal addresses.

The address decoding in expanded multiplexed mode is essentially handled by one signal. The AS signal from the 68HC11 is used as the trigger of a LS373 latch. This LS373 latches the lower address bits when AS is high, ensuring its availability during the data portion of the cycle. The R/$\overline{W}$ signal is used to determine the direction of data transfer on the bus.

A timing diagram for a typical 68HC11 bus transaction is shown in Figure 3-2 [5]. The E clock generated by the 68HC11 establishes the rate for bus cycles. In the startup module, the E clock frequency is 2 MHz. The address bits are available each E clock cycle while the clock is low. The AS occurs during this half cycle, allowing the LS373 to latch the lower address bits. In the next half cycle while the E clock is high, the data is active and can be considered valid at the falling edge of the E clock.

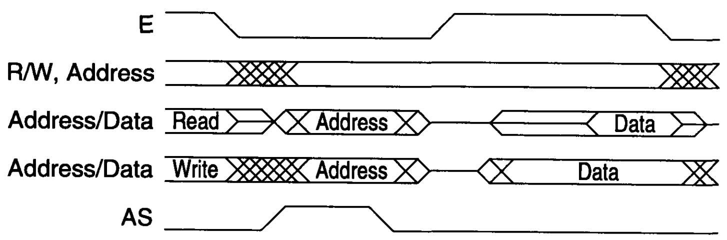

Figure 3-2: Data Bus Timing Diagram

The LS244 is used for buffering inputs and outputs between the 68HC11 and external elements. By buffering outputs, the LS244 becomes the part driving and receiving signals, insulating the 68HC11 from abnormal voltage levels. This is especially useful for signals which interact with other components of the Alewife machine not resident on the startup module.

The two LS373 latches which are used as additional outputs are connected directly to the data bus. The latches capture values from the data bus which are used as outputs to devices which are not glitch sensitive. The triggers to these latches are generated from the PAL. These triggers are based on the address targeted when the data is placed on the bus, as well as the $R/\overline{W}$ signal and the E clock. Depending on the particular address, either latch may be activated to capture the values from the data bus. The implementation only requires two address bits to decode which latch, if either, should be triggered.

The PAL is used for general combinational logic. Signals with a direct influence on the startup engine are the Alewife reset signal, the latch triggers, and the EEPROM output enable signal. The Alewife reset signal is generated by the 68HC11 and initialized low when the startup module is turned on. However, the signal should be initialized high so the PAL inverts the 68HC11 signal before it is driven to the EMRC. The latch triggers and the EEPROM output enable signal are all designed to be active during the data portion of the E clock cycle. The EEPROM output enable is activated whenever it is a read operation, and the latch triggers are active on write operations to particular address locations. The exact implementation of each of these signals is included in Appendix A.

Clock Synthesizer

The clock synthesizer generates the signal used as the Alewife system clock. Frequencies from 5 MHz to 80 MHz are supported by the synthesizer. Voltage levels produced are designed to meet pECL standards. Frequency changes by the synthesizer are smooth enough so that it can be changed while the machine is running without causing harm.

The clock synthesizer is built around a configurable PLL which generates an ECL signal between 20 MHz and 80 MHz. The output of the PLL is divided by four in order to generate frequencies between 5 MHz and 20 MHz. The combination of these two outputs provides the full frequency range specified. The desired signal is selected using an 2-to-1 multiplexer. The output is then converted to pECL voltage levels in two stages. A schematic of the clock synthesizer is shown in Figure 3-3.

The main component of the PLL is the 145170 [6]. This application-specific part contains most of the circuitry necessary for a configurable PLL, including a built-in serial interface compatible with the 68HC11. The PLL is completed with the addition of a VCO and a low pass filter. In this case, the VCO is the Motorola 1658 and the low pass filter is an active integrator built around a LF353 operational amplifier. The design of the PLL follows an example in the Motorola data book closely [7]. One of the important parameters is the resulting jitter in the PLL. This will provide a better signal, although it may result in a very long time-to-lock. However, the time-to-lock is not very critical in the case of the clock synthesizer allowing some flexibility

