

AMD CUSTOMER EDUCATION

ED2900A

**INTRODUCTION TO
DESIGNING WITH THE
Am2900 FAMILY OF
MICROPROGRAMMABLE
BIPOLAR DEVICES**

**LECTURE
VOLUME I**

ED2900A

INTRODUCTION TO DESIGNING WITH THE Am2900 FAMILY
OF MICROPROGRAMMING BIPOLAR DEVICES

Volume I

3rd Edition

January 1985
Advanced Micro Devices, Inc.
Customer Education Center

Volume I
Table of Contents

- Educational Objectives
- Introduction
 - Bit-slice Architecture
 - Microprogramming
- Am2900 Family of Microprogrammable Devices
- Analyzing and Designing a Computer Control Unit (CCU)
- Adding Program Control Flow to a CCU
- Improving CCU Speed
 - Further Improvements in Microprogram Control Flow
- CCU Implementations Using Am2900 Family Devices
- Supersequencer Am2910
- Microprogrammed Sequencers Am2909/2911
 - Next-Address Control Am29811
- Microsequencer Am29112
- Microcycle Timing for Am2910

**INTRODUCTION TO DESIGNING WITH THE Am2900 FAMILY
OF MICROPROGRAMMING BIPOLAR DEVICES**

EDUCATIONAL OBJECTIVES

1. Understanding the digital-computer, machine-instruction sequencing process (macro level) and associated architecture at the lower level (micro level).
2. Appreciation of digital-computer control-unit organization for machine-instruction sequencing and its implementation with Am2900 family devices.
3. Appreciation of digital-computer, arithmetic/logic unit (ALU) organization and its implementation with Am2900 family devices.
4. Understanding microprogramming terms (mnemonic programming at the micro level).
5. Understanding Am2900 family support devices for constructing an instruction sequencing system at the micro level.

I N T R O D U C T I O N

WELCOME TO THE WORLD OF MICROPROGRAMMING AND THE Am2900 FAMILY

1-40

ED2900A

1-40

ED2900A EDUCATIONAL GOALS

" Introduction to the Advanced Micro Devices (AMD) Am2900 family of devices and their use. "

DAY 1

- Introduction to bit-slice architecture, microprogramming, microprogram sequencers (controllers) and their use.

DAY 2

- Study of arithmetic/logic units (ALUs), their use (algorithms) and interfacing to sequencers

DAY 3

- Analysis of support chips for systems support and specialized applications:

Devices for dealing with interrupts

Register expansion for ALU's

Registered PROMs for ALU's

Shift and status control devices

Microprogrammable clocks

16-bit and 32-bit ALUs

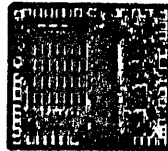
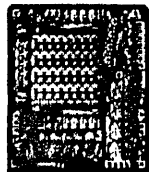
TECHNOLOGY TRENDS

- EARLY
1960s** SMALL SCALE INTEGRATION (SSI), 2-10 GATES PER CHIP.
- NAND gates
 - NOR gates
 - XOR gates
 - NOT gates (inverters)
 - Individual flip-flops (storage)
 - 256-bit RAM
- LATE
1960s** MEDIUM SCALE INTEGRATION (MSI), 20-100 GATES PER CHIP.
- Registers/Latches
 - Decoders/Encoders
 - Multiplexers
 - Adders/ Comparators
 - Arithmetic/Logic Units
 - 1K-bit RAM
- 1970s** LARGE SCALE INTEGRATION (LSI), 200-500 GATES PER CHIP.
- RALU-Arithmetic/Logic Unit (ALU) with registers
 - Interrupt controller/Direct Memory Access controller
 - Microprogram sequencer/Macro program controller
 - Memory controller/Input-Output controller
 - Microprocessors
 - 16K-bit RAM
- 1980s** VERY LARGE SCALE INTEGRATION (VLSI), MORE THAN 1000 GATES.
- 16-bit Bipolar and MOS ALUs
 - 16 and 32-bit Bipolar and MOS microprocessors
 - Multi-mode arithmetic on expandable RALUs
 - Special Data Manipulation (FFT, Signal processing, ooo)
 - 256K-bit RAM

Example of Bipolar Speed/Density Improvements

Am2901 FOUR-BIT MICROPROCESSOR SLICE

540 GATES
800mW
40-PIN DIP



DIE SIZE	Am2901 33,000 MILS ²	Am2901A 20,000 MILS ²	Am2901B 15,000 MILS ²	Am2901C 15,000 MILS ²
SPEED A, B G, P	80ns	65ns	50ns	37ns
TECHNOLOGY	LOW-POWER SCHOTTKY	DUAL LAYER METAL ION- IMPLANTATION	PROJECTION PRINTING	ECL INTERNAL TTL I/O IMOX
	1975	1977	1978	1981

