LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

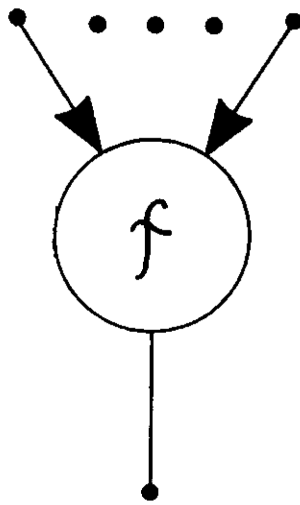
A Processor Module for Data Flow Computer Development

Computation Structures Group Memo 176
May 1979 21 March 1986

Ephraim M. Vishniac

Thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in the Department of Electrical Engineering and Computer Science, MIT.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139



Frontispiece: A Data Flow Operator

Table of Contents

Title	page 1
Abstract	2
Frontispiece	3
Table of Contents	4
Table of Figures	5
Table of Tables	6
Introduction	7
Design Objectives	11
Basic Design Decisions	11
Processor Module Overview	12
Arithmetic and Logic Unit	15
Status and Shift Control Unit	15
Sequence Control	18
Input/Output	20
Main Memory	20
Timing	23
Microcode Format	24
Sample Microcode	26
Microprogram Loading	26
Alternatives	33
Summary	34
Appendix A: Am2903 Instructions	46
Appendix B: Am2904 Instructions	49
Appendix C: Am2910 Instructions	51
Appendix D: Sample Microcode Commentary	52
Bibliography	55

Table of Figures

Frontispiece page 3

1. A Data Flow Computer 8

2. Reset Signalling 9

3. A 2x2 Router Module 10

4. Processor Module Block Diagram 13

5. ALU Block Diagram 16

6. Status and Shift Control Unit 17

7. Microprogram Sequencer and Address Selection Logic 19

8. I/O Block Diagram 21

9. Main Memory and Address Registers 22

10. Clock Cycle 23

11. Microcode Format 25

12. Sample Microcode 27

13. Program Loading Logic 31

14. Program Loading Timing 32

15. ALU Details 35

16. Status and Shift Control Details 36

17. Sequencer Details 37

18. I/O Details 38

Table of Tables

Table of Contents	page 4
Table of Figures	5
Table of Tables	6
Table 1. Microcode-Derived Signals	40
Table 2. Microcode Field Definitions	41
Table 3. Micro-Instruction Types	44

Introduction: Data Flow Computation

A computer based on data flow principles executes instructions in response to the arrival of their operands. Thus there need be no sequential control flow such as one finds in a conventional computer. A consequence of this is the possibility of highly parallel operation, given a suitable machin architecture and an appropriately expressed program.

Member of the Computation Structures Group are currently working to provide both of these. A broad introductio to their work may be found in [1]; this paper will pursue a relatively narrow question of hardware implementation.

The design of the data flow computer currently under development is shown schematically in figure 1. Each of the cell blocks shown contains some number of cells, together with a single bolock manager. These cells contain templates of operation packets, the units from which data flow programs are constructed. Each cell contains an operator, space for operands, and a list of result destinations. The block manager is responsible for discovering which cells are enabled, i.e. which operators have received all their operands.

Whenever the block manager finds such a cell, it must take action to see that the specified operation is performed. Rather than execute the operation itself, it makes a completely self-describing packet from the cell. It then dispatches this packet into its associated arbitration network.

The purpose of the arbitration networks is to route operation packets, according to the operators they contain, to appropriate processors. Since the packets are completely self-

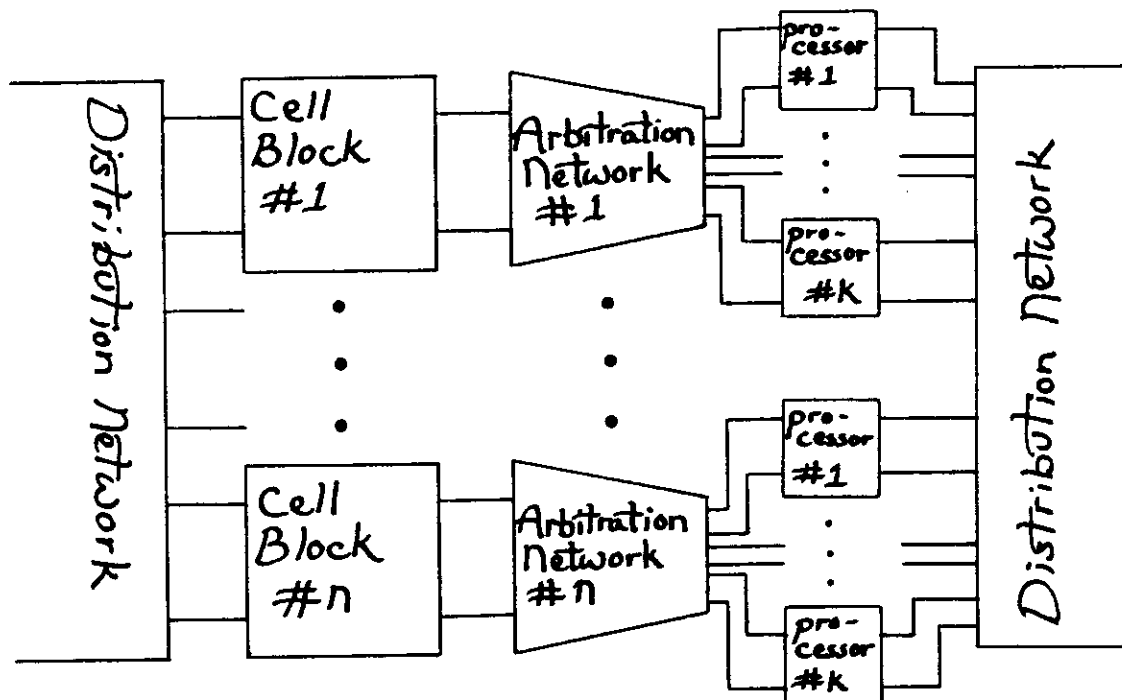


Figure 1: The major components of a data flow computer. The n cell blocks are identical, as are the n arbitration networks and the n sets of k processors. Each of the k processors within a set is different, however.

describing, the arbitration networks are delivering to the processors a stream of executable instructions: operators, their operands, and the addresses of cells for which the present results will become operands.

After executing the operation specified by a packet, a processor uses the result of the operation and the destination addresses to create result packets. These are dispatched into the distribution network, which sorts them according to their destinations.

The purpose of the distribution network is to effect the distribution of results, which will become the operands of successors of the cell just processed. The network delivers result

packets to appropriate cell blocks; the block managers distribute results within their respective blocks and check for newly enabled cells.

Two things are implicit in the above discussion: one is the use of local control to achieve concurrent operation; the other is the specialization of processors. Both of these present problems for implementation: local control calls for coordination of asynchoronous activities; specialization suggests difficulties in hardware design.

The problem of coordination is solved by use of a uniform communication scheme. All communication between sections is by transmission of packets, and all packets are sent in a single byte-serial format. Transmission of each byte is coordinated by reset signalling (figure 2); an additional data line is used to mark the final byte of each packet.

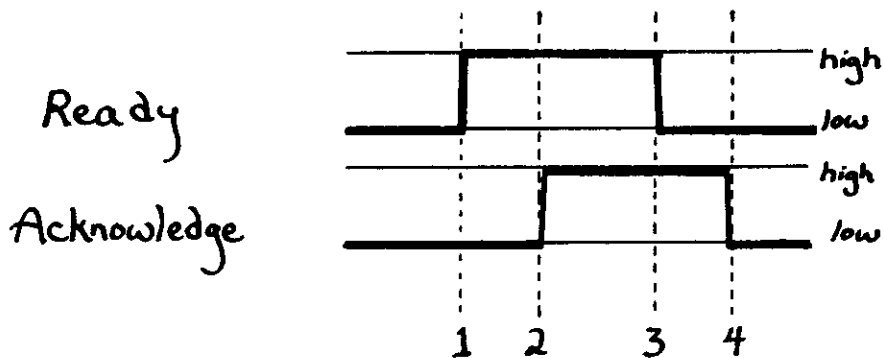


Figure 2: Reset signalling. Data is valid from the time the sender signals "ready" (1) until the receiver signals "ackowledge" (2). Signalling events must occur in the order shown, but may take any length of time.

The difficulties of specialization are attacked by stressing the similarities, rather than the differences, berween funtional units. Both the arbitration and distribution networks, for instance, route packets according to information contained in those packets. Both, in fact, can be built from the same basic building block, a module call a 2x2 router (figure 3). Such a module recieves packets on its two input ports and re-transmits them on its two output ports. Rather than simply pass them straight through, however, it chooses the path of each packet according to the first bit of its first byte. If the desired path is blocked by another packet already in progress, the router delays the new packet until its path is free. Such a router has already been designed and partially built by John Redford (see [2]).

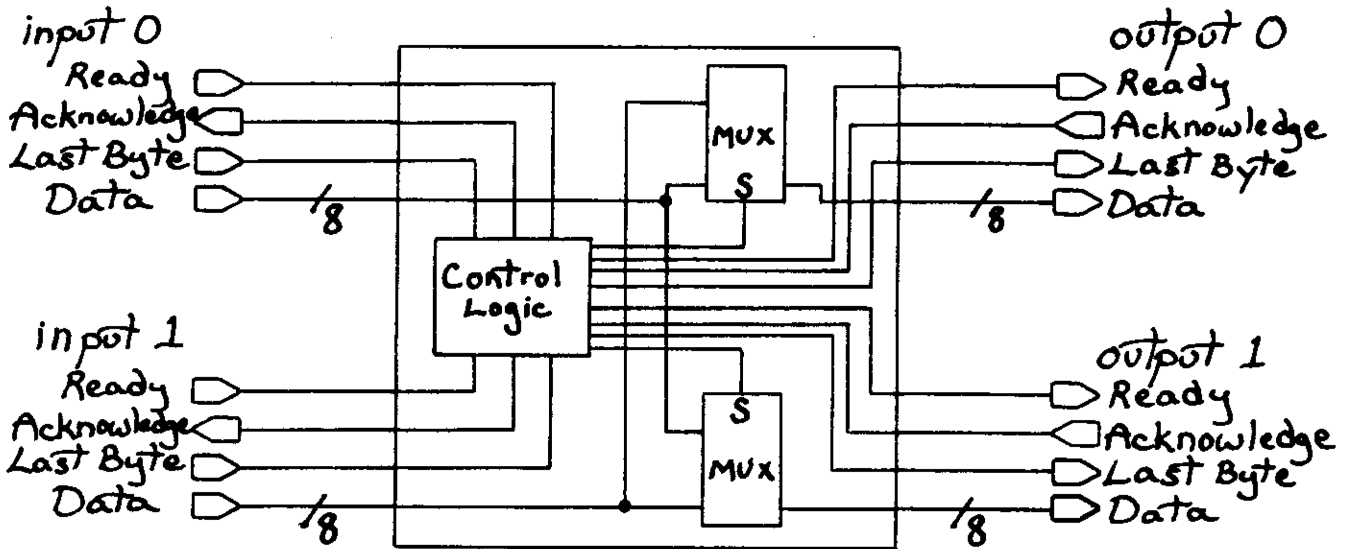


Figure 3: A 2x2 router module. Packets received on the two input ports are routed according to the first bit of their first byte.

Similarly, the cell blocks and all the processors have in common the need for some memory for cell or packet storage, and some processing capability for cell management or instruction execution. Also, processors and cell blocks will both send and receive packets to and from networks built of 2x2 routers: identiacl input/output facilities seem called for.

With these similarities in mind, the purpose of the project was to design a general-purpose processor module. This module must be suitable for use in a packet communication architecture and must be programmable as a cell block or as any one of several specialized types of instruction processors.

Design Objectives

As mentioned above, the basic requirements for the processor module were programmability and some capacity for packet communication. For a cell block, the program will involve both logic operations (such as setting and testing of flags), and integer arithmetic. The program for an instruction processor might additionally involve signed multiplication and division, and "floating point" operations. So, the processor hardware should facilitate all of these operations. Also, I/O handling should not be too cumbersome: the processor should be able to devote most of its time to computation.

More general design objectives were speed (i.e. rapid program execution), low chip cound, and moderate cost.

Basic Design Decisions

Two basic approaches to processor module design were considered: use of a single-chip processor; and use of bit-slice

components. A single-chip processor presented advantages mostly in ease of design. A module based on such a chip would be quite simple, cheap, and could make some use of pre-existing software for the processor chip chosen. Unfortunately, the fixed instruction set might prove awkward and slow for packet communication. Use of bit-slice components, although more complicated, presented advantages in speed and flexibility.

Processor width was the next major issue considered. Greater width (i.e. wider data paths) offered the potential for greater computational speed, but data handling was simplest with a processor whose width matched the I/O port width: eight bits. To overcome the limitation this imposes on address space, the address register was split into two separate registers of up to eight bits each. This provides up to sixteen bits of address at the cost of one additional processor cycle for register loading.

Finally, the processor module was designed with the idea that it would be programmed in microcode, not that the microcode would support some higher-level instruction set. This was done to simplify the processor and to preserve its advantages in speed and flexibility over a microprocessor-based module.