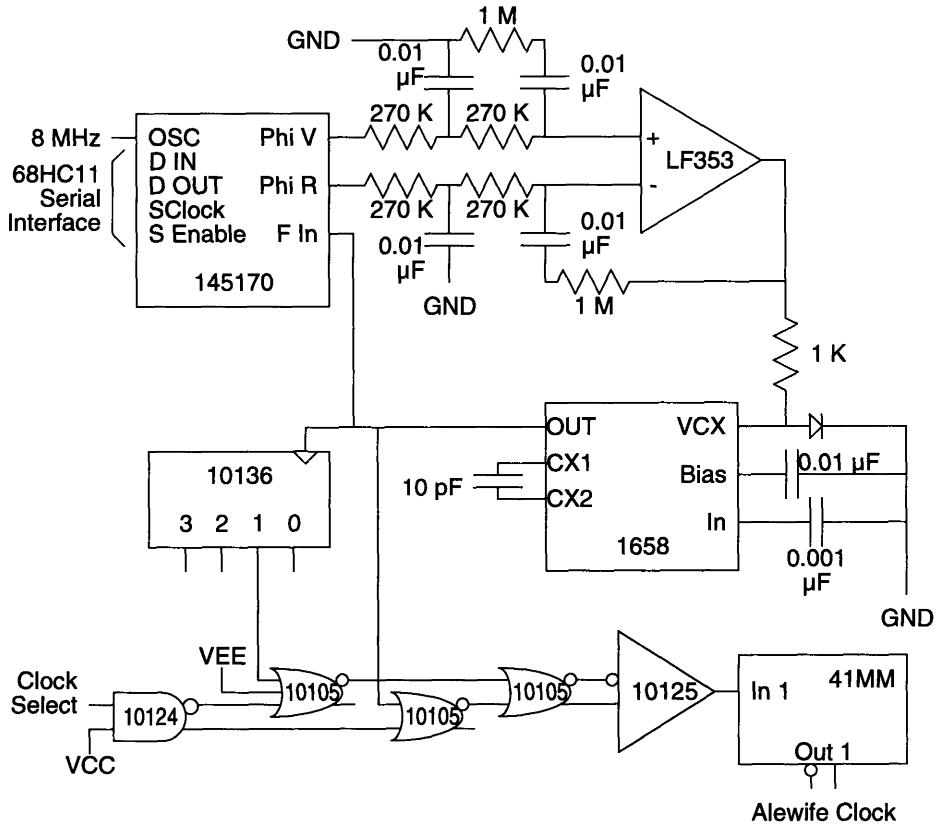

Figure 3-3: Clock Synthesizer Schematic

to minimize jitter. The details of the calculation of the PLL parameters are included in Appendix B.

Dividing the output of the PLL to generate the 5 MHz to 20 MHz range is straight forward with a good selection of ECL parts available. A simple method is to use the 10136 universal hexadecimal counter. By using the PLL output as the clock input to the 10136, the second least significant bit is equivalent to the clock divided by four when the counter is configured to count continuously [8].

The next step is to select the appropriate clock with a 2-to-1 multiplexer. Since the select signal will be at TTL voltage levels, the easiest method is to build a special multiplexer from standard logic parts. As a result, a 10105 triple 2-3-2 input OR/NOR gate and a 10124 TTL to ECL translator were used. The 10124 is capable of providing both the translated value and its complement, making the design of the multiplexer a little simpler. The two 2-input OR/NOR gates are used to select the signal to be passed through, and the remaining 3-input OR/NOR is used to combine the two signals. Both the output signal and its complement are used for the conversion to pECL voltage levels.

The converter to pECL voltage levels is built in two stages. The first stages uses the 10125 ECL to TTL translator. The 10125 requires both the input signal and its complement to function optimally. Fortunately, the 10105 is capable of providing the necessary signals. The output of the 10125 is then input into the AT&T 41MM transceiver. The 41MM converts from TTL to pECL voltage levels and outputs both the signal and its complement [9]. The result is a differential clock for the Alewife machine at pECL voltage levels.

By using ECL parts in the implementation of the clock synthesizer, it is also necessary to provide the appropriate voltage levels. The power supply for the startup module happens to provides -12 V for its negative voltage source. It is necessary to build a voltage regulator to provide the -5 V supply needed for ECL logic. In this case, the LM337 adjustable regulator is used to perform the conversion. The design is based on an example obtained from the LM337 data sheet [10]. The schematic for the voltage regulator used is shown in Figure 3-4. The first step in the conversion is a series of five diodes, providing an initial voltage change to approximately -8 V from the original -12 V. The LM337 is then used to regulate the voltage more precisely. The voltage change provided by the diodes reduces the voltage change required for the LM337, distributing the power dissipation among the parts.

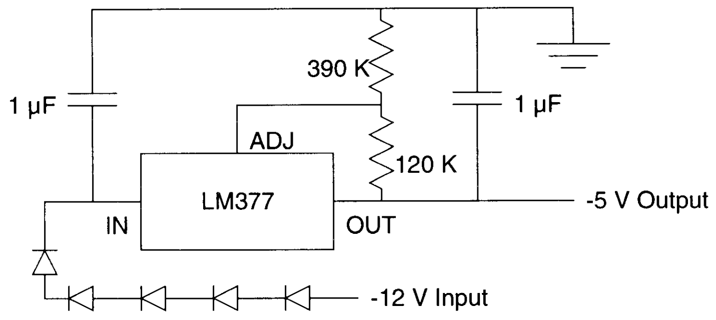

Figure 3-4: Voltage Regulator Schematic

Additional Hardware

There are several pieces of supporting hardware which are also critical to the overall operation of the startup module. These include the EMRC connector, RS-232 serial interface, the reset circuitry, and the hardware mode selector. In addition, hardware support for a power supply controller and the use of the analog port of the 68HC11 have been implemented on the startup module.

EMRC Connector

The connector which interfaces with the Alewife node is a 60-pin ribbon cable connector. It contains connections for the EMRC communications protocol, the Alewife reset signal, and the differential clock signal generated by the clock synthesizer.

RS-232 Serial Interface

The RS-232 serial interface is a fairly simple design. In this particular case, the Maxim 233A is interfaced directly with the 68HC11 [11]. The 233A is capable of converting between TTL and RS-232 voltage levels and is an ideal one-part solution. The 68HC11 uses only two lines for its serial interface. One line is for transmitting and the other is for receiving. The RS-232 interface requires a third line for reference which is connected to ground. The startup module uses a 26-pin ribbon cable connector for connecting to the serial port. The pins are chosen such that it converts directly to a standard RS-232 25-pin connector when pin 26 is omitted.

Power-up Reset

The power-up reset circuitry is necessary to ensure the 68HC11 runs properly when the power is first turned on. This can be accomplished using an RC circuit which slowly raises the 68HC11's active-low reset line when the board is first turned on. This guarantees that the 68HC11 is in reset until the power supply has stabilized. A diode bypasses the resistor to allow the capacitor to discharge when power is turned off.

Hardware Mode Selector

The 68HC11 reserves two pins for mode selection during reset [4]. Normally, the startup module operates with both of these pins high, enabling the expanded multiplexed mode. However, if it becomes necessary to debug software or change the configuration of the 68HC11, it may be helpful to use one of the special modes. As a result, jumpers are included to provide access to the special modes. Table 1 shows which modes are activated with each jumper combination.