TECHNOLOGY TYPES

(see Am2900 Family Data Book and Figure on next page)

" OF WHAT IS THE ACTUAL SEMICONDUCTOR CHIP MADE? "

- **BIPOLAR** - Earliest technology

- Fastest technology

- Transistor-Transistor Logic (TTL)

- Emitter-coupled Logic (ECL)

- Ion-implanted oxide-isolation (IMOX)

- TTL external/ECL internal

- IMOX used in Am2900 family

- **MOS** - Developed later than Bipolar

- Higher chip density

- Slower speed relative to Bipolar Technology

- Used in microprocessor chips (e.g. Intel 80286, Z8000)

PROBLEM

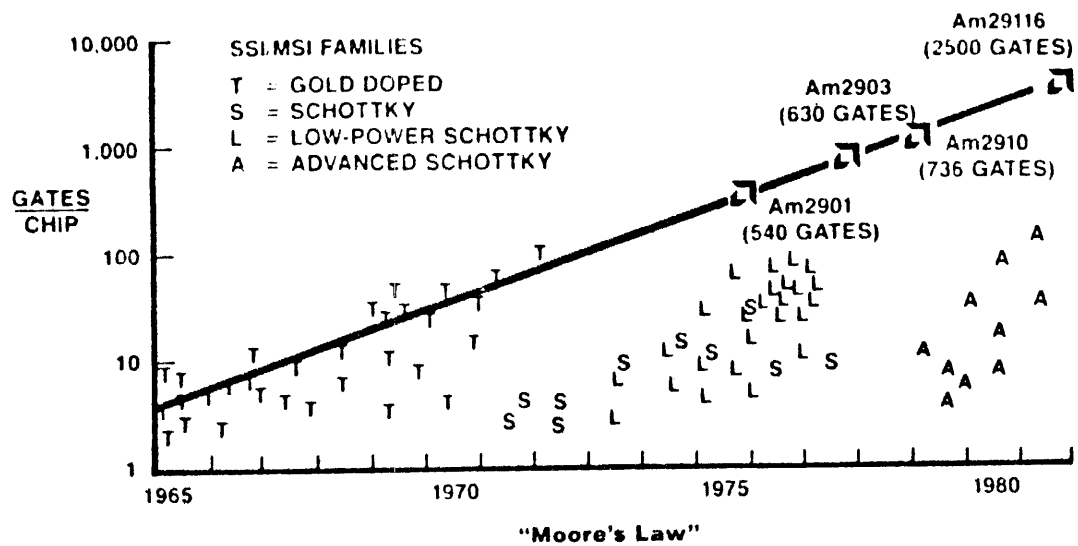
How do you build a large circuit (e.g. a microprocessor) with bipolar speed if it won't fit on a single chip?

SOLUTION

Use a bit-slice architecture!