Processor Module Overview

In normal operation, a processor module uses exactly the same external connections as the 2x2 router shown in figure 3. (Additional connections, used for microprogram loading, are explained in the section on program loading, below.) Figure 4 illustrates the internal structure of the processor module. This section presents basic information about the major data and control paths; following sections discuss the various parts

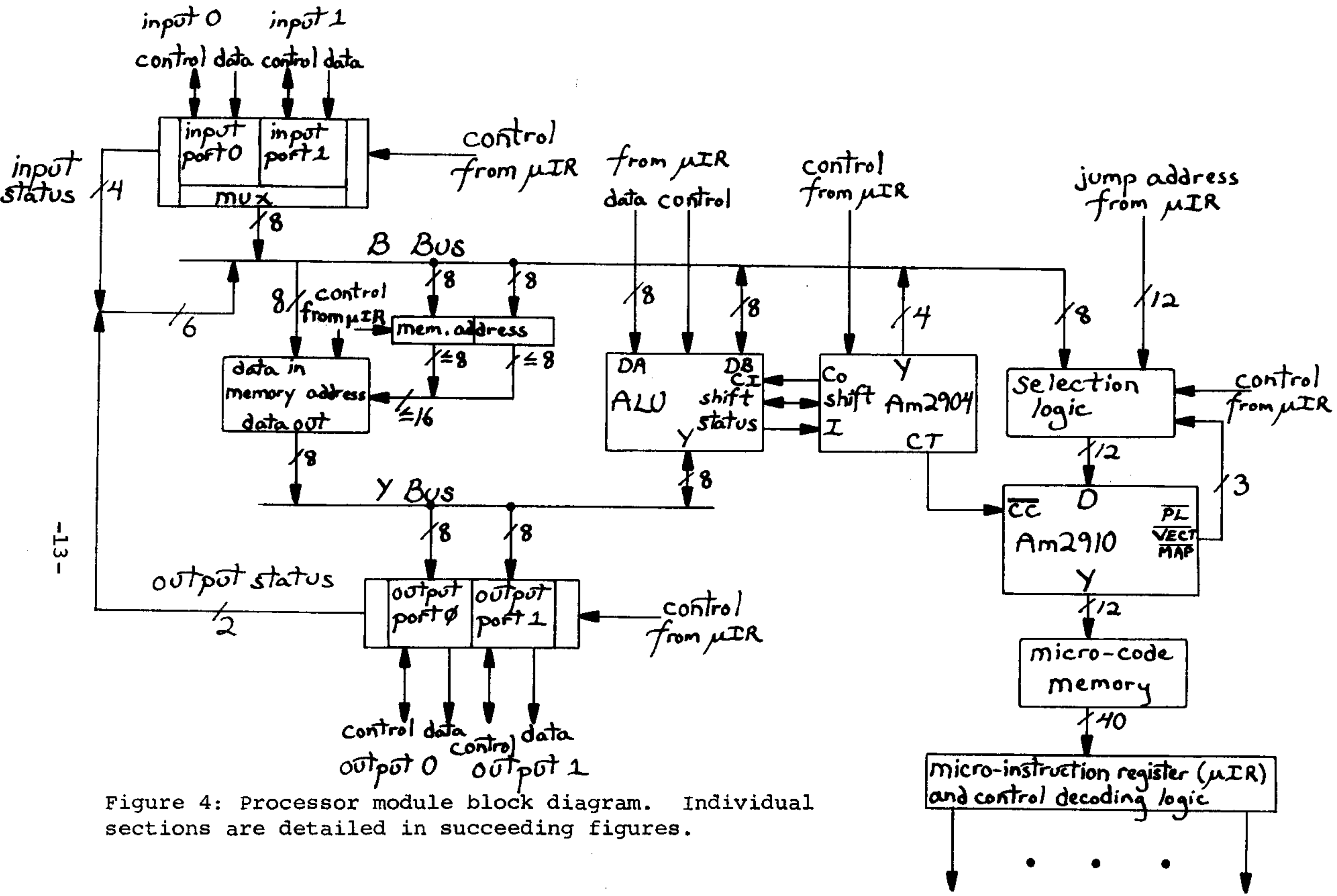


Figure 4: Processor module block diagram. Individual sections are detailed in succeeding figures.

of the processor in detail.

The module receives packets (byte-by-byte) via two input ports, each of which is connected to the "B" bus. The B bus can be read both by the main memory and by the ALU, so incoming packets can be built up in memory or processed immediately.

Besides the input ports, the ALU, the status register, and the I/O status lines are also sources of data for the B bus. In addition to the main memory and the ALU, the two parts of the address register and the sequencer selection logic are B bus destinations. The memory address registers are loaded from the ALU; the status register and I/O status can be read by the ALU. Certain sequencer instructions use data from the B bus for multi-way dispatches.

The module transmits packets (again, byte-by-byte) via the two output ports, each of which is connected to the "Y" bus. Both the main memory and the ALU can place data on the Y bus; this bus is also used to transmit data from the memory to the ALU. As implied above, the B bus is used to transmit data from the ALU to the memory.

The module is controlled during each instruction cycle by the contents of the micro-instruction register. This register is loaded at the beginning of each cycle with the instruction selected by the sequencer during the previous cycle. The sequencer selects the next micro-instruction using not only the current instruction, but also information from the status register and from the B bus if appropriate.

Arithmetic and Logic Unit

The ALU consists of two Am2903 4-bit processor slices. The Am2903 was chosen for its flexible architecture and extensive instruction set. The Am2903 contains sixteen general purpose registers, one special register, and selection logic permitting the use of a wide variety of operand sources for computations (figure 5). Beyond the standard arithmetic and logic instructions, it has operations useful in signed multiplication and division as well as in floating point arithmetic (see Appendix A).

The various control and address lines of the ALU ($\overline{EA}$, $\overline{OE_Y}$, $\overline{IEN}$, I_0-I_8, $\overline{OE_B}$, Aadr, and Badr) as well as the direct data bits (Direct Data 0-7) are all derived from the current micro-instruction. The four status outputs (Z, OVR, N, and CO) go to the status and shift control unit, which is the source of the carry in (CI) signal. The four shift I/O lines (SIO_0, SIO_7, QIO_0, and QIO_7) are also connected to the status and shift control unit. Finally, the DB and Y I/O pins of the Am2903's are connected to the corresponding B and Y bus lines. Thus, these busses are actually extensions of ALU internal busses.

Status and Shift Control Unit

The processor status and shift control unit consists of a single Am2904 chip. The Am2904 contains two four-bit status registers (only one of which is used in this design), condition code generation logic, shift connection logic, and carry in generation logic. The effective configuration of the Am2904 is shown in figure 6; unused portions and permanently set control

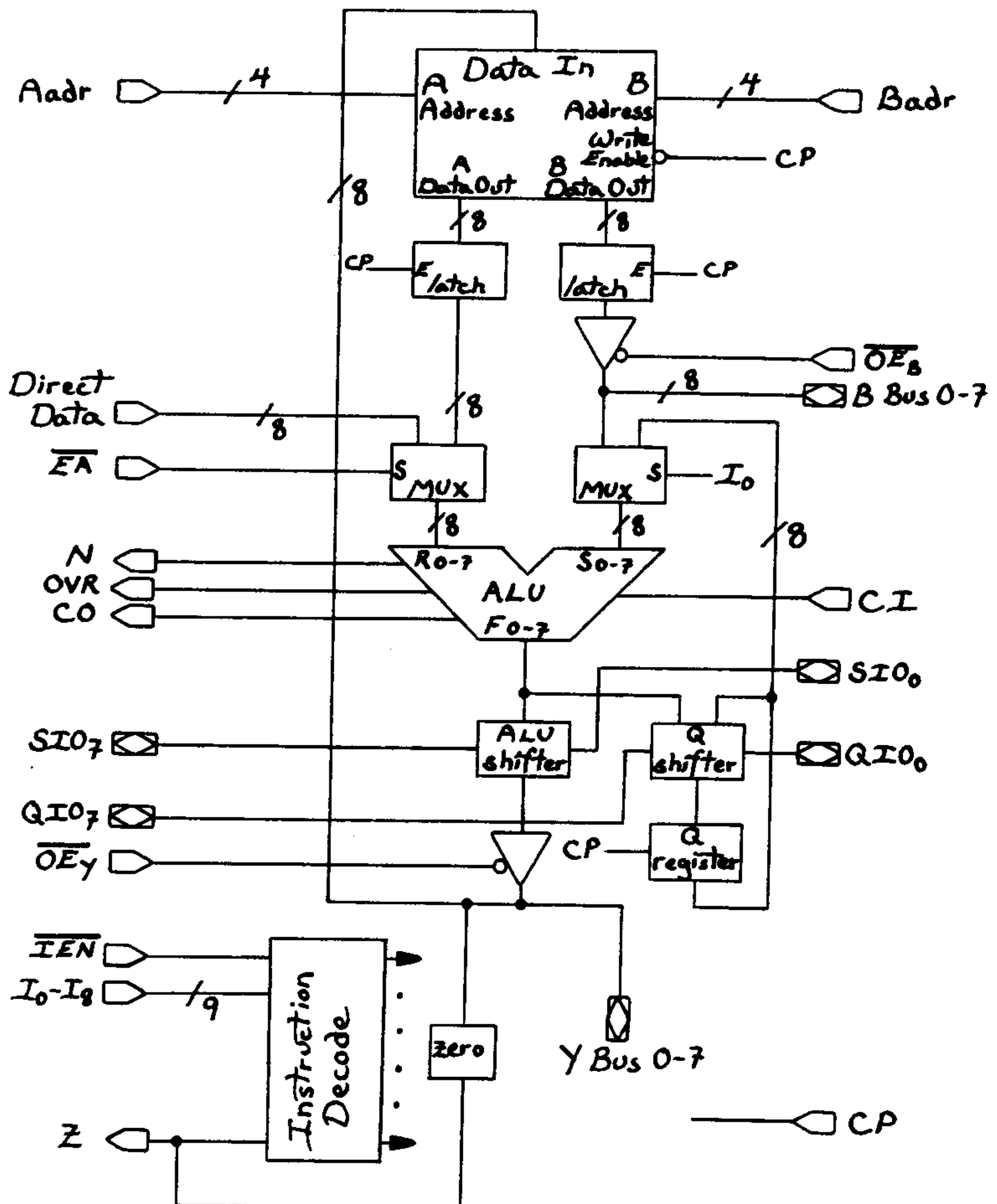


Figure 5: Internal organization of the arithmetic and logic unit (ALU). The ALU is built from two Am2903 chips; details of their arrangement are shown in figure 15.

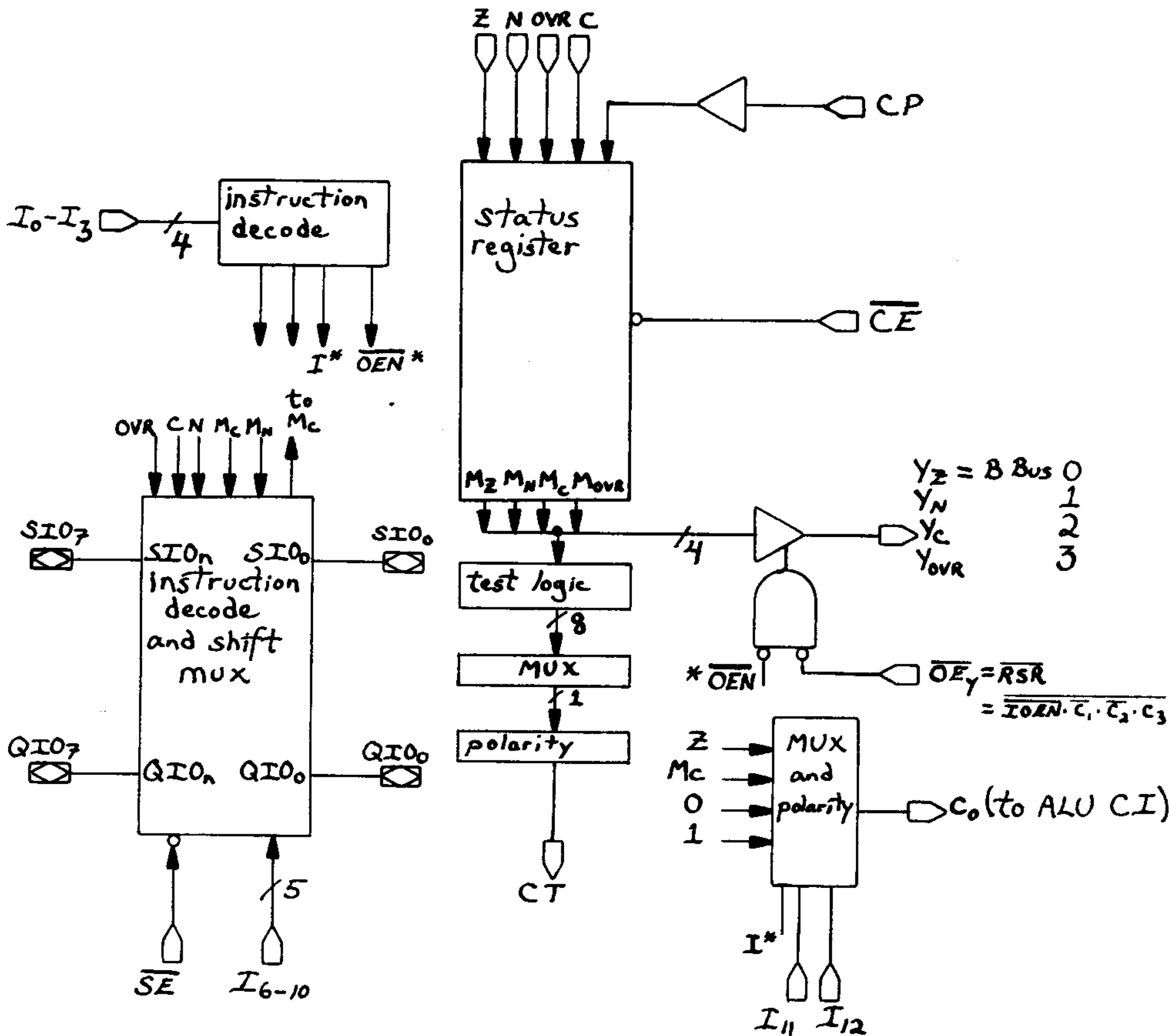


Figure 6: Internal arrangement of the status and shift control unit. This unit is built from a single Am2904 chip, as shown in figure 16.

lines are not indicated.

The various control signals to the Am2904 (I_0-I_3, I_6-I_{10}, I_{11}, I_{12}, $\overline{SE}$, $\overline{CE}$, and $\overline{OE_Y}$) are all derived from the current micro-instruction (see Appendix B for the Am2904 command set). The four status inputs (Z, N, OVR, and C) come from the ALU; the four status outputs (Y_Z, et c.) are connected to the B bus. The single condition output (CT) goes to the condition code input of the sequencer. Finally, the four shift I/O pins are connected to the corresponding ALU pins.