Table 1: Jumpers for Hardware Mode Selection

MODA	MODB	68HC11 Mode
		Expanded Multiplexed (Startup Module Default)
✔	✔	Special Bootstrap
✔		Single Chip
	✔	Special Test

Power Supply Controller

The startup module is also the ideal location to place the Alewife power supply controller. The power supply can be observed and controlled using six signals. There is no external logic needed for decoding these signals so the remainder of the system can be implemented in software for the 68HC11. The connector for the power supply controller is provided on the board.

Analog Applications

Port E of the 68HC11 is capable of capturing analog information. By using latches to extend the output capabilities of the 68HC11, it has allowed Port E to be reserved for analog applications. Potential applications such as temperature sensing can be useful in examining real-world aspects of the Alewife machine such as temperature.

Board Design and Layout

The final design for the startup module was built on a four-layer board. Two layers were used as power planes, leaving two for use in routing. One power plane was used for ground and the other plane was shared by both +5 V and -5 V. The ECL parts in the clock synthesizer which required -5 V were placed in a cluster, allowing the use of a small portion of the plane as the -5 V supply. The remainder of the plane was used for +5 V.

In addition to the circuitry for the startup engine and the startup module, extra connectors were included to support the power supply controller and Port E of the 68HC11. A large prototype area was also included on the board in case any additional circuitry is necessary. A picture of the assembled board is shown in Figure 3-5.

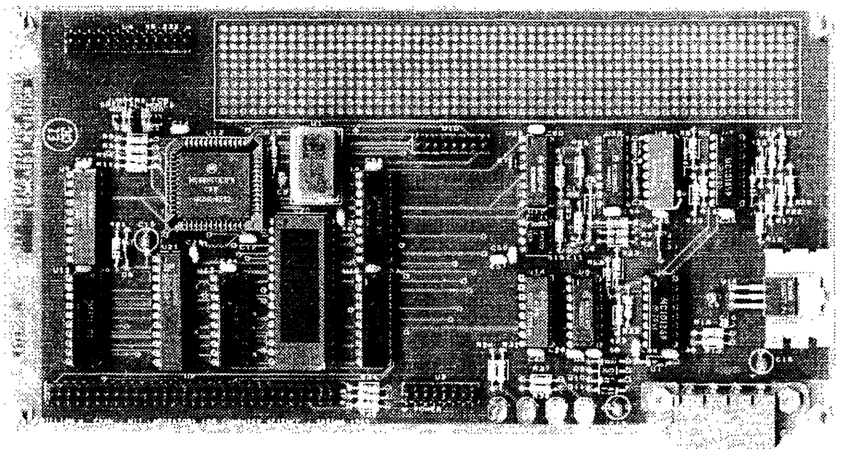

Figure 3-5: Alewife Startup Module Board

Chapter 4
Software Design & Implementation

The software for the startup module consists of several major routines. First, the setup routine initializes the 68HC11, I/O ports, and variables for the other routines. Then, there are two independent controls for the startup engine and the clock synthesizer. A flow chart of the software is shown in Figure 4-1. The actual code implemented for the startup module is included in Appendix C.

Global Setup

The setup routine prepares the 68HC11 for operation of both the startup engine and the clock synthesizer. Its primary purpose is to prepare the serial ports, set defaults for the startup engine, enable the proper I/O ports, and initialize the clock synthesizer to its default frequency.

The serial ports are initialized by setting the appropriate bits in the registers which control them. Features such as baud rate can be adjusted to allow optimal performance of the serial interface.

There are only two defaults for the startup engine. These are the X and Y offsets designating the target node for the data transfer. The default is for the offsets to be zero, thus sending the startup information to the node which the startup module is attached to directly.

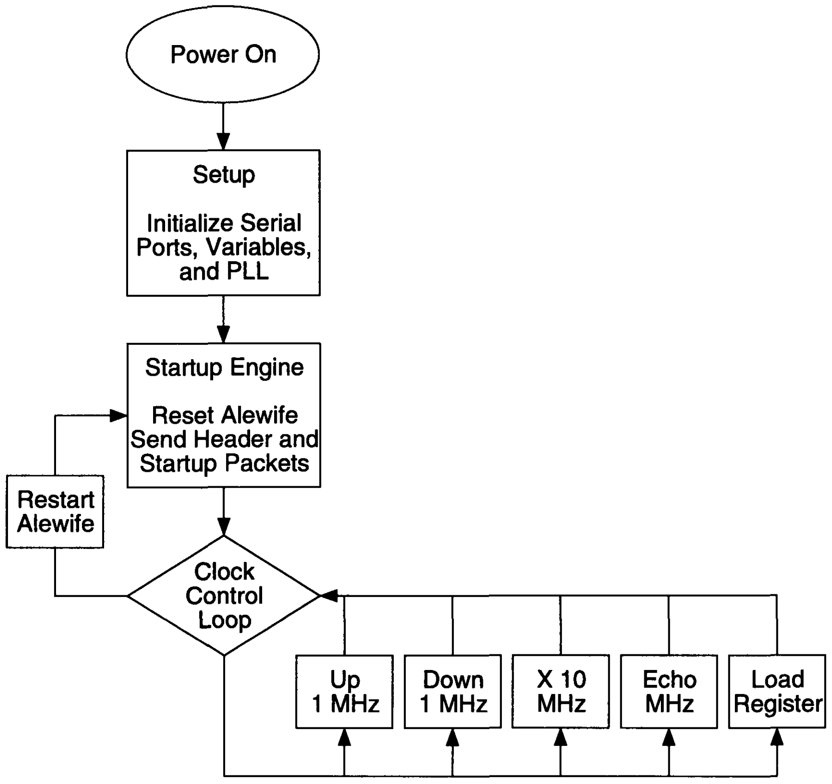

Figure 4-1: Software Flow Chart

Certain I/O ports need to be reconfigured from their default operation in order to perform as desired. This is usually the case where a particular pin can act as both an input and output. Changing parameters in the control registers allow the reconfiguration.