Bipolar Density Improvements

Am2900 Bipolar LSI/VLSI

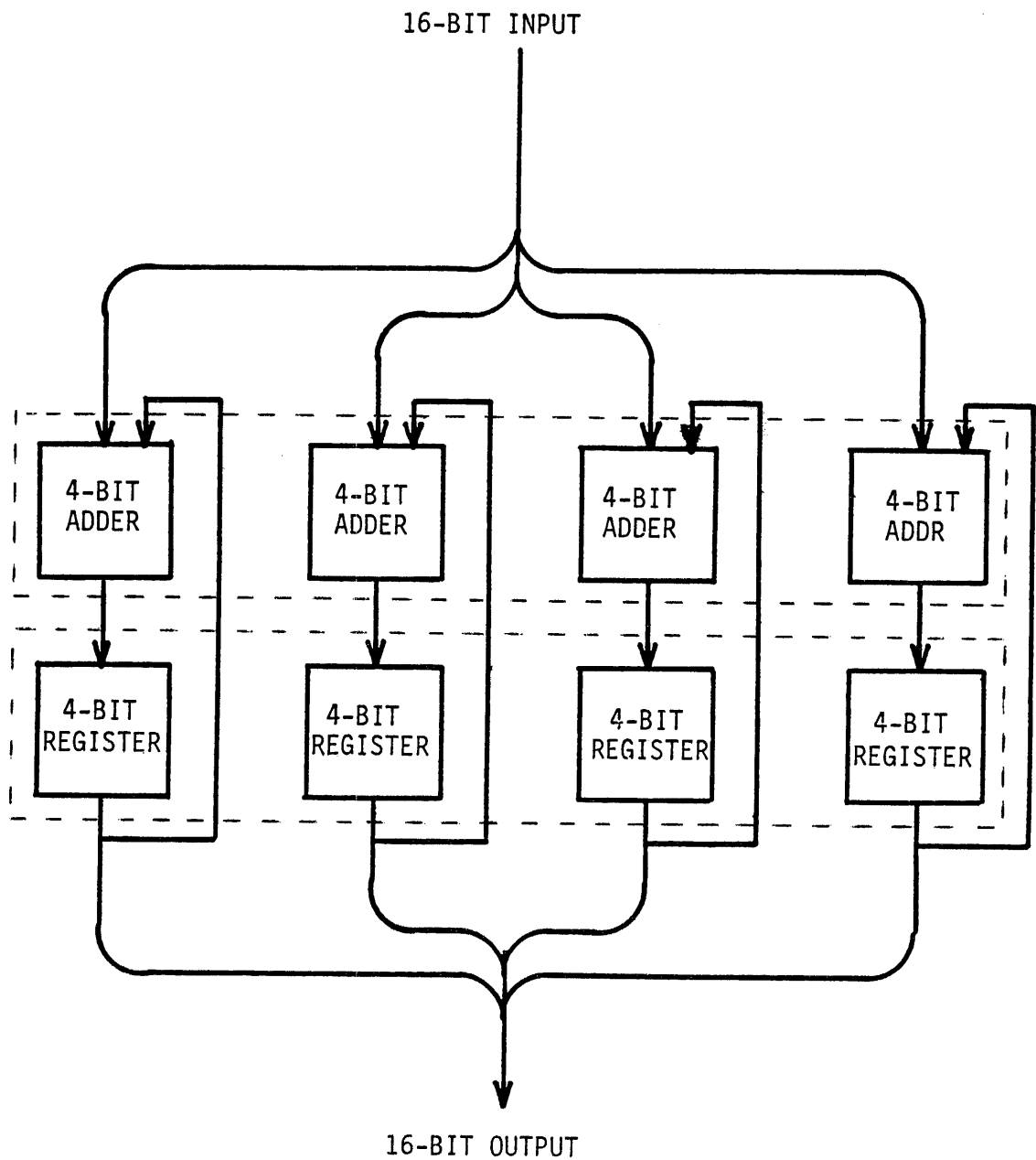


Moore's Law - Gates/chip increases by a factor of four approximately every two years.

BIT-SLICE ARCHITECTURE

- Since chip density is limited, a small processor chip (typically four bits wide) is made in such a way that several of these chips can be hooked together as building blocks to make a larger (8-, 16-, 24-, 32-, 64-bit) processor. This is defined as **bit-slice** architecture.
- This hardware implementation requires special features to handle problems like carry overflow, sign bit, etc. that involve data movement between slices.
- Note that the term microprogramming has not yet been defined. Microprogramming and bit-slice are two separate concepts, although they are closely related in most of the Am2900 family. **Bit-slice** generally refers to the structure of various devices and how they are connected. **Microprogramming** concepts involve the method by which these devices and others are controlled.

16 BIT ADDER/REGISTER



THERE ARE THREE BASIC IMPLEMENTATION CHOICES (LEVELS):

- SSI/MSI Hardware
- Bit-slice (LSI/VLSI) "Firmware"*
- MOS Microprocessors (LSI/VLSI) Software

* "0"s and "1"s stored in a Read-Only Memory (ROM)

USE BIT SLICES TO BUILD SYSTEMS

- **MACHINES WITH LONG WORD LENGTHS**

16, 24, 32, 36, 64 bit words and beyond

- **MACHINES WITH SPECIAL MACRO LEVEL INSTRUCTION SETS**

Emulators - such as Nanodata QM-1

MIL STD 1750 computers

Controllers

- **FAST MACHINES - 100ns cycle times**

Real-time data control

Real-time complex arithmetic

WORD LENGTH

<u>FIXED INSTR.</u>	<u>BIT SLICE</u>	<u>SSI/MSI</u>
4,8, or 16 bit fixed	any multiple of 4	any length

CHIP COUNT FOR SIMPLE SYSTEM

<u>FIXED INSTR.</u>	<u>BIT SLICE</u>	<u>SSI/MSI</u>
3-6	30-60	100-500

ARCHITECTURE

<u>FIXED INSTR.</u>	<u>BIT SLICE</u>	<u>SSI/MSI</u>
pre-determined	largely user defined	completely user defined