Sequence Control

The sequence of microprogram execution is controlled by an Am2910 Microprogram Controller together with some external selection logic (figure 7). The single-chip sequencer provides for a twelve bit micro-adress and includes a program counter register, a five deep push/pop address stack, and a register/-coünter for loop control.

The Am2910 instruction (I_0-I_3) and condition code enable ($\overline{CCEN}$) come from the current micro-instruction, as does the register load ($\overline{RLD}$) signal used by both the Am2910 and the selection logic. The condition code input ($\overline{CC}$) is the Am2904 CT output. The "D" inputs (D_0-D_{11}) are used for fixed jumps and register loading; these come from the microcode. The B bus is used for variable jumps (multi-way dispatches) and variable register loads. (See Appendix C for the complete sequencer command set.) The reset signal ($\overline{RST}$), an externally generated signal used to reset the entire processor, forces the Am2910 instruction to zero when active.

Figure 7: Microprogram sequencer and address selection logic. The sequencer is an Am2910 unit; details of its connections are shown in figure 17.

Input/Output

The input and output ports (figure 8) are built from standard logic chips. JK flip-flops are used to hold the state of the port and to carry out the external signalling; only a summary of the port state is available to the processor. For each input port, there are two status lines, one to indicate that the last byte of a packet is ready and one to indicate that any other byte of a packet is ready. For output ports, the single status line is used to indicate that the port is clear, i.e. that a byte can be sent.

Each input port simply buffers the current byte onto the B bus when read, while each output port has a register to hold the current byte. Reading from an input port also generates an acknowledge signal to the sender as part of the reset signalling. Similarly, writing to an output port sets the ready signal to the receiver. For both types, the rest of the "handshake" sequence requires no processor control.

Apart from the external communication signals, all I/O control signals come from the current micro-instruction. The I/O status signals aren't used directly as processor control signals, but can be used by the ALU to generate jump conditions or by the sequencer to control multi-way dispatches.

Main Memory

The main memory for the processor and the associated memory address registers are shown in figure 9. Since the size and access characteristics of the memory will depend on the specific application of the module, the memory itself is shown only schematically. Data is written into memory from the B bus and read

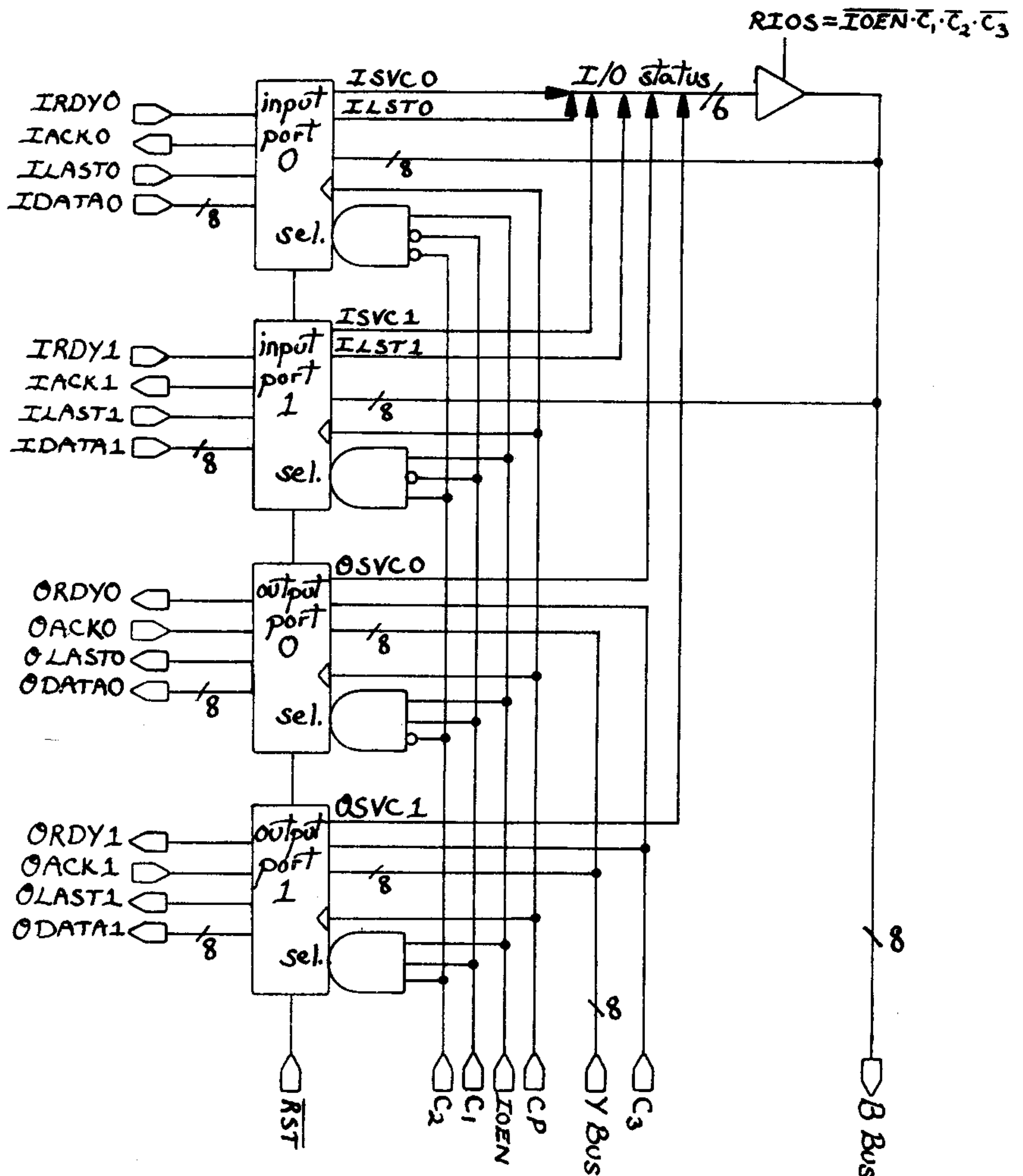


Figure 8: Input and output ports. The detailed design of the I/O ports, built from standard logic chips, is shown in figure 18.

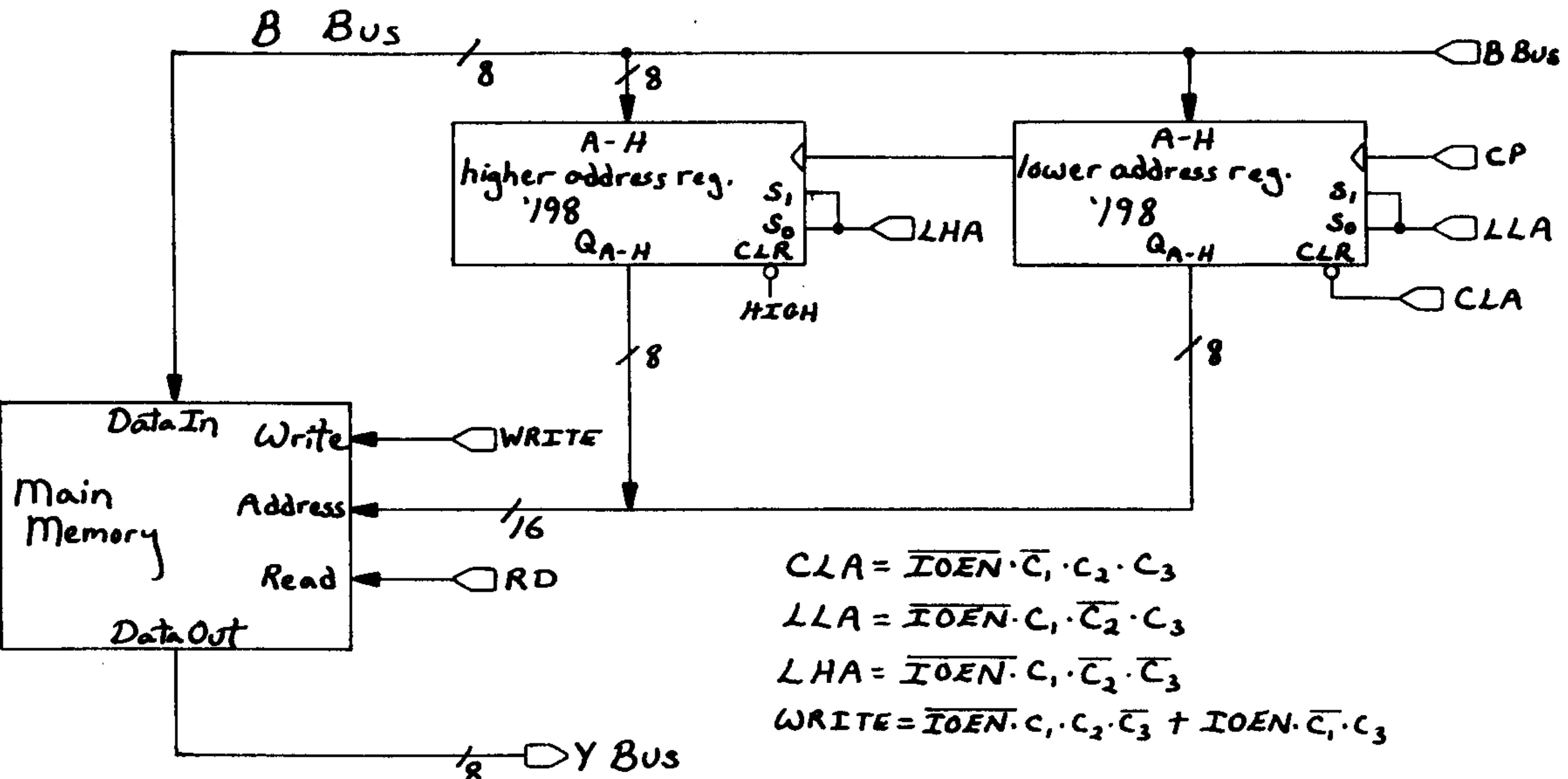


Figure 9: Main memory and address registers. Since memory will depend on the particular application, main memory is shown only schematically.

from memory onto the Y bus. So, data can come from either the input ports or the ALU, and can go to the ALU or to either output port. All control signals for the memory and memory address registers come from the current micro-instruction.

Timing

An instruction cycle begins when the clock signal changes from low to high and ends when it does so again (figure 10). Throughout the processor module, registers, flip-flops, and memory are written to while the clock is low and change their outputs when it goes high again. So, the first part of the clock cycle (when the clock is high) provides for the propagation and settling of signals through combinational logic, while the second part enables writing to memory elements.

Before a new instruction can be selected, all inputs to the sequencer must be stable, including the condition code input. If the condition code depended on the current operation, the selection of the next instruction would be delayed until the ALU output was stable. Instead, the condition code is generated from the contents of the previously loaded status register, so the ALU operation and sequencer operation can proceed in paral-

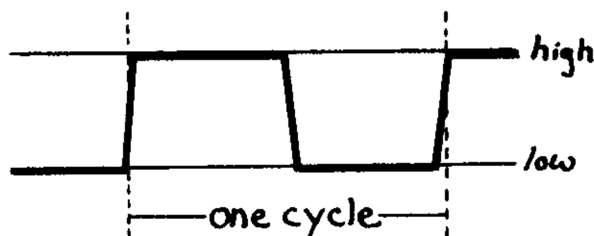


Figure 10: The processor module clock cycle. Most propagation and settling delays occur while the clock is high; all memory elements are written while the clock is low or at the rising edge.

lel. One consequence of this is that conditional branches can be made only on pre-tested conditions, but the speed advantage of this architecture more than offsets the inconvenience. This arrangement, known as a one-level pipeline based architecture, is discussed and compared with other architectures in [3].

Microcode Format

The microcode is the source of the overwhelming majority of control signals in the processor, as well as of some data signals. Signals derived from the current micro-instruction are summarized in table 1. To specify each of these signals independently in each micro-instruction would have been not only expensive (in terms of microcode memory), but also unnecessary.

Figure 11 illustrates how the microcode format was condensed. First, fields that were unlikely to be specified in the same instruction were overlapped. A full twelve-bit jump address, for example, is fairly rare. Direct data removes the need for an "A" address (Aadr field), and is unlikely to be combined with a left or right shift. So, the jump address, direct data, shift mux, and Aadr fields were combined as shown.

Another saving came from the encoding of some mutually dependent control signals. A number of signals affect the state of the B bus, but relatively few combinations of them are useful. So, control of the B bus is exercised through the encoded "C" field and the I/O enable (IOEN) bit, rather than by control of the individual signals invlolved. By these methods, the microcode width was substantially reduced with only a minor loss of flexibility.

	1	5	9	13	22	28	32		36 40
0.	'04 inst. I_9 I_6	Aadr A_3 A_0	Badr B_3 B_0	'03 inst. I_8 I_0	'04 inst. $I_{12} I_{11} I_3$ I_0	'10 inst. I_3 I_0	CCEN	C C_1 C_2 C_3	IOEN RD CE IEN EA
1.						*			0
2.	direct data DA_7	DA_0		no shifts		*			1
3.	D_{11}	D_4		no shifts		JMAP 2			0
4.	D_{11}	D_4		no shifts		CJV 6			0
5.	D_{11}		D_0	C 8 0		**			1 1 1
6.	any of above							0 port write	1 0
7.	any of above							1 port last	1
8.	D_{11} D_8	Aadr	Badr	no shifts		*		1 1 1	0
	1	5	9	13	22	28	32		36 40

*Any of 8, A, D, or E (hexadecimal). Also 4, if register/counter contents need not be saved.

**Not 2, 6, 8, A, D, or E. Only the remaining instructions use a full twelve-bit "D" field.

Figure 11: Microcode format. Line 0 gives the default field names. Line 1 illustrates the restrictions on a general arithmetic instruction; line 2 shows arithmetic with direct data. Lines 3, 4, and 5 show the formats for various kinds of jumps, while 6 and 7 detail the I/O fields. Line 8 gives the format for loading the register/counter with a computed variable.

The meaning of the various microcode fields shown in figure 11 is given in table 2; the different types of micro-instructions are summarized in table 3.