As the final setup procedure, the clock synthesizer is initialized with its default frequency. There are three variables which control the performance of the PLL. Two of these values remain constant throughout the operation of the synthesizer. The third is varied to adjust the frequency of the system. In addition to the PLL control, the signal controlling the multiplexer is also initialized to select the proper frequency range.

Startup Engine

The software design for the startup engine involves the implementation of the EMRC protocol. In addition, it includes the development of a standard technique for utilizing the output latches residing on the data bus.

The EMRC protocol calls for a simple handshaking protocol. The startup engine toggles the request line and the EMRC responds with the acknowledge line. The data to be transferred should be in the EMRC latch before the request signal is toggled. On the final byte to be transferred, the tail bit should be set. This signals the EMRC that the transfer is complete and the next request will be the beginning of a new packet.

The request and acknowledge lines are active on each transition, meaning that a change from low to high is a trigger, as well as a change from high to low. This saves time in the handshaking, but it complicates the code a little since the behavior of the instruction set routines used need two separate blocks to handle each direction of the signal transition.

The transfer of data begins with a few standard values. These include the X and Y offsets which determine the target node for the transfer in addition to the standard header and initial address for the node. This information is then followed by the startup information stored in the EEPROM beginning at location 0x1050. The X and Y offsets are both initialized to zero by the setup routine, but can be changed when using the restart function.

This hardware implementation of the startup engine is essentially useless if a good methodology cannot be developed for using the output latches.

The latches are triggered during a store operation to particular addresses by the 68HC11. Data for the store operation is visible on the data bus when the address being accessed is not considered an internal address such as RAM or registers. On writes to internal addresses, the values on the data bus are considered invalid. Therefore, the latches are only useful for capturing data on external address values.

In the implementation, the latches are triggered based on two address bits, the $R/\overline{W}$ signal, and the E clock. The two address bits are A15 and A13, both in the higher eight bits of address so they are not related to internal addresses. The latch for the EMRC data triggers when A15 is high and A13 is low. The other latch, used for supplemental outputs like LEDs and the EMRC tail bit, is triggered when both A15 and A13 are high. In order to generate these bit patterns, the EMRC data latch can be considered to be at the 0xD000 location in memory and the supplemental latch is considered to be at 0xF000. Of course, any addresses with the same values for A15 and A13 will work. In addition to the address bits, the latch triggers are active only when the E clock is high and the $R/\overline{W}$ signal is low. A high E clock represents the data portion of the cycle and the low $R/\overline{W}$ signal signifies a write operation.

Clock Synthesizer

The core of the clock synthesizer routine is a loop which waits for serial input from the RS-232 port. Input can come in many forms as described in Table 2. There are basically two types of inputs expected. The first type is for an interactive user and used mainly for debugging purposes. The other is intended for software control from a host computer and does not provide feedback.

Table 2: Startup Module Control Keys

Key(s)	Effect
=	Echo Frequency
+	Increment 1 MHz
-	Decrement 1 MHz
1-8	Frequency = <Input> X 10 MHz
x	Captures next 24 bits defining the new clock frequency
r	Captures following two bytes as X & Y offsets for Alewife reset

The user functions include displaying the current frequency, changing the frequency to a multiple of 10 MHz, incrementing the frequency by 1 MHz, and decrementing the frequency by 1 MHz. Each of these operations echoes the frequency to the RS-232 port for display on the user's terminal.

The host functions are less interactive and provide more control over the system. The load routine allows the host to load a 16-bit value into the 145170 control register. The load routine first loads an 8-bit flag to check if the clock signal selected should be in the 5 MHz to 20 MHz range or the 20 MHz to 80 MHz range. When the clock synthesizer is in the 20 MHz to 80 MHz range, the 16-bit value is directly related to the actual clock frequency.

The other host function is not directly related to the clock synthesizer, but is a useful addition to the software control. It allows the host to restart the Alewife machine and redefine the X and Y offsets for the reset. This does not cause the clock synthesizer to reset. By redefining the X and Y offsets, it is possible to startup the machine beginning with a different node.

One important aspect of the clock synthesizer is its smooth switching when changing frequencies. However, it is not as smooth when switching from frequencies in the 5 MHz to 20 MHz range to ones in the 20 MHz to 80 MHz range. In order to guarantee a safe frequency switch, it is necessary to make some software adjustments. The primary concern when switching frequencies is a momentary glitch when the clock becomes much faster than expected. In this case, it can be resolved by forcing the clock to be much slower while adjustments are made and then placing the clock at its new frequency.

The situation is most easily seen when switching from a 19 MHz to 20 MHz clock. The 19 MHz clock is generated using a 76 MHz PLL signal which is divided by four. The 20 MHz clock is generated directly by the PLL, and its divide-by-four counterpart is only 5 MHz. The PLL signals and their associated divide-by-four signals are multiplexed together to select the proper signal for the clock. In the transition from 19 MHz to 20 MHz, the 76 MHz PLL signal is enabled by the multiplexer before the transition to 20 MHz has begun. This is a problem especially when the devices using the clock are not able to handle it.

The solution is to force the multiplexer to select the slow signal while changes to the PLL are made. In addition, a short delay is required after changes are made to allow the PLL time to lock in the new frequency. Then, the multiplexer can be set to select the proper signal for the clock. This guarantees the switch from 19 MHz to 20 MHz will actually change to 5 MHz before becoming 20 MHz. Changes from 20 MHz to 19 MHz will first slow to 5 MHz and then increase to 19 MHz. Finally, changes from 20 MHz to 21 MHz will slow to 5 MHz before increasing to 5.25 MHz and then 21 MHz. The slower clock signal during the transition period does not have any negative effects on the devices using the clock.