INSTRUCTION SET

<u>FIXED INSTR.</u>	<u>BIT SLICE</u>	<u>SSI/MSI</u>
pre-determined; primitive	user-defined in firmware	user-defined firmware/ hardware

CONCLUSIONS

<u>FIXED INSTR.</u>	<u>BIT SLICE</u>	<u>SSI/MSI</u>
cheapest: use 9080A whenever possible	use whenever high speeds and/or unique instructions are needed	fastest: use Schottky MSI where very high speed is a must

IF YOU'RE GOING TO BUILD A BIPOLAR MACHINE, YOU SHOULD USE

LSI

- LSI reduces costs (less chips and connections)
- LSI improves reliability (fewer total pins)

IF YOU'RE GOING TO BUILD A BIPOLAR MACHINE IT SHOULD BE

MICROPROGRAMMED INSTEAD OF USING HARDWARE LOGIC

(Microprogramming is a level above hardware logic)

- Easier design, using application-specific variable names and operations
- Easier implementation
- Easier testing
- Easier maintenance
- Better documentation (easier to understand)

MICROPROGRAMMING

- In order to appreciate the position of the microprogramming level (micro level) in systems design consider the ...

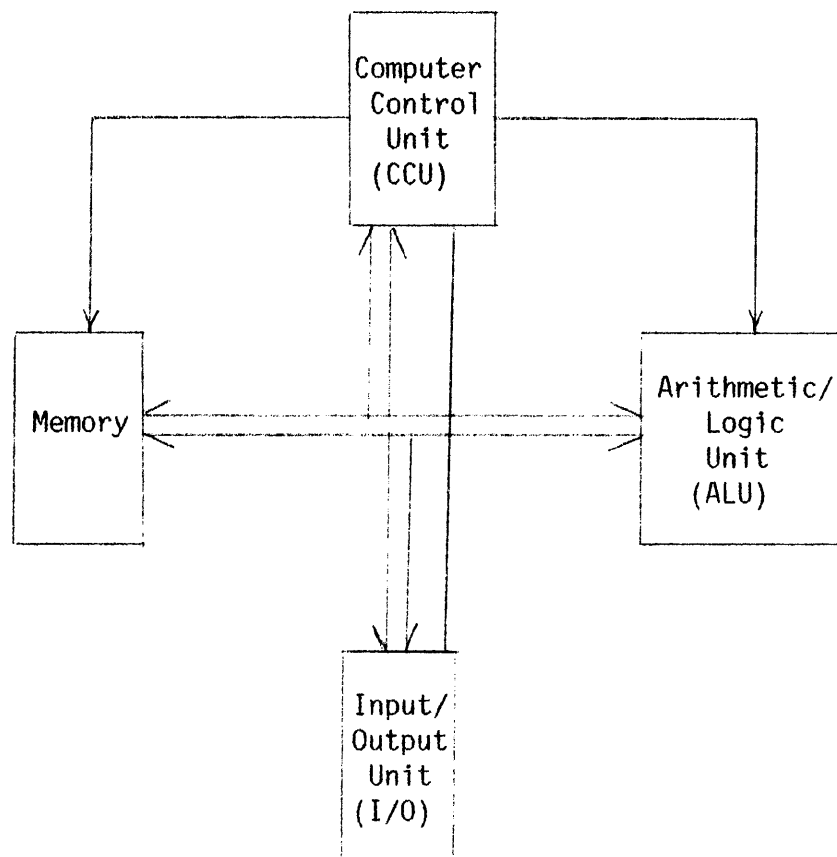
HIERARCHY OF COMPUTER ALGORITHM DESCRIPTIONS/LANGUAGES

- Higher-order languages (compiler/interpreter translators)
- Lower-order languages (assembler translators)
- Machine language (macro level)
- Register-transfer languages-RTL (microprogramming)
- Boolean algebra (symbolic logic - state diagrams)
- Logic levels (timing diagrams - waveforms)

Note: One can design, implement and test algorithms on any one or more of the above levels, the choice depending upon application and constraints. Specific languages at each level are used to define a desired algorithm as well as its implementation. Various design approaches using some of the above languages are employed in this course.

**MICROPROGRAMMING IS A TECHNIQUE FOR
DESIGNING COMPUTER CONTROL UNITS (CCUs) FOR CONTROLLERS**

- Instead of defining information movements and manipulations in terms of Boolean algebra, they are described on a higher symbolic level using register and arithmetic/logic operation designations (register transfer language-RTL). With Boolean algebra, all hardware operations are described at the logic level. RTL permits a more concise description of the desired process using names and operations reflective of the original design process.
- Initially consider computer control as an example of a microprogrammed architecture, i.e.