Sample Microcode

Figure 12 shows a sample of microcode for the processor module. The code fragment shown is the service portion of an I/O handler for the module; it polls the I/O ports and services (if necessary) those that are marked as active.

When called (by a "jump subroutine" to DISPATCH), this code reads the I/O status and masks the status with the contents of ALU register zero. If the result is non-zero, indicating that I/O service is needed, an eight-way dispatch to the specific service routines follows. Otherwise, the routine returns to its caller.

Since the two highest-order I/O status bits are necessarily zero, there are actually only six I/O service routines: I0 and I1 for input bytes other than the last of a packet; O0 and O1 for all output bytes; and LI0 and LI1 for input bytes which are last in their respective packets. Each routine either returns to DISPATCH+1 (if more I/O service is required) or to the original caller. A detailed exegesis of the sample microcode is given in Appendix D.

Microprogram Loading

Usually, the microcode for a processor is stored in read-only-memory, because it is only infrequently changed. Since this processor module is designed for the purpose of implementing an entirely new machine, however, it seems likely that the micro-

label	'04 I_{9-6}	Aadr A_{3-0}	Badr B_{3-0}	'03 I_{8-5}	I_{4-1}	I_0	'04 I_{12-11}	I_{3-0}	'10 I_{3-0}	$\overline{CCEN}$	C $C_1 C_2 C_3$	$\overline{IOEN}$	RD	$\overline{CE}$	$\overline{IEN}$	$\overline{EA}$	Comments
DISPATCH	2	0	1	F	C	0	0	0	PUSH	1	RIOS	0	0	0	0	0	R1 ← R0 ∧ I/O status; load status reg. Push next address.
	—IOSVC—		1	C	8	0	0	4	CJV	0	2	0	0	1	1	0	8-way dispatch if R1 ≠ 0
	2	0	0	C	8	0	0	0	CRTN	1	2	0	0	1	1	0	else return to caller
IOSVC	—	—	IO	C	8	0	0	0	JRP	1	2	0	0	1	1	0	Jump to appropriate I/O service routines
	—	—	I1	|	|	|	|	|	|	1	|			|			
	—	—	O0	|	|	|	|	|	|	1	|			|			
	—	—	O1	|	|	|	|	|	|	1	|			|			
	—	—	LIO	|	|	|	|	|	|	1	|			|			
	—	—	LI1	|	|	|	|	|	|	1	|			|			
IO	2	0	2	F	4	0	1	0	CONT	1	LLA	0	0	1	0	0	lower address ← R2; R2 ← R2 + 1
	2	0	3	C	8	0	0	0	CONT	1	LHA	0	0	1	1	0	higher address ← R3
	—62.		1	F	C	0	0	0	CONT	1	2	0	0	0	0	1	mask bit 0 of R1; load status reg.
	2	0	0	C	8	0	0	4	LOOP	0	001	1	0	1	1	0	mem ← input byte; loop if R1 ≠ 0
	2	0	0	C	8	0	0	0	CRTN	1	2	0	0	1	1	0	else return
I1	...																I1 is identical to IO with the substitution of R4 for R2 and R5 for R3, and the change of "62." (which masks bit 0) to "61." (which will mask bit 1).

Figure 12a: Sample microcode, part 1 of 3.

label	'04 I_{9-6}	Aadr A_{3-0}	Badr B_{3-0}	'03 I_{8-5}	I_{4-1}	I_0	'04 I_{12-11}	I_{3-0}	'10 I_{3-0}	$\overline{CCEN}$	C $C_1 C_2 C_3$	$\overline{OEN}$	RD	$\overline{CE}$	$\overline{IEN}$	$\overline{EA}$	Comments
00	2	0	7	C	8	0	0	0	CONT	1	LHA	0	0	1	1	0	higher address ← R7
	2	0	0	C	8	0	0	0	CONT	1	CLA	0	0	1	1	0	clear lower address
	2	0	10.	F	8	0	0	0	CONT	1	2	0	1	1	0	0	R10 ← mem(R7,0)
	2	0	6	F	4	0	1	0	CONT	1	LLA	0	0	1	0	0	lower addr. ← R6; R6 ← R6+1
	2	6	10.	C	B	0	0	0	CONT	1	2	0	0	0	1	0	R6 ⊕ R10 (test for R6=R10)
	—	—	L00	C	8	0	0	5	CJP	0	2	0	0	1	1	0	if R6=R10 then go to L00.
	—	59.	1	F	C	0	0	0	CONT	1	2	0	0	0	0	1	mask bit 2 of R1
	2	0	0	C	8	0	0	4	LOOP	0	1 0 0	1	1	1	1	0	output ∅ ← mem(R7,R6); loop if R1≠0
	2	0	0	C	8	0	0	0	CRTN	1	2	0	0	1	1	0	else return
L00	—	59.	0	F	C	0	0	0	CONT	1	2	0	0	1	0	1	mask bit 2 of R0
	—	59.	1	F	C	0	0	0	CONT	1	2	0	0	0	0	1	mask bit 2 of R1
	2	0	0	C	8	0	0	4	LOOP	0	1 0 1	1	1	1	1	0	output ∅ ← mem(R7,R6); last byte; loop if R1≠0
	2	0	0	C	8	0	0	0	CRTN	1	2	0	0	1	1	0	else return
01	...																01 is identical to 00 with the substitution of R9 and R8 for R7 and R6, and the change of "59." to "55."

Figure 12b: Sample microcode, part 2 of 3.

label	'04 I_{9-6}	Aadr A_{3-0}	Badr B_{3-0}	'03 I_{8-5}	I_{4-1}	I_0	'04 I_{12-11}	I_{3-0}	'10 I_{3-0}	$\overline{CCEN}$	C $C_1C_2C_3$	IOEN	RD	$\overline{WE}$	$\overline{HEN}$	$\overline{EA}$	Comments
LIO	——	46.	0	F	C	0	0	0	CONT	1	2	0	0	1	0	1	Mask bits 0 and 4 of R0
	2	0	2	F	4	0	1	0	CONT	1	LLA	0	0	1	0	0	lower addr. ← R2; R2 ← R2+1
	2	0	3	C	8	0	0	0	CONT	1	LHA	0	0	1	1	0	higher addr. ← R3
	2	0	0	C	8	0	0	0	CONT	1	0 0 1	1	0	1	1	0	mem(R3,R2) ← input byte
	——	47.	1	F	C	0	0	0	CONT	1	CLA	0	0	0	0	1	mask bit 4 from R1; clear lower addr.
	2	0	2	C	8	0	0	4	LOOP	0	WRITE	0	0	1	1	0	mem(R3,0) ← R2; loop if R1 ≠ 0
	2	0	0	C	8	0	0	0	CRTN	1	2	0	0	1	1	0	else return
LI1	...																LI1 is identical to LI0 with the substitution of R5 and R4 for R3 and R2, and the changes of "46." to "29." and of "47." to "31."

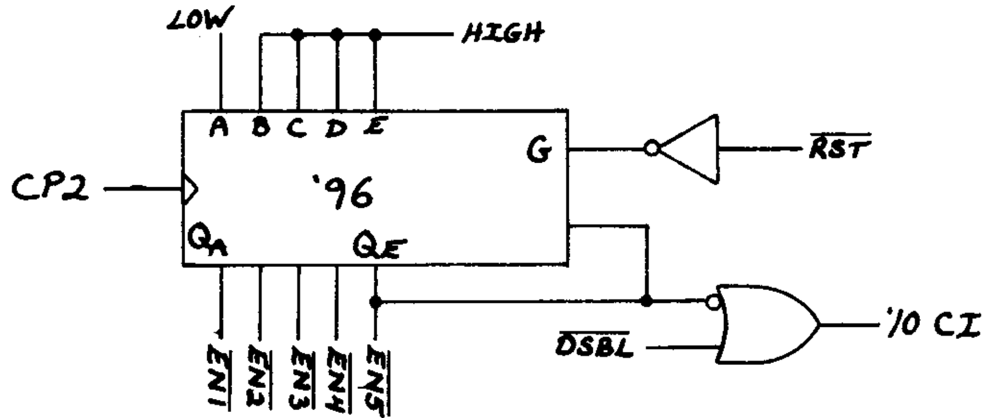
Figure 12c: Sample microcode, part 3 of 3.

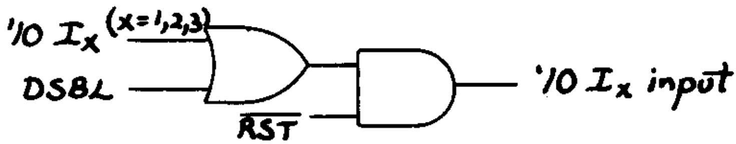
code will change frequently and significantly as it is debugged and refined. So, provision must be made for loading the microprogram from an external source.

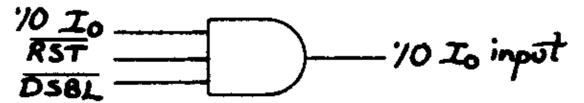
Figure 13 shows the additional logic necessary to control program loading. Most significant in these additions is the use of the disable signal ($\overline{\text{DSBL}}$) to stop the usual operation of the processor. While the disable signal is active, the sequencer instruction will always be "continue" (unless the reset signal, too, is active) and the sequencer will use the externally supplied clock (CP2) instead of the module's clock (CP).

Figure 14 shows the timing of a program loading operation. With both reset and disable active, CP2 is cycled once to reset the sequencer and ring counter. Next, with only disable active, the first byte of the first micro-instruction is placed on the program load data lines and CP2 is cycled again. This sequence continues, with five bytes to every micro-instruction, until the entire microcode has been loaded. Because of the arrangement of the sequencer carry in (CI) input and the ring counter used to select memory bytes, the microprogram counter will increment only after every fifth byte is loaded. When disable is inactive (i.e. in normal operation of the module) the carry in input is always high, so the microprogram counter is normally incremented.

When the entire microprogram has been loaded, the reset signal is made active again and the disable inactive, so the processor is again running with its own clock. When the reset signal is released, the microprogram will begin execution with the instruction in location zero.







ENx is the write enable for μcode byte x.

The '10 instruction logic yields:

$$
\begin{aligned}
I_{3-0} &= 0 \text{ (reset)} & \text{if } \overline{RST} = 0 \\
&= D \text{ (continue)} & \text{if } \overline{RST} \cdot DSBL = 1 \\
&= I_{3-0} \text{ from } \mu\text{code} & \text{if } \overline{RST} \cdot \overline{DSBL} = 1
\end{aligned}
$$

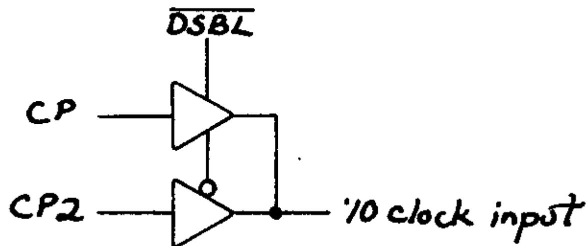


Figure 13: Additional logic for microprogram loading. If used, the arrangements shown here supercede those shown in figures 6 and 17.

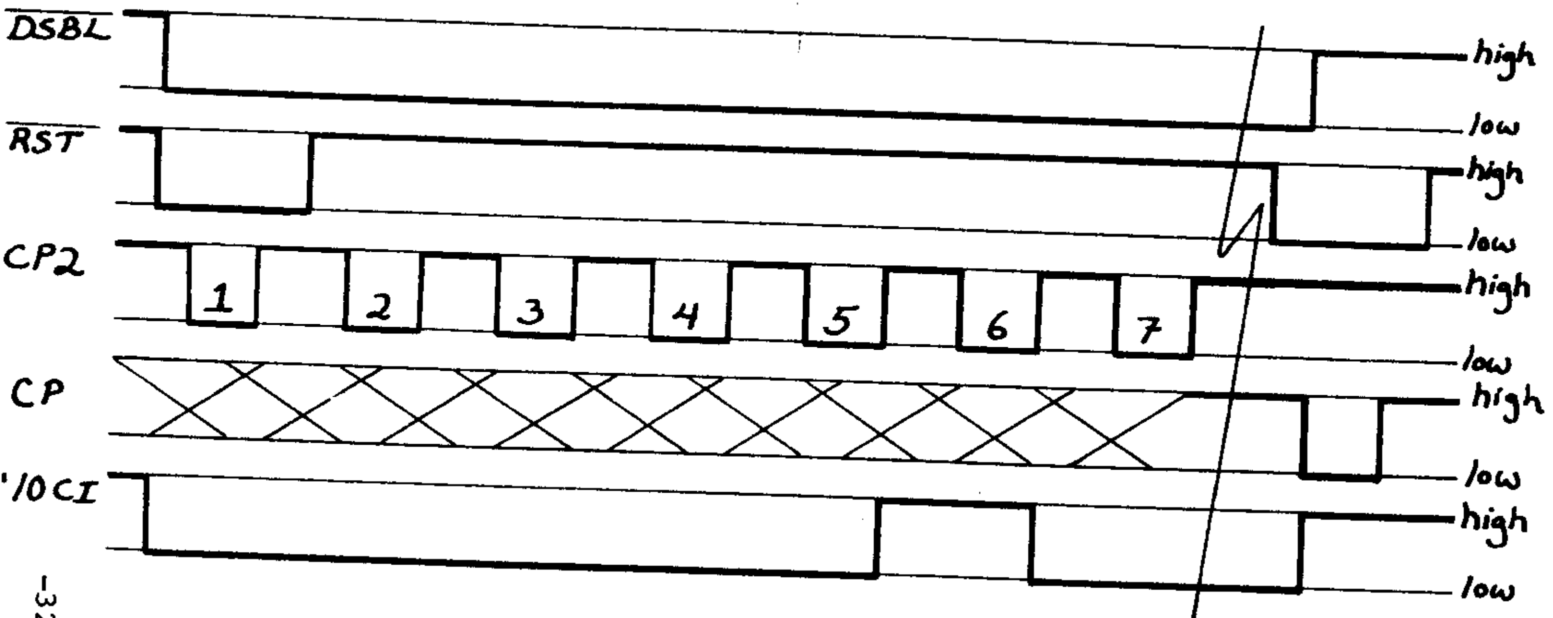


Figure 14: Timing for microprogram loading. The first negative-going pulse of CP2 resets the microprogram counter and the ring counter (see figure 13). Pulses 2-6 load the five bytes of the first micro-instruction. Since CI is high following pulse 5, the microprogram counter increments during the next cycle of CP2. So, pulse 7 starts the loading of the second micro-instruction. After the entire program has been loaded, $\overline{RST}$ is held low (with $\overline{DSBL}$ high) long enough for the processor clock to cycle at least once.