Chapter 5
Testing

Startup Engine

The startup engine was tested by attempting to download a simple piece of code to an Alewife node. The code causes the node's display to cycle through the hexadecimal numbers. Successful execution of the code is sufficient to fully test the startup engine. Of course, several obstacles were encountered in the process of testing.

The main problem involved the use of the output latches connected to the data bus. Apparently, the latches were not loading the proper values. Through further testing, it appeared the latches were capturing the lower address bits of the next bus operation. The problem was solved when the trigger signal was modified to include the E clock, guaranteeing the trigger would complete while data is still valid. The original trigger did not include the E clock and apparently remained active until the beginning of the next bus cycle.

The other problem is related specifically to the EMRC latch. In the initial hardware implementation, only one address line is used to control the triggers for the two latches. However, it is not sufficient to use only one address bit to select the proper latch since writes to internal addresses are visible externally. As a result, an additional address line is required for decoding the triggers for the two output latches, guaranteeing that the correct values are loaded into the proper latches each time and avoiding all potential conflicts with internal addresses.

Clock Synthesizer

Testing for the clock synthesizer was conducted by attaching the output signals with an oscilloscope and verifying that the signal satisfied the pECL voltage characteristics. In addition, the exact frequency of the signal is compared to the frequency set by the 68HC11 to verify its accuracy. A couple problems were encountered in the testing.

One ECL logic problem in the clock synthesizer was discovered during testing. The problem was the absence of the 5 MHz to 20 MHz clock. The cause was traced back to an error in wiring the 10136 counter. The carry-in of the device should have been connected to -5 V instead of ground to enable the counter to count continuously. Instead, by grounding the input, the counter had been configured to count once and stop.

In addition to the ECL logic problem, the PLL experienced some problems relating to jitter and time-to-lock. The original circuit implemented would not lock to any frequency. The parameters chosen were aimed at minimizing jitter, but the resulting time-to-lock was too large. After selecting new parameters and recalculating component values, time-to-lock is no longer an issue. Unfortunately, the solution to the locking problem is causing the PLL to jitter at certain frequencies. The jitter is most noticeable around 70 MHz and is also slightly visible at 27 MHz. Another evaluation of the PLL parameters is necessary to find the best compromise between the time-to-lock and jitter problems.

Chapter 6
Startup Module Operation

The original intention of the startup module is to be able to startup the Alewife machine independently. This implementation realizes its goal. When power is first applied to the system, the 68HC11 comes out of reset after a slight delay and establishes a default clock frequency. The startup module then pauses approximately one second to allow the Alewife machine to stabilize before attempting communication with the nodes. The startup engine then begins to transfer the startup information to the Alewife node.

After the transfer is complete, the startup module contains the ability to change the clock frequency and restart the machine. In most cases, there will be a host machine connected to the startup module via the RS-232 port. The host has precise control of the clock frequency and can also restart the Alewife machine using a particular node. In addition, the clock frequency can be adjusted by a user from a terminal using only a few keystrokes. The control is not as precise as that of the host interface, but it is sufficient for simple testing.

Chapter 7
Conclusion

Summary

The startup module satisfies the goal of being a stand-alone unit to startup the Alewife machine. Together with a fully functional SCSI node, the VME interface will no longer be necessary. In addition, the frequency range of the clock synthesizer provides a much wider and more useful range for testing the Alewife prototype. This allows for the low-speed examination of some of the details of the Alewife architecture.

Future Work

There are many areas in which the functionality of the startup module, and the environment it supports, can be explored further. The immediate future is the development of the hardware interfaces already built into the startup module such as the power supply controller and the analog port of the 68HC11. The power supply controller can be completely implemented in software for the 68HC11. In addition, many analog applications can be developed using Port E of the 68HC11. A prototype area was included on the startup module for the development of new hardware and the 68HC11 includes many functions designed for the analog interface.

There are also many issues to be examined for the long-term future of the Alewife machine. Some of these include the reliability of the startup module, a redesigned startup sequence to take advantage of the startup module and SCSI interface, and an improved host interface.

Appendix A

File: startup.abl

```
module startup
title 'Startup Module PAL'

    startup device 'p22v10';

"Inputs
    e                           pin 1;
    reset_cir                   pin 2;
    ra4, runa5                  pin 3,4;
    adr13, adr15, r_w           pin 10,11,13;

"Outputs
    run_pal                     pin 14;
    r_pal                       pin 15;
    rom_oe                      pin 16;
    wi_e                        pin 17;
    out_e                       pin 18;
    _reset                      pin 19;

equations

    _reset = reset_cir;
    out_e = adr13 & adr15 & !r_w & e;
    wi_e = !adr13 & adr15 & !r_w & e;
    rom_oe = !(r_w & e);
    r_pal = !ra4;
    run_pal = !runa5;

end
```

Appendix B

Phase-Locked Loop Design Calculations

1. Choose $f_{ref} = 2$ KHz

   This provides good signal resolution

2. $N_{max} = f_{max} / f_{ref} = 80$ MHz / 2 KHz = 40000

   $N_{min} = f_{min} / f_{ref} = 20$ MHz / 2 KHz = 10000

3. Choose $\zeta = 2.0$

   The recommended ζ is 0.5, but to minimize jitter, 2.0 is used.

4. $\omega_n = \omega_n t / t = 3.0 / 0.01$ s = 300 rad/s

   This for a lock-time of 10 ms.