MICROPROGRAMMING DEFINITIONS

- Using a register transfer language (i.e. microprogramming) to define desired information movements and operations permits the system to be developed with a hierarchical modular (chip and firmware-RTL) structure. For example, ALU bit-slice chips are given a coded CCU command, such as ADD Register 2 to Register 1. The ALU bit-slice chips then execute the operation internally with the CCU not having to control the exact step by step addition process
- Microprogramming then consists of defining in an encoded fashion using system variables (registers/variables operations), a step-by-step process of information movement and manipulation. The mnemonic microprogram is then decoded into zeros and ones and is put into a PROM. Each line statement or sequence of ones and zeros of the PROM program is sent to the ALU or other system chips under clock control for proper sequential execution.

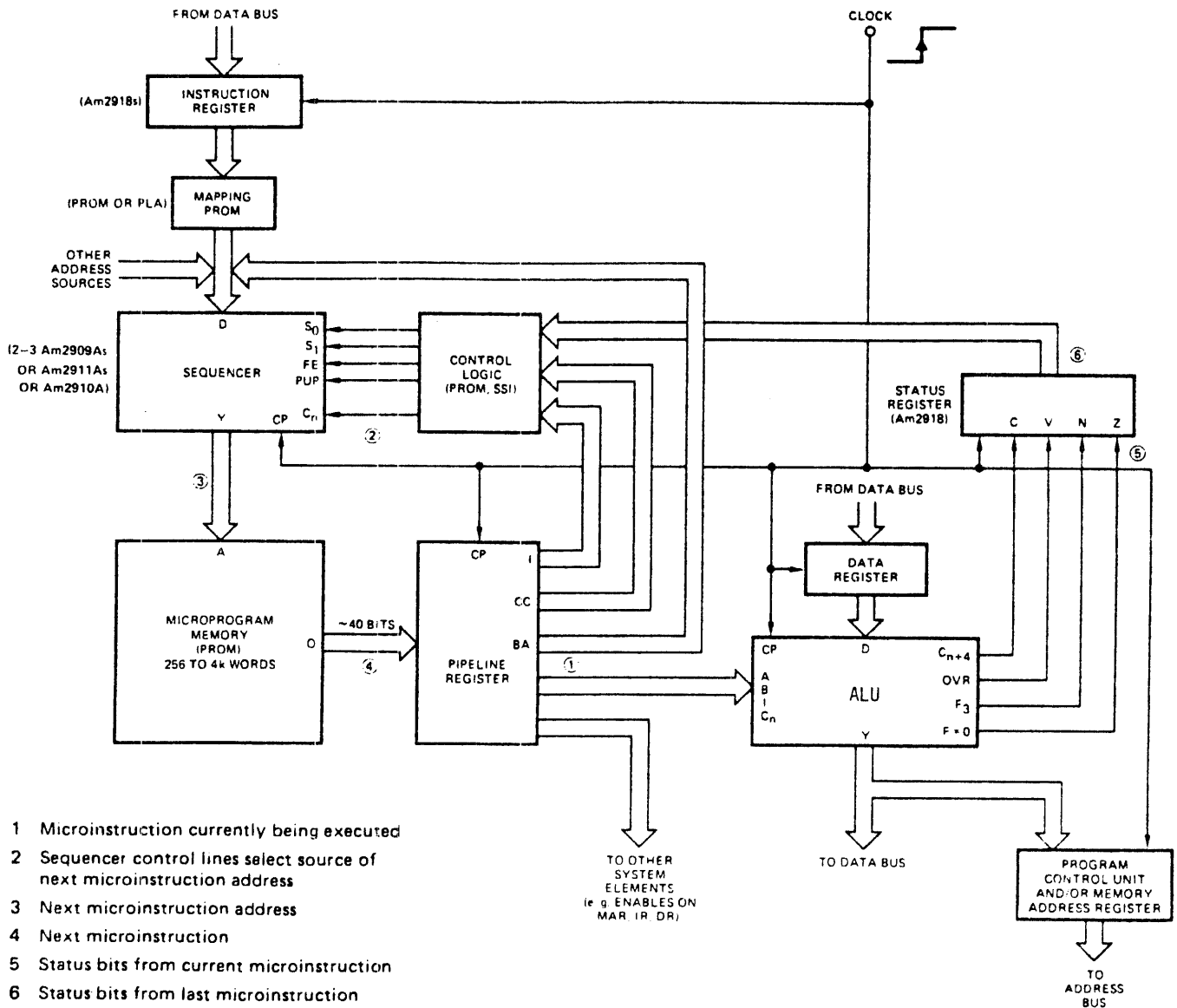
Microcode

```

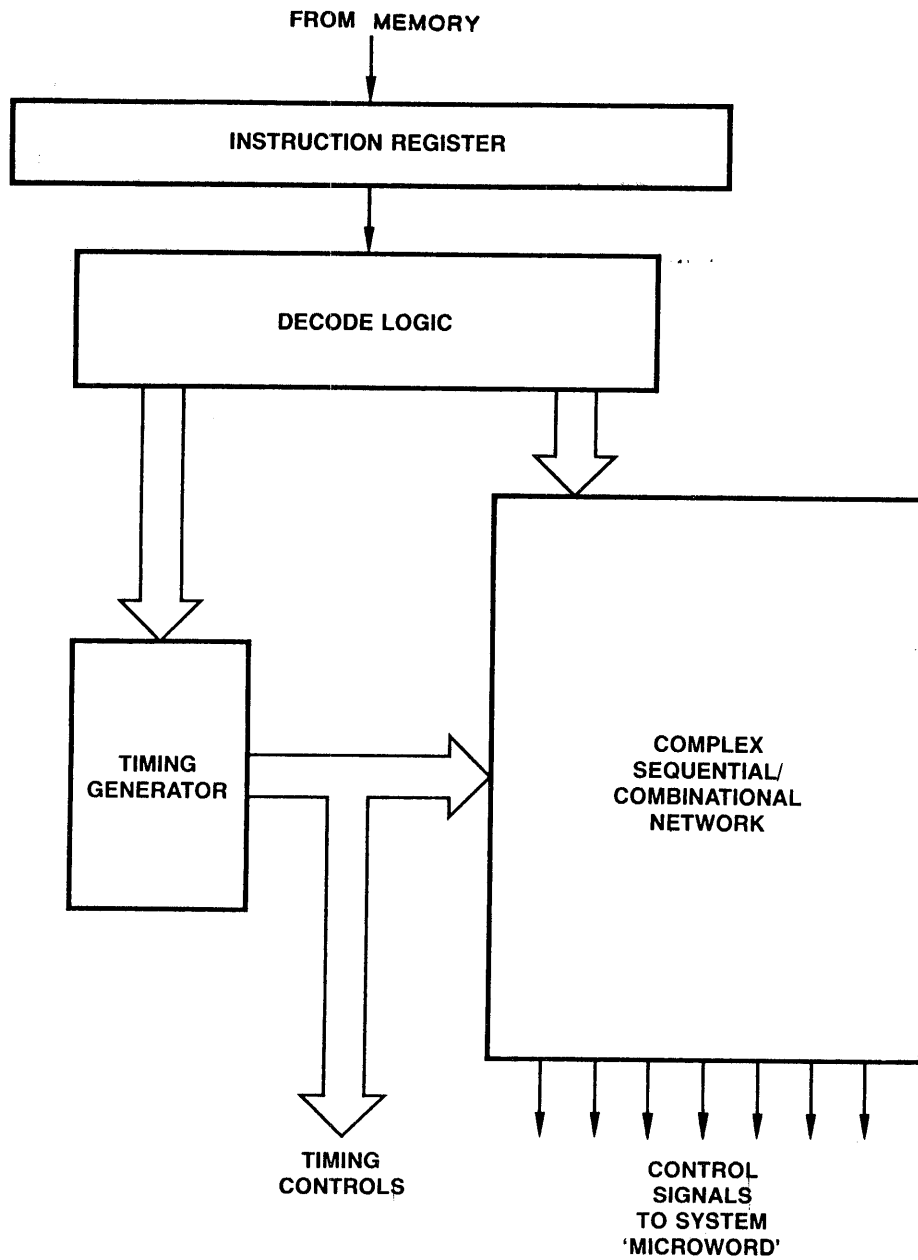
      .
      .
      .
110100010
010101011    bit patterns control
001100111    individual logic gates
110011000
      .
      .
      .