Alternatives

The processor described above is expected to be quite fast, because it will be microprogrammed, and will be reasonably economical in terms of chip count. There are some design alternatives, however, which could improve in one direction or the other, but probably not in both.

The first major alternative is the use of direct memory access (DMA) techniques for I/O. Instead of a single pair of memory address registers, there would be at least five: one for each port and one or more for the processor. The I/O port address registers would hold the memory address for the next byte to be received or transmitted. Additional registers would keep track of the number of bytes received or transmitted in a particular packet. Instead of needing service from the processor for every byte, the I/O ports would require service only before each packet, relieving the processor of a great deal of work. Memory access would have to be arbitrated so as to give absolute preference to the processor (preventing interruption of the program) and rotating preference to the four ports (to prevent any one from "hogging" the memory).

Unfortunately, such a DMA system would require a substantial number of chips (perhaps about twenty) if built of standard discrete logic. So, it does not appear as an economical alternative for a development system.

A second design alternative is precisely the one that was not pursued in this project: use of a single-chip microprocessor. Such a chip would replace the ALU, microprogram sequencer and selection logic, and the status and shift control unit. As was

mentioned very early in this paper, a module based on such a processor would probably run substantially more slowly than the processor module described above. If the single-chip processor were combined with specialized I/O ports and a DMA controller, however, it might regain some of that speed; how the resulting module would compare in speed and economy to the one developed in this project is an unresolved question.

Summary

The processor module described above, although it has highly specialized facilities for packet communication, has excellent facilities for general computation. Use of LSI parts keeps the chip count low, but the choice of bit-slice components provides flexibility. The pipelined architecture, as well as the use of bit-slice parts, allows for a very quick instruction cycle. Careful re-use and encoding of microcode fields makes the microcode fairly compact without greatly restricting the programmer. In short, the design presented meets the design objectives of flexibility, economy, and speed.

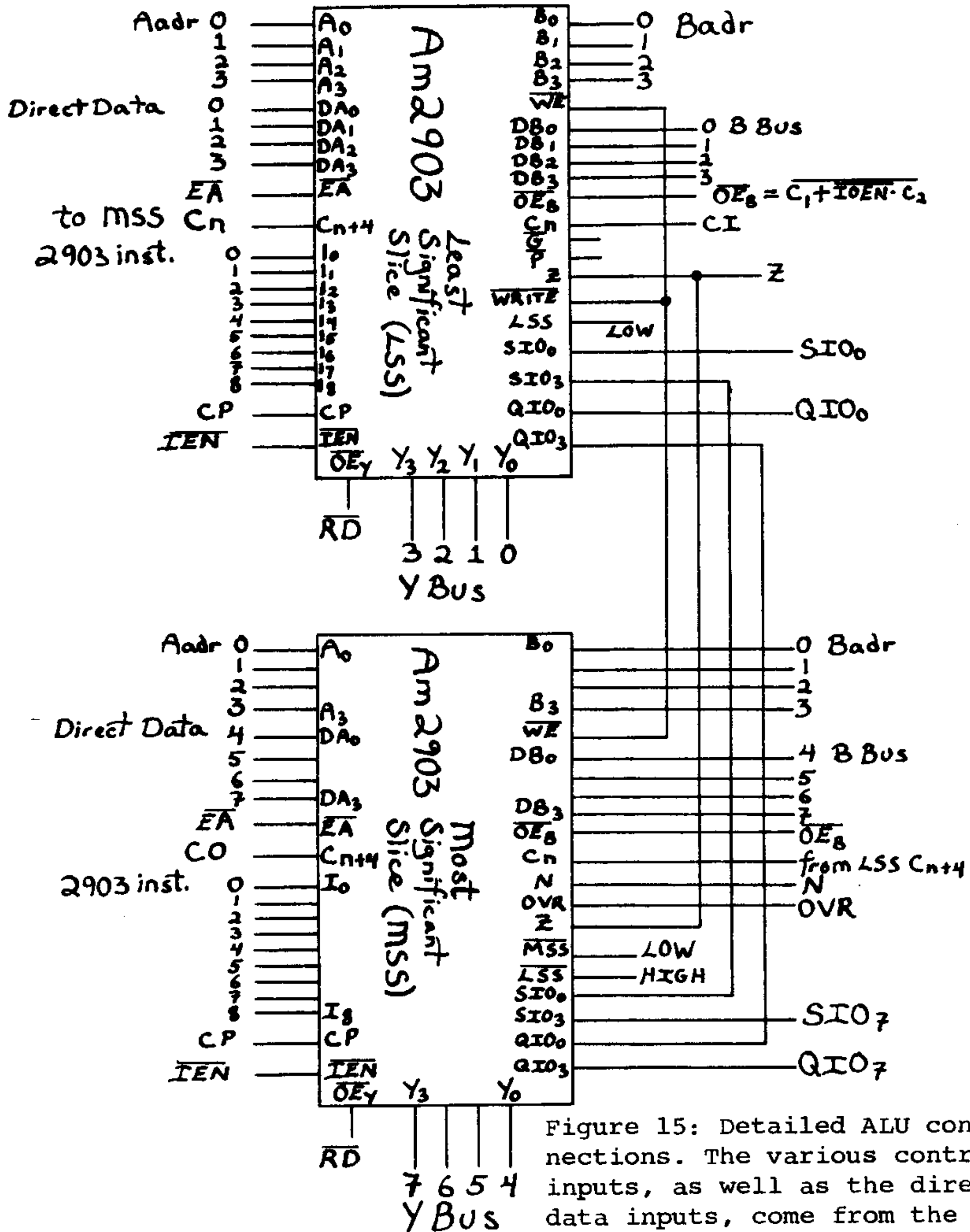
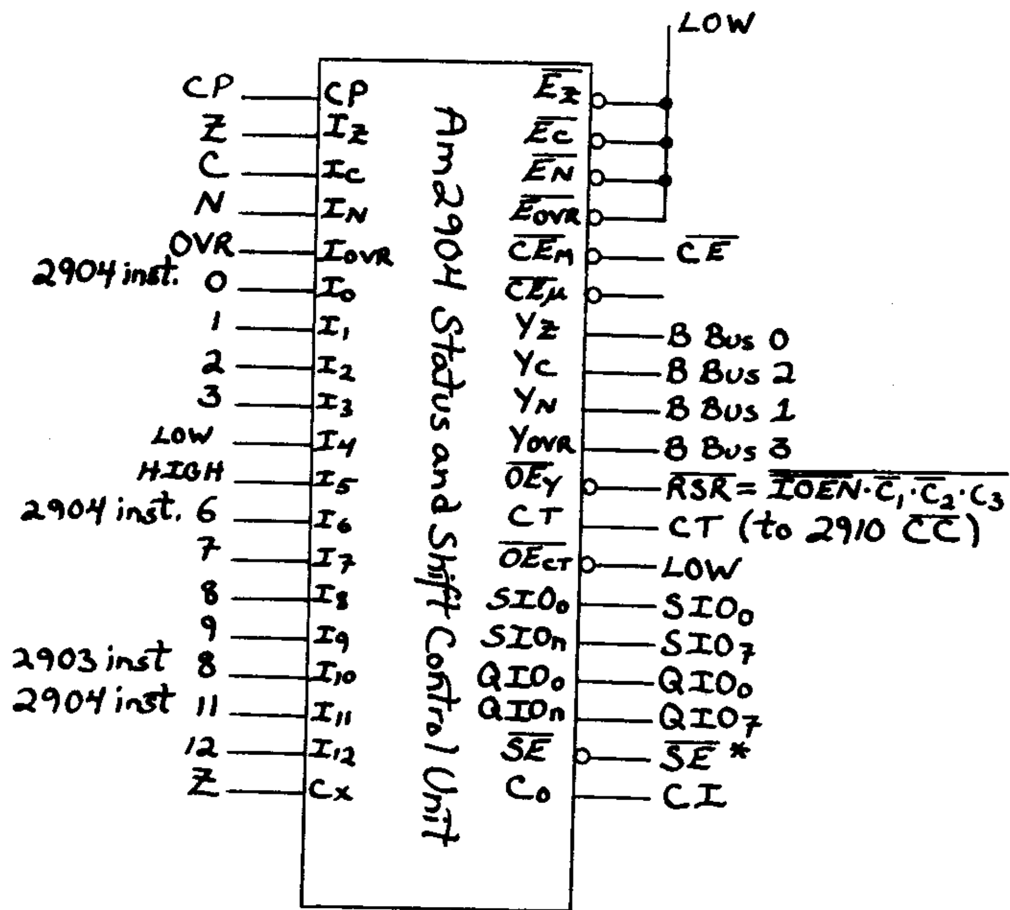


Figure 15: Detailed ALU connections. The various control inputs, as well as the direct data inputs, come from the current micro-instruction.



* $\overline{SE} = \overline{EA} + \overline{I_3 \cdot (\overline{I_0} \cdot I_1 + \overline{I_1}(I_2 \oplus I_0))} + RLD$, where I_{0-3} refer to the 2910 instruction bits, not the 2904 instruction bits. This disables the shift lines when the shift instruction field (μIR 1-4) is used for another purpose.

Figure 16: Detailed status and shift control unit connections. The various control lines tied HIGH or LOW reduce the usable sections of the Am2904 to those shown in figure 6.

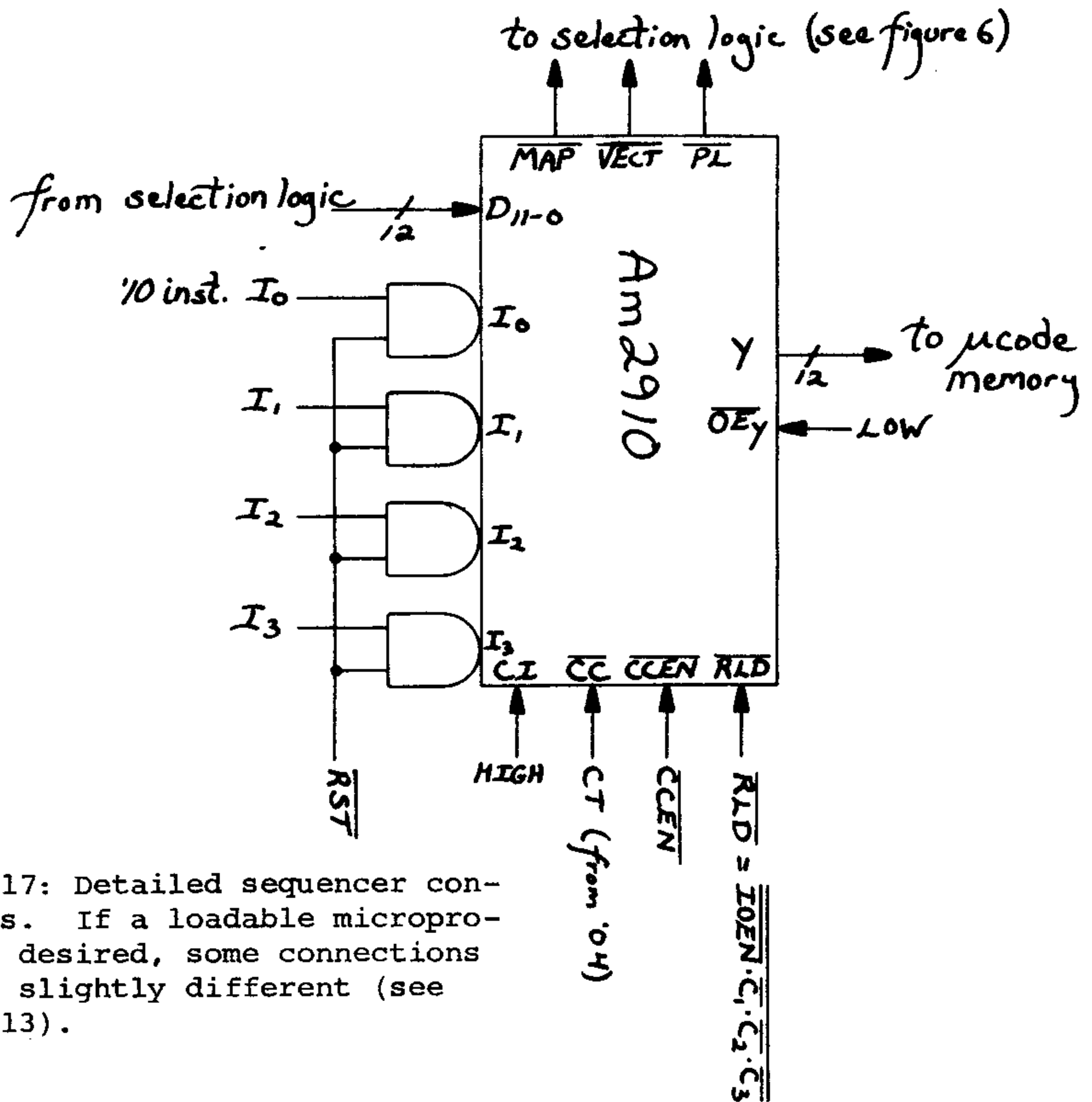
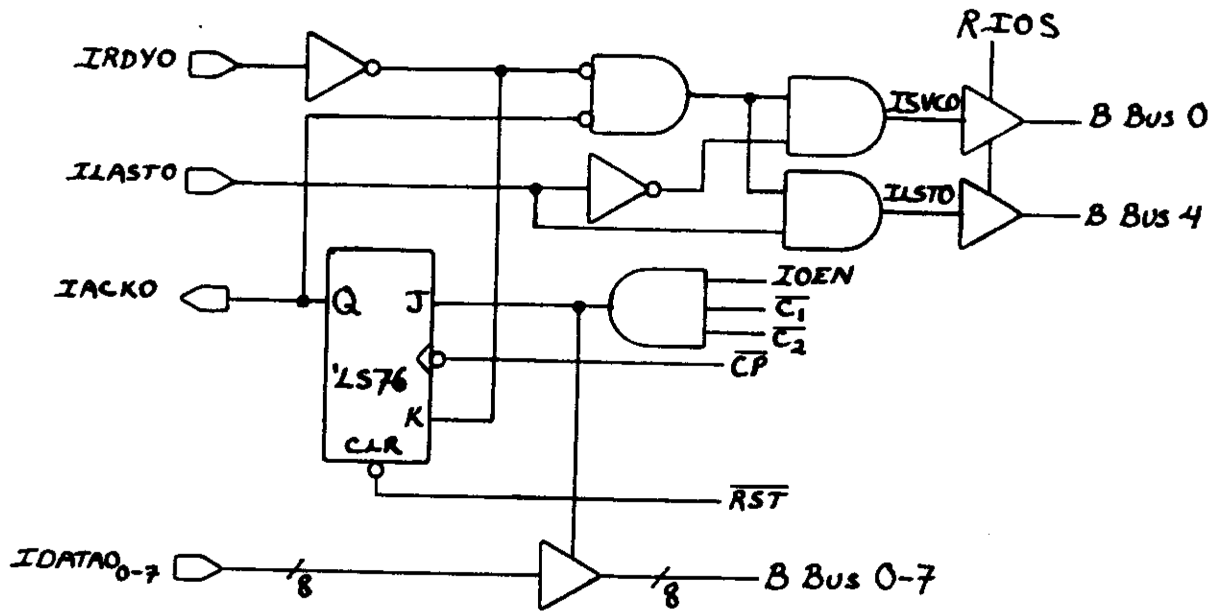
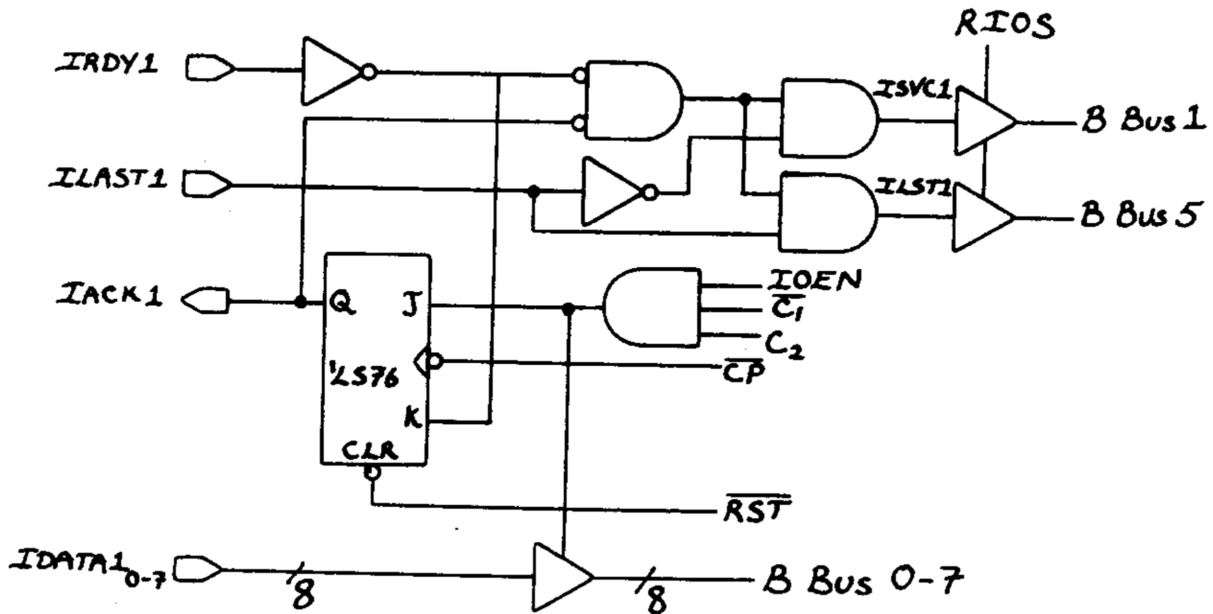