5. $K_\phi = 0.8$ V/rad

   $K_{VCO} = 5.26E7$ rad/s/V

   C = 0.8 X 5.26E7 / 40000 / 300 / 300 / 540000 = 0.021 μF

   Guess R_1 is 540 KΩ (same as original)

   Closest C is 0.01 μF

6. $R_2 = 2\zeta_{min} / \omega_n C = 4 / 300 / 0.01E\text{-}6 = 1.33$ MΩ

   Closest value available is 1 MΩ

Values Used in Implementation

$R_1 = 540$ KΩ

$R_2 = 1$ MΩ

C = 0.01 μF

Slight changes in the selection of parameters may solve the jitter problems.

Appendix C

File: startup.azm

```
! startup.azm: startup module
! wkchan@lcs.mit.edu

              .inc    define                ! Include define.azm

vector        .ent    FFFEh                 ! Reset Vector -> 0200h
              .word   code

data          .ent    0000h                 ! Data at bottom of RAM
saved_N       .word   FFFFh                 ! FFFFh for Data I/O Sanity
slow_clock    .byte                         ! Flag for Slow Clock
X_offset      .byte                         ! X Offset for Alewife Reset
Y_offset      .byte                         ! Y Offset for Alewife Reset
print_temp1   .word                         ! Temp Storage for print_N
print_temp2   .byte                         ! Temp Storage for print_N

code          .ent    0200h                 ! The Real Thing...
              lds     01FFh                 ! Stack Pointer at top of RAM
              jsr     setup                 ! Setup Ports & Stuff
              jsr     engine                ! Startup Engine

! Clock Synthesizer

clock         ldd     clock
              pshd
              ldx     1000h
              jsr     input_serial
              cmpa    RESTART_CODE
              beq     _restart_ptr
              cmpa    PRINT_CODE
              beq     _printf_ptr
              cmpa    LOAD_CODE
              beq     _load_clk
              cmpa    UP_CODE
              beq     _up_clk
              cmpa    DOWN_CODE
              beq     _down_clk
              cmpa    '1'
              beq     _ten_mhz
              cmpa    '2'
              blo     _bogus
              cmpa    '8'
              bls     _num_mhz

_bogus        ldaa    '?'
              jsr     output_serial
              ldaa    CR
              jsr     output_serial
```

```
                ldaa    LF
                jmp     output_serial

_restart_ptr    jmp     _restart

_printf_ptr     jmp     _printf

_load_clk       jsr     input_serial                ! Load Flag
                cmpa    SLOW_CODE
                beq     _slow_set
                ldab    00h
                stab    (slow_clock)
                bra     _load

_slow_set       ldab    01h
                stab    (slow_clock)
_load           jsr     input_serial                ! Load 16-bit value
                tab
                jsr     input_serial
                bra     _update_clk

_ten_mhzldab    01h                                 ! Select slow signal of 40 MHz PLL
                stab    (slow_clock)
                ldaa    '4'
                bra     _calc_N

_num_mhz        ldab    00h
                stab    (slow_clock)
_calc_N         suba    '0'                         ! Calculate N for new frequency
                tab
                ldaa    20
                mul
                ldaa    250
                mul
                bra     _update_clk

_up_clk         ldd     (saved_N)
                addd    500                         ! Increment 1 MHz
                brclr   (x + PORTA), CLK_MASK, _up_more
                addd    1500                        ! In the Slow Zone
                cpd     high_N
                blo     _up_more
                ldab    00h
                stab    (slow_clock)
                ldd     low_N                       ! 20 MHz
_up_more        cpd     high_N
                bhi     _bogus
                bra     _update_clk

_down_clk       ldd     (saved_N)                   ! Decrement 1 MHz
                addd    -500
                brclr   (x + PORTA), CLK_MASK, _down_more
                addd    -1500                       ! In the Slow Zone
                cpd     low_N
                blo     _bogus
                bra     _update_clk
_down_more      cpd     low_N
                bhs     _update_clk
                ldab    01h
```

```
              stab    (slow_clock)
              ldd     high_N - 2000                ! 19 MHz

_update_clk   bset    (x + PORTA), CLK_MASK        ! Update PLL and Echo Frequency
              std     (saved_N)
              jsr     write_N
              ldy     1000h
_update_wait  dey
              bne     _update_wait
              ldab    (slow_clock)
              cmpb    01h
              beq     _printf
              bclr    (x + PORTA), CLK_MASK

_printf       ldd     (saved_N)
              jmp     print_N

_restart      jsr     input_serial                 ! Restart Alewife
              staa    (X_offset)                   ! Store New Offsets
              jsr     input_serial
              staa    (Y_offset)
              jmp     engine                       ! Use Startup Engine

! Setup Startup Module

setup         ldx     1000h                        ! X points at the registers

              ldaa    BAUD_SETUP                   ! Serial Setup
              staa    (x + BAUD)
              ldaa    SCCR2_SETUP
              staa    (x + SCCR2)
              ldaa    2Fh
              staa    (x + PORTD)
              ldaa    DDRD_SETUP
              staa    (x + DDRD)
              ldaa    SPCR_SETUP
              staa    (x + SPCR)

              ldaa    PACTL_SETUP                  ! Port A Setup
              staa    (x + PACTL)

              ldaa    DEF_X_OFF                    ! Set Default X & Y Offsets
              staa    (X_offset)                   !  for Alewife Startup
              ldaa    DEF_Y_OFF
              staa    (Y_offset)

              ldaa    0h                           ! Slow Clock Flag
              staa    (slow_clock)

              bset    (x + PORTA), CLK_MASK        ! PLL Initialization
              clrb
              jsr     write_C
              ldd     0FA0h
              jsr     write_R