```


GENERAL MICROPROGRAMMED SYSTEM



TRADITIONAL HARDWIRED CCU:



HARDWIRED CCU

Advantages

- May be faster solution (execution time)
- Custom designed for the specific problem
- May be smaller (part count and size)

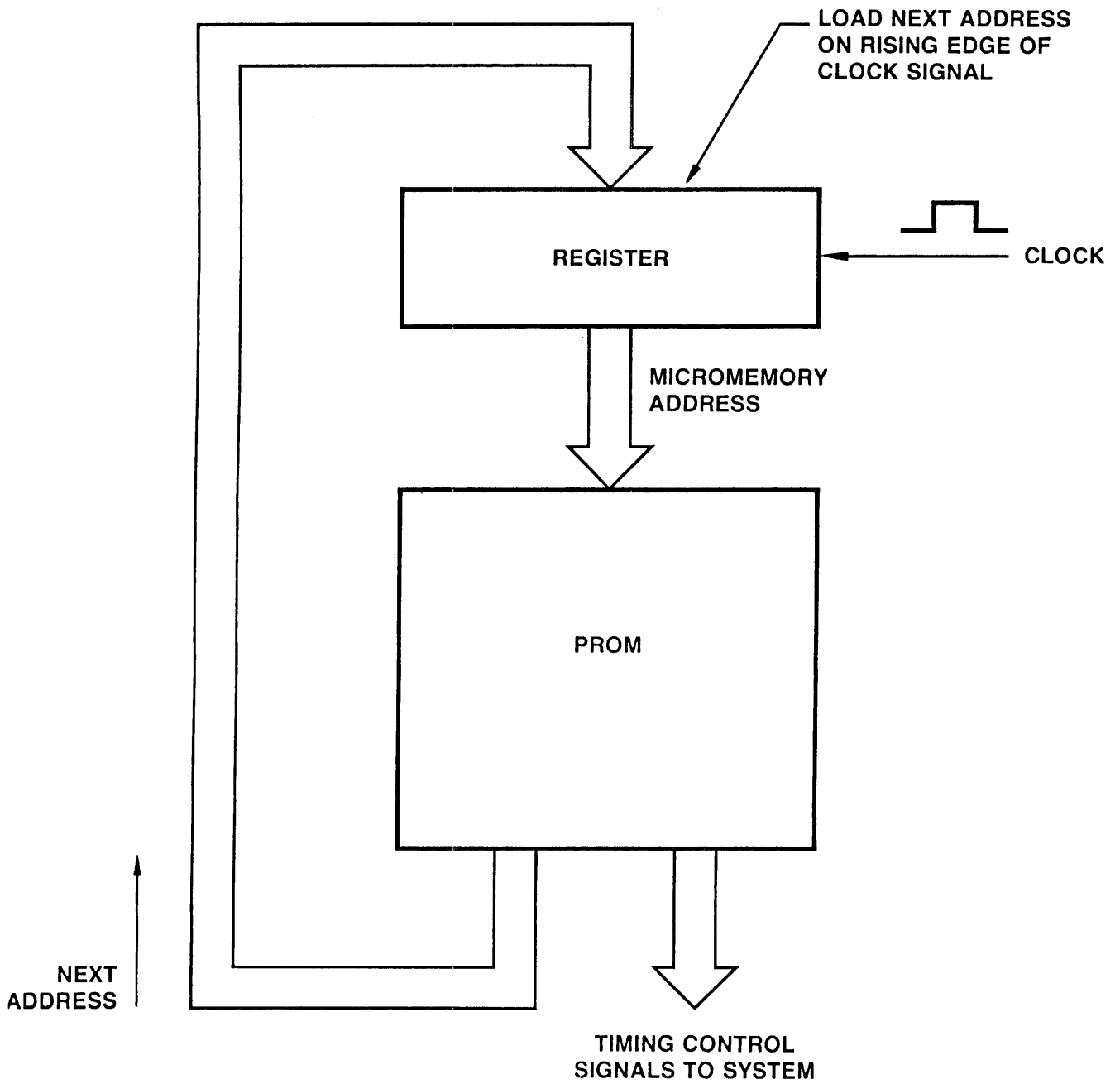
Justification

- Suitable if design is rigid or fixed for high volume production

Disadvantages

- Lengthy design time with Boolean algebra descriptions (logic equations)
- Bulky documentation - long parts lists, detailed logic schematics, etc.
- Any changes require partial or total redesign
- Pin count, board space high
- Board may have very limited modular structure (modularity in design layout is difficult)
- Testing difficult - minimization effort is difficult
- Debug at logic level is more complex than for LSI solutions

THE SIMPLEST CONTROL UNIT
CCU - Computer Control Unit



MICROPROGRAMMED CCU:

- CCU memory, usually programmable read-only memory (PROM), contains a sequence of "microinstructions"

- Each microinstruction contains two parts:
 - microinstruction sequencer portion contains CCU memory address of next word