Figure 17: Detailed sequencer connections. If a loadable microprogram is desired, some connections will be slightly different (see figure 13).





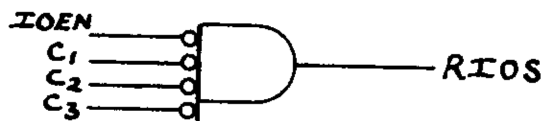


Figure 18a: Input ports 0 and 1. Note that reading the input ports causes an automatic acknowledge to the sender; the rest of the reset signalling sequence occurs automatically.

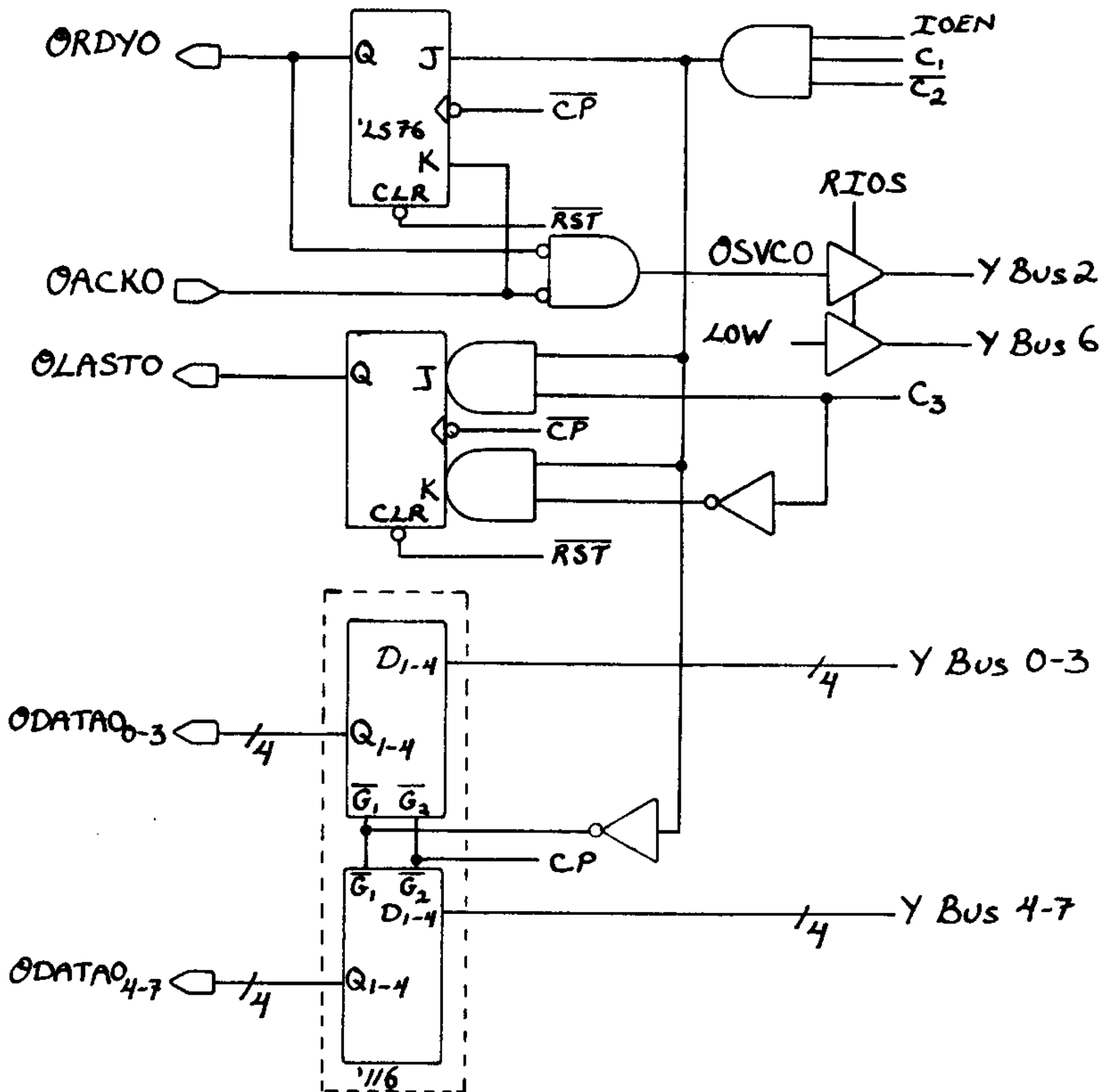


Figure 18b: Output port 0. Output port 1 is nearly identical, with suitable changes in signal names and the substitution of C2 for $\overline{C2}$, Y Bus 3 for Y Bus 2, and Y Bus 7 for Y Bus 6.

Area	Signal Name	# Bits
2903	A address	4
	B address	4
	Direct Data	8
	$\overline{EA}$, $\overline{IEN}$, $\overline{OE_Y}$, $\overline{OE_B}$	4
	I_0-I_8 (instruction)	9
2904	$\overline{SE}$, $\overline{OE_Y}$, $\overline{CE_M}$	3
	I_{0-3}, I_{6-9}, I_{11-12}	10
2910	I_{0-3} (sequence instruction)	4
	$\overline{CCEN}$	1
	D_{0-11} (jump address)	12
I/O	write output byte	1
	write last output byte	1
	read input byte	1
	read I/O status	1
	select I/O port	1
Memory	load higher address	1
	load/clear lower address	2
	read/write	2 (total: 69 bits)

Table 1: Summary of Microcode-Derived signals.

Field	Name	Description
1-4	'04 inst. I_{9-6}	Bits 6-10 of the Am2904 instruction are the shift linkage mux instruction. However, bit 10 is the same as Am2903 instruction bit 8. These bits are used as the shift mux bits only if $\overline{EA}$=0 (no direct data) and the Am2910 instruction is 8, A, D, or E.
5-8	Aadr	The Am2903 "A" address, provided $\overline{EA}$=0 and the sequencer instruction is 8, A, D, or E.
1-8	Direct Data	Data for the Am2903 DA input port, if $\overline{EA}$=1. Used for arithmetic constants and logical masks.
9-12	Badr	The Am2903 "B" address, if the sequencer instruction is 2, 6, 8, A, D, or E.
1-12	Jump Address D_{11-0}	The Am2910 "D" input data. Only the first (i.e. most significant) four bits are used for a variable register load; the last eight bits are read from the B bus. The first eight bits are used for the JMAP (16-way dispatch) and CJV (8-way priority dispatch) instructions; these take their last four bits from the B bus. All twelve bits are used for other instructions that use the D input.
13-21	'03 inst. I_{8-0}	The Am2903 instruction.

Table 2a: Microcode field definitions. Part 1 of 3.

Field	Name	Description
22-27	'04 inst. I_{12-11} I_{3-0}	Bits 3-0 of the Am2904 instruction select the condition code to be generated; bits 12-11 control the carry in bit generated.
28-31	'10 inst. I_{3-0}	The Am2910 microprogram sequencer instruction.
32	$\overline{CCEN}$	Condition code enable for the Am2910. Active low.
33-35	C_{1-3}	If IOEN=0,

C	B Source	B Dest.	Mnemonic
0	I/O status	ALU (DB port)	RIOS
1	status reg.	"	RSR
2	ALU (DB)	(none)	(default)
3	"	(clear lower addr. reg.)	CLA
4	"	higher addr. reg.	LHA
5	"	lower addr. reg.	LLA
6	"	memory input	WRITE
7	"	2910 counter	RLD

If IOEN=1, the current instruction is an I/O instruction. C=0xy is input; C=1xy is output. For both, x selects the input or output port to be used. For input, y indicates whether the input byte is written to memory. For output, y indicates whether the byte being sent is the last one in a packet.

Field	Name	Description
36	IOEN	I/O enable. Indicates an I/O instruction.

Table 2b: Microcode field definitions. Part 2 of 3.

Field	Name	Description
37	RD	Selects source of Y bus. RD=1 selects main memory (read); RD=0 selects the ALU Y port.
38	$\overline{CE}$	Enables loading of the status register. Active low.
39	$\overline{IEN}$	ALU instruction enable. Enables write to the B address, Q register, and sign FF, if specified by the instruction. Active low.
40	$\overline{EA}$	Selects the "R" source for the ALU (see figure 5). $\overline{EA}$=0 selects the "A" memory output; $\overline{EA}$=1 selects the DA input port (i.e. the direct data field, see above).

Table 2c: Microcode field definitions. Part 3 of 3.

1: General Arithmetic. The Am2910 instruction is restricted to 8, A, D, or E because these four instructions do not use the "D" input. So, the first bits of the micro-instruction are available for use as the shift mux field. It is possible to use 4 (PUSH) as the sequencer instruction also, but the register/counter will be loaded conditionally with the first twelve micro-instruction bits. Also, the shift mux will be disabled ($\overline{SE}$=1), so the shift mux field will have no effect. A general arithmetic instruction may include I/O (see below).
2: Arithmetic with Direct Data. This is very similar to General Arithmetic, but $\overline{EA}$=1 causes the DA input instead of the A memory output to be used as operand "R" (see figure 5). The shift mux is disabled and the "A" address is irrelevant, so the first eight instruction bits are used as the source of the direct data.
3: Computed Jump (16-way dispatch). Use of Am2910 instruction 2 (JMAP) enables the $\overline{MAP}$ selection line. The eight high order bits of the jump address are the first eight micro-instruction bits, but the last four bits are read directly from the B bus. Again, the shift mux is disabled. Also, since the "A" address cannot be independently specified, the Am2903 instruction should not use the R operand.
4: Priority Dispatch (8-way dispatch). Use of the Am2910 instruction 6 (CJV) enables the $\overline{VECT}$ selection line. The eight high order bits of the jump address are just as for JMAP, but the last four bits are a zero followed by the number of the least significant non-zero bit on the B bus.

Table 3: Microcode Instruction Types. Part 1 of 2.

5: General Jump (or fixed register load). All other sequencing instructions which use the Am2910 "D" inputs (D_{11-0}) enable the PL selection line, which selects the first twelve micro-instruction bits. This allows for all varieties of jumps to fixed locations, as well as for loading the register/counter with a fixed number.