              ldd     DEF_F * 500                  ! Default Frequency
              std     (saved_N)
              jsr     write_N
              bclr    (x + PORTA), CLK_MASK
              rts
```

```
! SPI Routines for MC145170

write_R        bclr   (x + PORTD), SS_MASK        ! assert SS*
               clr    (x + SPDR)
_wait          brclr  (x + SPSR), 80h, _wait      ! wait for SPIF

write_N        bclr   (x + PORTD), SS_MASK        ! assert SS*
               staa   (x + SPDR)
_wait          brclr  (x + SPSR), 80h, _wait      ! wait for SPIF

write_C        bclr   (x + PORTD), SS_MASK        ! assert SS*
               stab   (x + SPDR)
_wait          brclr  (x + SPSR), 80h, _wait      ! wait for SPIF

               bset   (x + PORTD), SS_MASK        ! deassert SS*
               rts

! Serial Port Routines

input_serial   brclr  (x + SCSR), 20h, input_serial
               ldaa   (x + SCDR)
               rts

output_serial  psha
_loop          ldaa   (SCSR)
               anda   80h
               beq    _loop
               pula
               staa   (SCDR)
               rts

print_N        brclr  (x + PORTA), CLK_MASK, _print_fast
               ldx    2000                        ! Compute MHz for Slow
               bra    _print_more
_print_fast    ldx    500                         ! Compute MHz for Fast
_print_more    idiv
               std    (print_temp1)               ! Save Remainder
               ldaa   1
               staa   (print_temp2)               ! Leading Digit
               bsr    putdec                      ! Print Quotient

               ldaa   '.'
               jsr    output_serial

               ldd    (print_temp1)               ! Retrieve Remainder
               addd   (print_temp1)               ! Multiply by 2
               addd   1000                        ! 3 digits
               xgdx
               clr    (print_temp2)
               bsr    putdec                      ! Print Remainder

               ldy    _print_units                ! Load "MHz"
               bsr    puts                        ! Print Units
               ldaa   CR
               jsr    output_serial
               ldaa   LF
               jmp    output_serial
```

```
_print_units    .asciz " MHz"

putdec          xgdx
                ldx     10
                idiv
                cpx     0
                beq     _leading
                pshb
                bsr     putdec
                pulb
_print          ldaa    '0'
                aba
                jmp     output_serial
_leading        ldaa    (print_temp2)
                bne     _print
                rts

puts            ldaa    (y)
                beq     _exit_puts
                iny
                jsr     output_serial               ! Send Character
                bra     puts
_exit_putsrts

! Startup Engine

engine          bclr    (x + PORTA), WIR_MASK! Clear WIR

                ldy     FFFFh                       ! Wait 1 Second for
_rwait1         dey                                 !  Alewife Power-On
                bne     _rwait1
                ldy     FFFFh                       ! Each Loop = 0.2 s
_rwait2         dey
                bne     _rwait2
                ldy     FFFFh
_rwait3         dey
                bne     _rwait3
                ldy     FFFFh
_rwait4         dey
                bne     _rwait4
                ldy     FFFFh
_rwait5         dey
                bne     _rwait5

                bset    (x + PORTA), MRC_R_MASK     ! Send Reset
                ldy     0200h
_rloop1         dey
                bne     _rloop1
                bclr    (x + PORTA), MRC_R_MASK     ! Clear Reset
                ldy     0200h
_rloop2         dey
                bne     _rloop2

                ldaa    00h                         ! Clear WIT
                staa    (OUT_LATCH)

                ldaa    (X_offset)                  ! Send X Offset
                staa    (MRC_LATCH)
                bset    (x + PORTA), WIR_MASK
```

```
_xwait         brclr  (x + PORTA), WIA_MASK, _xwait

               ldaa   (Y_offset)                  ! Send Y Offset
               staa   (MRC_LATCH)
               bclr   (x + PORTA), WIR_MASK
_ywait         brset  (x + PORTA), WIA_MASK, _ywait

               ldy    base_ROM                    ! Send Startup Data
                                                  ! Initial Location = base_ROM
_eloop         cpy    high_ROM                    ! Branch _tail on Last Location
               beq    _tail
               ldaa   (y)
               staa   (MRC_LATCH)
               brclr  (x + PORTA), WIR_MASK, _rlow

_rhigh         bclr   (x + PORTA), WIR_MASK       ! If WIR High, Change to Low
_rhigh_wia     brset  (x + PORTA), WIA_MASK, _rhigh_wia
               bra    _wia

_rlow          bset   (x + PORTA), WIR_MASK       ! If WIR Low, Change to High
_rlow_wia      brclr  (x + PORTA), WIA_MASK, _rlow_wia

_wia           iny                                ! Next Location
               bra    _eloop

_tail          staa   (MRC_LATCH)                 ! Last Location
               ldaa   WIT_CODE_HIGH               ! Set WIT
               staa   (OUT_LATCH)
               brclr  (x + PORTA), WIR_MASK, _t_rlow

_t_rhigh       bclr   (x + PORTA), WIR_MASK       ! If WIR High, Change to Low
_t_rhigh_wia   brset  (x + PORTA), WIA_MASK, _t_rhigh_wia
               bra    _tail_end

_t_rlow        bset   (x + PORTA), WIR_MASK       ! If WIR Low, Change to High
_t_rlow_wia    brclr  (x + PORTA), WIA_MASK, _t_rlow_wia

_tail_end      ldaa   WIT_CODE_LOW                ! Clear WIT
               staa   (OUT_LATCH)
               rts                                ! Reset Complete

! Header for Alewife Startup

boot           .ent   1040h
               .long  42941200h                   ! BOOTHEADER
               .long  00030000h                   ! BOOTADDR
               .long  0                           ! Boot Offset
               .long  0                           ! Padding
                                                  ! Code should start at 1050h
```

File: define.azm

```
! define.azm: definitions for startup.azm
! wkchan@lcs.mit.edu

! Port A Inputs

WIA_MASK       .equ   80h                     ! West In Acknowledge

! Port A Outputs

CLK_MASK       .equ   08h                     ! Clock Slow / Clock Fast *
MRC_R_MASK     .equ   10h                     ! Alewife Reset
WIR_MASK       .equ   40h                     ! West In Request

! Latch Outputs

WIT_CODE_LOW   .equ   18h                     ! West In Tail Low
WIT_CODE_HIGH  .equ   19h                     ! West In Tail High

! Clock Control

LOAD_CODE      .equ   'x'                     ! Load 24 bits (SLOW + 16 bits N)
SLOW_CODE      .equ   's'                     ! For LOAD_CODE
UP_CODE        .equ   '+'                     ! Up 1 MHz
DOWN_CODE      .equ   '-'                     ! Down 1 MHz
RESTART_CODE   .equ   'r'                     ! Restart (16 bits for offsets)
PRINT_CODE     .equ   '='                     ! Print Frequency to Serial Port
DEF_F          .equ   30                      ! Default Frequency
low_N          .equ   10000                   ! N minimum
high_N         .equ   40000                   ! N maximum

! Latch Control

MRC_LATCH      .equ   D000h                   ! MRC Latch Address
OUT_LATCH      .equ   F000h                   ! Output Latch Address

! Port A Control

PACTL_SETUP    .equ   08h                     ! Direction of PA3 = Output

! Serial Control

SS_MASK        .equ   20h                     ! bit-5 is used as an output
BAUD_SETUP     .equ   30h                     ! /13 and /1 = 9600 @ 8MHz in
DDRD_SETUP     .equ   3ah                     ! SPI and SCI set up.
SCCR2_SETUP    .equ   0ch
SPCR_SETUP     .equ   5ch                     ! changed cpol to 1

! Startup Engine Control

base_ROM       .equ   1040h                   ! Base Address of Boot ROM
high_ROM       .equ   203Fh                   ! High Address of Boot ROM
DEF_X_OFF      .equ   0                       ! Default X Offset
DEF_Y_OFF      .equ   0                       ! Default Y Offset
```

File: init.azm

```
! init.azm: definitions for MC68HC11E1

! HC11 control/status register definitions:

REG_BASE       .ent   01000h                  ! start of registers
PORTA          .byte
               .blkb  1                       ! reserved
PIOC           .byte
PORTC          .byte
PORTB          .byte
PORTCL         .byte
               .blkb  1                       ! reserved
DDRC           .byte
PORTD          .byte
DDRD           .byte
PORTE          .byte
CFORC          .byte
OC1M           .byte
OC1D           .byte
TCNT           .word
TIC1           .word
TIC2           .word
TIC3           .word
TOC1           .word
TOC2           .word
TOC3           .word
TOC4           .word
TIC4                                          ! synonym
TOC5           .word
TCTL1          .byte
TCTL2          .byte
TMSK1          .byte
TFLG1          .byte
TMSK2          .byte
TFLG2          .byte
PACTL          .byte
PACNT          .byte
SPCR           .byte
SPSR           .byte
SPDR           .byte
BAUD           .byte
SCCR1          .byte
SCCR2          .byte
SCSR           .byte
SCDR           .byte
ADCTL          .byte
ADR1           .byte
ADR2           .byte
ADR3           .byte
ADR4           .byte
BPROT          .byte
               .blkb  3                       ! reserved
OPTION         .byte
COPRST         .byte
```

```
PPROG          .byte
HPRIO          .byte
INIT           .byte
TEST1          .byte
CONFIG         .byte

! character definitions

XON            .equ   11H                          ! XON (^Q)
XOFF           .equ   13H                          ! XOFF (^S)
LINEABORT      .equ   18H                          ! abort input line (^X)
SYNCABORT      .equ   7                            ! abort input expression (^G)
ASYNCABORT     .equ   3                            ! recover from infinite loops (^C)
HARDABORT      .equ   19H                          ! restart user system (^Y)
EOF            .equ   4                            ! end-of-file for host (^D)
TAB            .equ   9
SPACE          .equ   32
CR             .equ   13
LF             .equ   10
BS             .equ   8
DEL            .equ   7FH
EOL            .equ   0A0DH                        ! CR/LF
QUOTE          .equ   34                           ! double quote
SQUOTE         .equ   39                           ! single quote
LPAREN         .equ   '('
RPAREN         .equ   ')'
DOT            .equ   '.'

! handy definitions

NIL            .equ   0

               .macro pshd
               pshb                                ! low-order, high-address
               psha
               .endm
               .macro puld
               pula
               pulb
               .endm
```

References

1 Agarwal, Anant, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. *The MIT Alewife Machine: A Large-Scale Distributed Memory Multiprocessor.* Cambridge, MA: Laboratory for Computer Science, Massachusetts Institute of Technology, 1991.

2 *Second-Generation Multicomputer Mesh Routing Chip (MRC).* Pasadena, CA: Submicron Systems Architecture Project, Department of Computer Science, California Institute of Technology, 1992.

3 Muntz, Gary. *Alewife Clock Synthesizer Design.* Cambridge, MA: Laboratory for Computer Science, Massachusetts Institute of Technology, 1993.

4 *M68HC11 Reference Manual.* Rev. 3. Motorola, Inc., 1991

5 *MC68HC11E9 HCMOS Microcontroller Unit.* Motorola, Inc., 1991.

6 *CMOS Application-Specific Standard Digital-Analog Integrated Circuits.* Third Printing. Motorola, Inc., 1991.

7 *MECL Integrated Circuits.* 4th ed. Motorola, Inc., 1989.

8 *ECL Logic and Memory Data Book.* Hitachi America Ltd., 1985.

9 *The 41 Series of High-Performance Line Drivers, Receivers, and Transceivers - Data Book and Designers Guide.* AT&T Microelectronics, 1991.

10 *Power IC's Databook.* National Semiconductor, Inc., 1993.

11 *1993 New Releases Data Book Volume II.* Maxim, Inc., 1992.