6: Input. I/O instructions are marked by IOEN=1; input is distinguished further by $C_1=0$. For an input instruction, C_2 selects the particular port (0 or 1) and C_3 specifies if the input byte is to be written to memory.

7: Output. For output, $C_1=1$, C_2 selects the particular port, and C_3 indicates whether the current byte is the last of a packet. I/O instructions may be combined with all of the preceeding instruction types, but not with the one following.

8: Variable Register/Counter Load. If a variable load of the Am2910 internal register/counter is specified (see "C" field definition, Table 2), bits 1-4 of the current micro-instruction are used the the highest order "D" inputs (D_{11-8}). The remaining eight bits are read from the B bus. The Am2910 instruction is restricted to those codes which do not use the "D" inputs, although one might be able to achieve some tricky effects by ignoring this prohibition.

Table 3: Microcode Instruction Types. Part 2 of 2.

Appendix A: Am2903 Instructions

The following tables summarize the operation of the Am2903; see [3] for more detailed information and application notes.

ALU FUNCTIONS

I_4	I_3	I_2	I_1	Hex Code	ALU Functions
L	L	L	L	0	I_0 L: Special Functions I_0 H: F_i HIGH
L	L	L	H	1	F = S Minus R Minus 1 Plus C_n
L	L	H	L	2	F = R Minus S Minus 1 Plus C_n
L	L	H	H	3	F = R Plus S Plus C_n
L	H	L	L	4	F = S Plus C_n
L	H	L	H	5	F = $\bar{S}$ Plus C_n
L	H	H	L	6	F = R Plus C_n
L	H	H	H	7	F = $\bar{R}$ Plus C_n
H	L	L	L	8	F_i = LOW
H	L	L	H	9	F_i = $\bar{R}_i$ AND S_i
H	L	H	L	A	F_i = R_i EXCLUSIVE NOR S_i
H	L	H	H	B	F_i = R_i EXCLUSIVE OR S_i
H	H	L	L	C	F_i = R_i AND S_i
H	H	L	H	D	F_i = R_i NOR S_i
H	H	H	L	E	F_i = R_i NAND S_i
H	H	H	H	F	F_i = R_i OR S_i

L = LOW H = HIGH i = 0 to 3

ALU OPERAND SOURCES

$\overline{E_A}$	I_0	$\overline{OE_B}$	ALU Operand R	ALU Operand S
L	L	L	RAM Output A	RAM Output B
L	L	H	RAM Output A	DB_{0-3}
L	H	X	RAM Output A	Q Register
H	L	L	DA_{0-3}	RAM Output B
H	L	H	DA_{0-3}	DB_{0-3}
H	H	X	DA_{0-3}	Q Register

L = LOW H = HIGH X = Don't Care

ALU DESTINATION CONTROL FOR I_0 OR I_1 OR I_2 OR I_3 OR I_4 = HIGH, $\overline{IEN}$ = LOW.

I_8	I_7	I_6	I_5	Hex Code	ALU Shifter Function	SIO_3 Most Sig. Slice	SIO_3 Other Slices	Y_3 Most Sig. Slice	Y_3 Other Slices	Y_2 Most Sig. Slice	Y_2 Other Slices	Y_1	Y_0	SIO_0	$\overline{Write}$	Q Reg & Shifter Function	QIO_3	QIO_0
L	L	L	L	0	Arith F/2 → Y	Input	Input	F_3	SIO_3	SIO_3	F_3	F_2	F_1	F_0	L	Hold	Hi-Z	Hi-Z
L	L	L	H	1	Log F/2 → Y	Input	Input	SIO_3	SIO_3	F_3	F_3	F_2	F_1	F_0	L	Hold	Hi-Z	Hi-Z
L	L	H	L	2	Arith F/2 → Y	Input	Input	F_3	SIO_3	SIO_3	F_3	F_2	F_1	F_0	L	Log Q/2 → Q	Input	Q_0
L	L	H	H	3	Log F/2 → Y	Input	Input	SIO_3	SIO_3	F_3	F_3	F_2	F_1	F_0	L	Log Q/2 → Q	Input	Q_0
L	H	L	L	4	F → Y	Input	Input	F_3	F_3	F_2	F_2	F_1	F_0	Parity	L	Hold	Hi-Z	Hi-Z
L	H	L	H	5	F → Y	Input	Input	F_3	F_3	F_2	F_2	F_1	F_0	Parity	H	Log Q/2 → Q	Input	Q_0
L	H	H	L	6	F → Y	Input	Input	F_3	F_3	F_2	F_2	F_1	F_0	Parity	H	F → Q	Hi-Z	Hi-Z
L	H	H	H	7	F → Y	Input	Input	F_3	F_3	F_2	F_2	F_1	F_0	Parity	L	F → Q	Hi-Z	Hi-Z
H	L	L	L	8	Arith 2F → Y	F_2	F_3	F_3	F_2	F_1	F_1	F_0	SIO_0	Input	L	Hold	Hi-Z	Hi-Z
H	L	L	H	9	Log 2F → Y	F_3	F_3	F_2	F_2	F_1	F_1	F_0	SIO_0	Input	L	Hold	Hi-Z	Hi-Z
H	L	H	L	A	Arith 2F → Y	F_2	F_3	F_3	F_2	F_1	F_1	F_0	SIO_0	Input	L	Log 2Q → Q	Q_3	Input
H	L	H	H	B	Log 2F → Y	F_3	F_3	F_2	F_2	F_1	F_1	F_0	SIO_0	Input	L	Log 2Q → Q	Q_3	Input
H	H	L	L	C	F → Y	F_3	F_3	F_3	F_3	F_2	F_2	F_1	F_0	Hi-Z	H	Hold	Hi-Z	Hi-Z
H	H	L	H	D	F → Y	F_3	F_3	F_3	F_3	F_2	F_2	F_1	F_0	Hi-Z	H	Log 2Q → Q	Q_3	Input
H	H	H	L	E	SIO_0 → Y_0, Y_1, Y_2, Y_3	SIO_0	SIO_0	SIO_0	SIO_0	SIO_0	SIO_0	SIO_0	SIO_0	Input	L	Hold	Hi-Z	Hi-Z
H	H	H	H	F	F → Y	F_3	F_3	F_3	F_3	F_2	F_2	F_1	F_0	Hi-Z	L	Hold	Hi-Z	Hi-Z

Parity = $F_3 \forall F_2 \forall F_1 \forall F_0 \forall SIO_3$
$\forall$ = Exclusive OR

L = LOW
H = HIGH

Hi-Z = High Impedance

SPECIAL FUNCTIONS: $I_0 = I_1 = I_2 = I_3 = I_4$ = LOW, $\overline{IEN}$ = LOW

I_8	I_7	I_6	I_5	Hex Code	Special Function	ALU Function	ALU Shifter Function	SIO_3 Most Sig. Slice	SIO_3 Other Slices	SIO_0	Q Reg & Shifter Function	QIO_3	QIO_0	$\overline{WRITE}$
L	L	L	L	0	Unsigned Multiply	$F = S + C_n$ if Z=L $F = R + S + C_n$ if Z=H	Log. F/2→Y (Note 1)	Hi-Z	Input	F_0	Log. Q/2→Q	Input	Q_0	L
L	L	H	L	2	Two's Complement Multiply	$F = S + C_n$ if Z=L $F = R + S + C_n$ if Z=H	Log. F/2→Y (Note 2)	Hi-Z	Input	F_0	Log. Q/2→Q	Input	Q_0	L
L	H	L	L	4	Increment by One or Two	$F = S + 1 + C_n$	F→Y	Input	Input	Parity	Hold	Hi-Z	Hi-Z	L
L	H	L	H	5	Sign/Magnitude-Two's Complement	$F = S + C_n$ if Z=L $F = \bar{S} + C_n$ if Z=H	F→Y (Note 3)	Input	Input	Parity	Hold	Hi-Z	Hi-Z	L
L	H	H	L	6	Two's Complement Multiply, Last Cycle	$F = S + C_n$ if Z=L $F = S - R - 1 + C_n$ if Z=H	Log. F/2→Y (Note 2)	Hi-Z	Input	F_0	Log. Q/2→Q	Input	Q_0	L
H	L	L	L	8	Single Length Normalize	$F = S + C_n$	F→Y	F_3	F_3	Hi-Z	Log. 2Q→Q	Q_3	Input	L
H	L	H	L	A	Double Length Normalize and First Divide Op.	$F = S + C_n$	Log 2F→Y	$R_3 \forall F_3$	F_3	Input	Log. 2Q→Q	Q_3	Input	L
H	H	L	L	C	Two's Complement Divide	$F = S + R + C_n$ if Z=L $F = S - R - 1 + C_n$ if Z=H	Log. 2F→Y	$\overline{R_3 \forall F_3}$	F_3	Input	Log. 2Q→Q	Q_3	Input	L
H	H	H	L	E	Two's Complement Divide, Correction and Remainder	$F = S + R + C_n$ if Z=L $F = S - R - 1 + C_n$ if Z=H	F→Y	F_3	F_3	Hi-Z	Log. 2Q→Q	Q_3	Input	L

NOTES:
1. At the most significant slice only, the C_{n+4} signal is internally gated to the Y_3 output.
2. At the most significant slice only, $F_3 \forall$ OVR is internally gated to the Y_3 output.
3. At the most significant slice only, $S_3 \forall F_3$ is generated at the Y_3 output.
4. Op codes 1, 3, 7, 9, B, D, and F are reserved for future use.

L = LOW
H = HIGH
X = Don't Care

Hi-Z = High Impedance
∀ = Exclusive OR
Parity = $SIO_3 \forall F_3 \forall F_2 \forall F_1 \forall F_0$

Am2903 STATUS OUTPUTS

(Hex) $I_8I_7I_6I_5$	(Hex) $I_4I_3I_2I_1$	I_0	GI (I = 0 to 3)	PI (I = 0 to 3)	C_{n+4}	$\overline{P}$/OVR Most Sig. Slice	$\overline{P}$/OVR Other Slices	$\overline{G}$/N Most Sig. Slice	$\overline{G}$/N Other Slices	Z Most Sig. Slice	Z Intermediate Slice	Z Least Sig. Slice
X	0	H	0	1	0	0	0	F_3	$\overline{G}$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$
X	1	X	$\overline{R}_i \wedge S_i$	$\overline{R}_i \vee S_i$	$G \vee PC_n$	$C_{n+3} \forall C_{n+4}$	$\overline{P}$	F_3	$\overline{G}$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$
X	2	X	$R_i \wedge \overline{S}_i$	$R_i \vee \overline{S}_i$	$G \vee PC_n$	$C_{n+3} \forall C_{n+4}$	$\overline{P}$	F_3	$\overline{G}$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$
X	3	X	$R_i \wedge S_i$	$R_i \vee S_i$	$G \vee PC_n$	$C_{n+3} \forall C_{n+4}$	$\overline{P}$	F_3	$\overline{G}$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$
X	4	X	0	S_i	$G \vee PC_n$	$C_{n+3} \forall C_{n+4}$	$\overline{P}$	F_3	$\overline{G}$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$
X	5	X	0	$\overline{S}_i$	$G \vee PC_n$	$C_{n+3} \forall C_{n+4}$	$\overline{P}$	F_3	$\overline{G}$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$
X	6	X	0	R_i	$G \vee PC_n$	$C_{n+3} \forall C_{n+4}$	$\overline{P}$	F_3	$\overline{G}$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$
X	7	X	0	$\overline{R}_i$	$G \vee PC_n$	$C_{n+3} \forall C_{n+4}$	$\overline{P}$	F_3	$\overline{G}$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$
X	8	X	0	1	0	0	0	F_3	$\overline{G}$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$
X	9	X	$\overline{R}_i \wedge S_i$	1	0	0	0	F_3	$\overline{G}$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$
X	A	X	$R_i \wedge S_i$	$R_i \vee S_i$	0	0	0	F_3	$\overline{G}$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$
X	B	X	$\overline{R}_i \wedge S_i$	$\overline{R}_i \vee S_i$	0	0	0	F_3	$\overline{G}$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$
X	C	X	$R_i \wedge S_i$	1	0	0	0	F_3	$\overline{G}$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$
X	D	X	$\overline{R}_i \wedge \overline{S}_i$	1	0	0	0	F_3	$\overline{G}$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$
X	E	X	$R_i \wedge S_i$	1	0	0	0	F_3	$\overline{G}$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$
X	F	X	$\overline{R}_i \wedge \overline{S}_i$	1	0	0	0	F_3	$\overline{G}$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$
0	0	L	0 if Z = L $R_i \wedge S_i$ if Z = H	S_i if Z = L $R_i \vee S_i$ if Z = H	$G \vee PC_n$	$C_{n+3} \forall C_{n+4}$	$\overline{P}$	F_3	$\overline{G}$	Input	Input	Q_0
2	0	L	0 if Z = L $R_i \wedge S_i$ if Z = H	S_i if Z = L $R_i \vee S_i$ if Z = H	$G \vee PC_n$	$C_{n+3} \forall C_{n+4}$	$\overline{P}$	F_3	$\overline{G}$	Input	Input	Q_0
4	0	L	See Note 1	See Note 2	$G \vee PC_n$	$C_{n+3} \forall C_{n+4}$	$\overline{P}$	F_3	$\overline{G}$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$	$\overline{Y}_0\overline{Y}_1\overline{Y}_2\overline{Y}_3$
5	0	L	0	S_i if Z = L $\overline{S}_i$ if Z = H	$G \vee PC_n$	$C_{n+3} \forall C_{n+4}$	$\overline{P}$	F_3 if Z = L $F_3 \forall S_3$ if Z = H	$\overline{G}$	S_3	Input	Input
6	0	L	0 if Z = L $\overline{R}_i \wedge S_i$ if Z = H	S_i if Z = L $\overline{R}_i \vee S_i$ if Z = H	$G \vee PC_n$	$C_{n+3} \forall C_{n+4}$	P	F_3	G	Input	Input	Q_0
8	0	L	0	S_i	See Note 3	$Q_2 \forall Q_1$	$\overline{P}$	Q_3	$\overline{G}$	$\overline{Q}_0\overline{Q}_1\overline{Q}_2\overline{Q}_3$	$\overline{Q}_0\overline{Q}_1\overline{Q}_2\overline{Q}_3$	$\overline{Q}_0\overline{Q}_1\overline{Q}_2\overline{Q}_3$
A	0	L	0	S_i	See Note 4	$F_2 \forall F_1$	$\overline{P}$	F_3	$\overline{G}$	See Note 5	See Note 5	See Note 5
C	0	L	$R_i \wedge S_i$ if Z = L $\overline{R}_i \wedge S_i$ if Z = H	$R_i \vee S_i$ if Z = L $\overline{R}_i \vee S_i$ if Z = H	$G \vee PC_n$	$C_{n+3} \forall C_{n+4}$	$\overline{P}$	F_3	$\overline{G}$	Sign Compare FF Output	Input	Input
E	0	L	$R_i \wedge S_i$ if Z = L $\overline{R}_i \wedge S_i$ if Z = H	$R_i \vee S_i$ if Z = L $\overline{R}_i \vee S_i$ if Z = H	$G \vee PC_n$	$C_{n+3} \forall C_{n+4}$	$\overline{P}$	F_3	$\overline{G}$	Sign Compare FF Output	Input	Input

L = LOW = 0
H = HIGH = 1
$\vee$ = OR
$\wedge$ = AND
$\forall$ = EXCLUSIVE OR
$P = P_3P_2P_1P_0$
$G = G_3 \vee G_2P_3 \vee G_1P_2P_3 \vee G_0P_1P_2P_3$
$C_{n+3} = G_2 \vee G_1P_2 \vee G_0P_1P_2 \vee C_nP_0P_1P_2$

NOTES:
1. If $\overline{LSS}$ is LOW, $G_0 = S_0$ and $G_{1,2,3} = 0$
   If $\overline{LSS}$ is HIGH, $G_{0,1,2,3} = 0$
2. If $\overline{LSS}$ is LOW, $P_0 = 1$ and $P_{1,2,3} = S_{1,2,3}$
   If $\overline{LSS}$ is HIGH, $P_i = S_i$
3. At the most significant slice, $C_{n+4} = Q_3 \forall Q_2$
   At other slices, $C_{n+4} = G \vee PC_n$
4. At the most significant slice, $C_{n+4} = F_3 \forall F_2$
   At other slices, $C_{n+4} = G \vee PC_n$
5. $Z = \overline{Q}_0\overline{Q}_1\overline{Q}_2\overline{Q}_3\overline{F}_0\overline{F}_1\overline{F}_2\overline{F}_3$

Appendix B: Am2904 Instructions

See [4] for more information.

CONDITION CODE OUTPUT (CT) INSTRUCTION CODES.

I_{3-0} HEX	I_3	I_2	I_1	I_0	$I_5 = 1, I_4 = 0$
0	0	0	0	0	$(M_N \oplus M_{OVR}) + M_Z$
1	0	0	0	1	$(M_N \odot M_{OVR}) \cdot \overline{M}_Z$
2	0	0	1	0	$M_N \oplus M_{OVR}$
3	0	0	1	1	$M_N \odot M_{OVR}$
4	0	1	0	0	M_Z
5	0	1	0	1	$\overline{M}_Z$
6	0	1	1	0	M_{OVR}
7	0	1	1	1	$\overline{M}_{OVR}$
8	1	0	0	0	$M_C + M_Z$
9	1	0	0	1	$\overline{M}_C \cdot \overline{M}_Z$
A	1	0	1	0	M_C
B	1	0	1	1	$\overline{M}_C$
C	1	1	0	0	$\overline{M}_C + M_Z$
D	1	1	0	1	$M_C \cdot \overline{M}_Z$
E	1	1	1	0	M_N
F	1	1	1	1	$\overline{M}_N$

Notes: 1. $\oplus$ Represents EXCLUSIVE-OR
$\odot$ Represents EXCLUSIVE-NOR or coincidence.

CARRY-IN CONTROL MULTIPLEXER INSTRUCTION CODES.

I_{12}	I_{11}	I_3	I_2	I_1	C_0
0	0	X	X	X	0
0	1	X	X	X	1
1	0	X	X	X	C_X
1	1	0	X	X	M_C
1	1	X	1	X	M_C
1	1	X	X	1	M_C
1	1	1	0	0	$\overline{M}_C$

CRITERIA FOR COMPARING TWO NUMBERS FOLLOWING "A MINUS B" OPERATION.

	For Unsigned Numbers			For 2's Complement Numbers		
		I_{3-0}			I_{3-0}	
Relation	Status	CT = H	CT = L	Status	CT = H	CT = L
A = B	Z = 1	4	5	Z = 1	4	5
A ≠ B	Z = 0	5	4	Z = 0	5	4
A ≥ B	C = 1	A	B	$N \odot OVR = 1$	3	2
A < B	C = 0	B	A	$N \oplus OVR = 1$	2	3
A > B	$C \cdot \overline{Z} = 1$	D	C	$(N \odot OVR) \cdot \overline{Z} = 1$	1	0
A ≤ B	$\overline{C} + Z = 1$	C	D	$(N \oplus OVR) + Z = 1$	0	1

$\oplus$ = Exclusive OR
$\odot$ = Exclusive NOR
H = HIGH
L = LOW
Note: For Am2910, the CC input is active LOW, so use I_{3-0} code to produce CT = L for the desired test.

Register Load Operations

CE	I_{3-0}	Operation
0	0-7, A-F	Normal Load
0	8, 9	Load with inverted carry
1	(any)	No Load

SHIFT LINKAGE MULTIPLEXER INSTRUCTION CODES.

I_{10}	I_9	I_8	I_7	I_6	M_C RAM Q	SIO_0	SIO_n	QIO_0	QIO_n	Loaded into M_C
					MSB LSB MSB LSB					
0	0	0	0	0		Z	0	Z	0	
0	0	0	0	1		Z	1	Z	1	
0	0	0	1	0		Z	0	Z	M_N	SIO_0
0	0	0	1	1		Z	1	Z	SIO_0	
0	0	1	0	0		Z	M_C	Z	SIO_0	
0	0	1	0	1		Z	M_N	Z	SIO_0	
0	0	1	1	0		Z	0	Z	SIO_0	
0	0	1	1	1		Z	0	Z	SIO_0	QIO_0
0	1	0	0	0		Z	SIO_0	Z	QIO_0	SIO_0
0	1	0	0	1		Z	M_C	Z	QIO_0	SIO_0
0	1	0	1	0		Z	SIO_0	Z	QIO_0	
0	1	0	1	1		Z	I_C	Z	SIO_0	
0	1	1	0	0		Z	M_C	Z	SIO_0	QIO_0
0	1	1	0	1		Z	QIO_0	Z	SIO_0	QIO_0
0	1	1	1	0		Z	$I_N \oplus I_{OVR}$	Z	SIO_0	
0	1	1	1	1		Z	QIO_0	Z	SIO_0	
					MSB LSB MSB LSB					
1	0	0	0	0		0	Z	0	Z	SIO_n
1	0	0	0	1		1	Z	1	Z	SIO_n
1	0	0	1	0		0	Z	0	Z	
1	0	0	1	1		1	Z	1	Z	
1	0	1	0	0		QIO_n	Z	0	Z	SIO_n
1	0	1	0	1		QIO_n	Z	1	Z	SIO_n
1	0	1	1	0		QIO_n	Z	0	Z	
1	0	1	1	1		QIO_n	Z	1	Z	
1	1	0	0	0		SIO_n	Z	QIO_n	Z	SIO_n
1	1	0	0	1		M_C	Z	QIO_n	Z	SIO_n
1	1	0	1	0		SIO_n	Z	QIO_n	Z	
1	1	0	1	1		M_C	Z	0	Z	
1	1	1	0	0		QIO_n	Z	M_C	Z	SIO_n
1	1	1	0	1		QIO_n	Z	SIO_n	Z	SIO_n
1	1	1	1	0		QIO_n	Z	M_C	Z	
1	1	1	1	1		QIO_n	Z	SIO_n	Z	

Appendix C: Am2910 Instructions

HEX I_3-I_0	MNEMONIC	NAME	REG/ CNTR CON-TENTS	FAIL $\overline{CCEN}$ = LOW and $\overline{CC}$ = HIGH		PASS $\overline{CCEN}$ = HIGH or $\overline{CC}$ = LOW		REG/ CNTR	ENABLE
				Y	STACK	Y	STACK		
0	JZ	JUMP ZERO	X	0	CLEAR	0	CLEAR	HOLD	PL
1	CJS	COND JSB PL	X	PC	HOLD	D	PUSH	HOLD	PL
2	JMAP	JUMP MAP	X	D	HOLD	D	HOLD	HOLD	MAP
3	CJP	COND JUMP PL	X	PC	HOLD	D	HOLD	HOLD	PL
4	PUSH	PUSH/COND LD CNTR	X	PC	PUSH	PC	PUSH	Note 1	PL
5	JSRP	COND JSB R/PL	X	R	PUSH	D	PUSH	HOLD	PL
6	CJV	COND JUMP VECTOR	X	PC	HOLD	D	HOLD	HOLD	VECT
7	JRP	COND JUMP R/PL	X	R	HOLD	D	HOLD	HOLD	PL
8	RFCT	REPEAT LOOP, CNTR ≠ 0	≠ 0	F	HOLD	F	HOLD	DEC	PL
			= 0	PC	POP	PC	POP	HOLD	PL
9	RPCT	REPEAT PL, CNTR ≠ 0	≠ 0	D	HOLD	D	HOLD	DEC	PL
			= 0	PC	HOLD	PC	HOLD	HOLD	PL
A	CRTN	COND RTN	X	PC	HOLD	F	POP	HOLD	PL
B	CJPP	COND JUMP PL & POP	X	PC	HOLD	D	POP	HOLD	PL
C	LDCT	LD CNTR & CONTINUE	X	PC	HOLD	PC	HOLD	LOAD	PL
D	LOOP	TEST END LOOP	X	F	HOLD	PC	POP	HOLD	PL
E	CONT	CONTINUE	X	PC	HOLD	PC	HOLD	HOLD	PL
F	TWB	THREE-WAY BRANCH	≠ 0	F	HOLD	PC	POP	DEC	PL
			= 0	D	POP	PC	POP	HOLD	PL

Note 1: If $\overline{CCEN}$ = LOW and $\overline{CC}$ = HIGH, hold; else load. X = Don't Care

The above table summarizes the Am2910 instruction set. Additionally, RLD forces loading of the internal register/counter regardless of the current instruction. See [3] for further details and application notes.

Appendix D: Sample Microcode Commentary

The microcode fragment shown in figure 12 is designed to poll the I/O prots and service those which are both servicable and marked as active.

The "active list" is maintained as a mask kept in register zero (R0). A particular port is listed as active if the bit(s) corresponding to its status bit(s) are set to one. Input port 0, for example, is marked by bits zero and four; output port one by bit three.

The first step of the microcode shown reads the I/O status (C=RIOS), masks it with R0 (Aadr=0, '03 inst.=F,C,0), and writes the result in register one (Badr=1). The status register is loaded ($\overline{CE}$=0), and the next address (i.e. DISPATCH+1) is pushed onto the stack ('10 inst.=PUSH). The PUSH instruction also conditionally loads the register/counter with the first twelve bits of the current micro-instruction, but this is an unimportant side effect in this case.

The next step does a conditional eight-way dispatch ('10 inst.=CJV). The dispatch is made if the result of the previous step (now the contents of R1) was non-zero ('04 I_{3-0}=4). The First eight bits of the destination address are the first eight bits of the micro-instruction; the next bit is zero; and the last three bits are the number of the least significant non-zero bit on the B bus. Badr=1, so the contents of R1 are on the B bus. As a result, IOSVC must have an address whose last four bits are zero. It and the following five lines (since there are only six status bits) constitute a dispatch table for I/O service.

If the dispatch is not made, control goes to the next statement, which pops the address stack and jumps to that address ('10 inst.=CRTN). The jump will be to the previous instruction, the dispatch condition will fail agian, and on the second pass this instruction will cause a return to the caller. The first invocation clears "DISPATCH+1" off the stack; the second has the actually desired destination.

Why is the extra address on the stack? So that the dispatch step can be the first of a loop. In addition to servicing the appropriate port, each of the six particular service routines masks out the bit that caused it to be called, then loops back to the dispatch step is R1 is still non-zero.

Associated with each port is a pair of registers containing the memory address for the next byte to be written or read. Inpur port 0 uses R2 and R3, so I0 starts by loading the address registers from R2 and R3. It also increments R2, the less significant half of the address, so the next byte will go in the next location. The assumption is made here that packets are less than 256 bytes long, so R2 will not overflow. If overflow were an issue here, one could add the carry out from the increment of R2 to R3 without any additional steps.

Next, I0 masks out bit zero of R1, reads the input byte into memory, and loops if R1 is not zero. If clearing of bit zero made R1 zero, the routine returns to the original caller.

L0 is similar to I0, but masks bits zero and four of R0 as well, to mark input port zero as inactive. It also writes R2 into memory at the front of the packet just received. Assuming R2 was initially one, its final value will be the length of the

packet plus one (by induction).

The output routines are designed to transmit packets of the format written into memory by the input routines. Thus, they check in front of the packet for an indication of its length: if the current byte is the last one, they mask the appropriate R0 bit to mark the port inactive.

Bibliography

[1]: Dennis, J.B., Misunas, D.P., and Leung, C.K., "A Highly Parallel Processor Using a Data Flow Machine Language." Computation Structures Group Memo 134, MIT Laboratory for Computer Science, Cambridge, Massachusetts, January 1977.

[2]: Redford, J.L., "A Design for a Routing Module." Computation Structures Note 39, MIT Laboratory for Computer Science, Cambridge, Massachusetts, February 1977.

[3]: The 2900 Family Data Book. Advanced Micro Devices (AM-PUB003), Sunnyvale, California, April, 1978.

[4]: Am2904 Status and Shift Control Unit. Advanced Micro Devices (AM-PUB077), Sunnyvale, California, 1978.