 - controller portion contains control bits for system

Advantages

- Design now becomes a programming effort (software engineering)
- Development time shortened with appropriate tools
- Major documentation contained in program listings
- Changes may require little or no redesign
- Part count small (mainly memory)
- Modular, structured techniques can be easily applied
- Testing and debugging are easier

Disadvantages

- May be slower than hardwired CCU

WHY MICROPROGRAMMING IS BETTER

- More structured organization
 - random hardware logic is replaced by zeros and ones in a memory (PROM)

- Field changes are easy - PROM replacement

- Adaptations are easy (extendability) - additional PROMs

- System definition can be expanded - additional chips & PROMs

- Documentation and service are easier (understandability)
 - structured, modular microcode instead of possible unstructured schematics and wire lists

LANGUAGE INTERRELATIONSHIPS

It is helpful to develop a more detailed understanding of where microprogramming fits in relation to "conventional" levels of programming.

- **High Level Languages (HLL)** - Basic, FORTRAN, Pascal, ADA, etc.
 - expressed in pseudo-math ($Z=X+Y$)
 - converted to machine language (ML) by compiler/interpreter
 - each HLL statement translates into many ML statements
 - user is largely isolated from the particular hardware system
 - fixed instruction set (FIS)

- **Assembly Language**
 - expressed in mnemonics (ADD R1, R2)
 - converted to machine language by assembler
 - ratio to machine language statements is usually 1:1
 - user no longer isolated from knowledge of system hardware
 - fixed instruction set (operations and format)

- **Machine Language**

- expressed in binary code (01101110)
- each machine language instruction interpreted by a microprogram routine
- fixed instruction set (operations and format)
- knowledge of system hardware

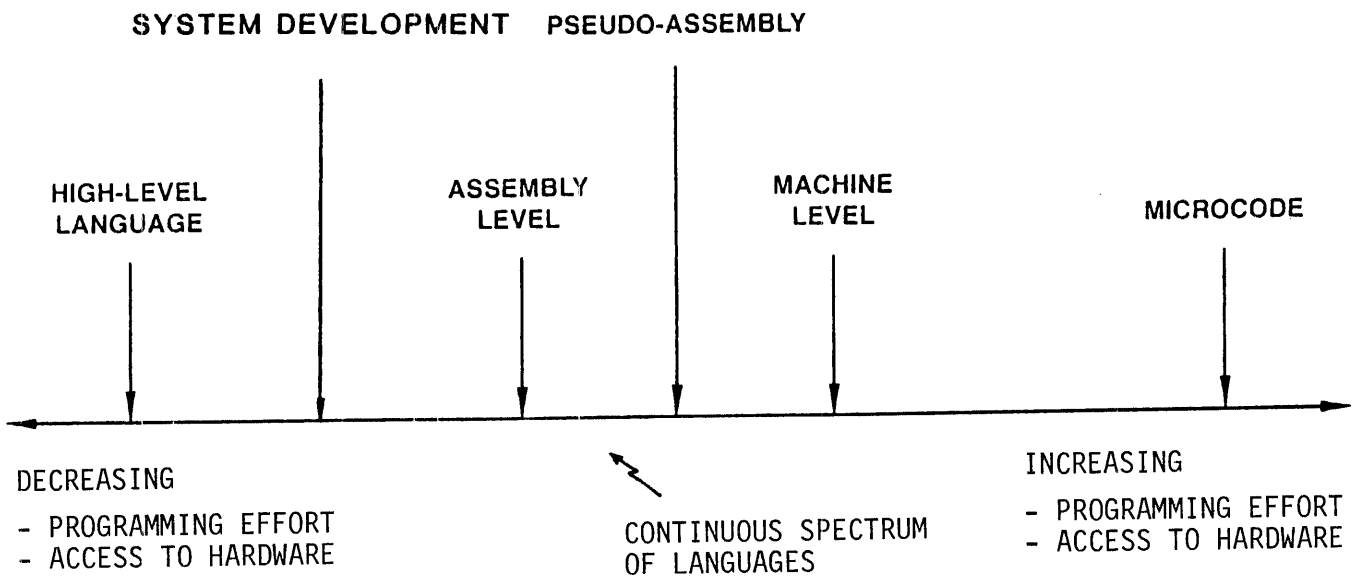
- **Register Transfer Language (Microprogramming)**

- direct control of hardware at register transfer level
- must know complete system hardware
- format of microprogram instruction statements defined
- microprogramming often stored in PROM (firmware)

- **Boolean Language (Hardware logic)**

- logic function realization in SSI/MSI circuits

LANGUAGE RELATIONSHIPS



COMPARING LANGUAGE IMPLEMENTATIONS

BASIC	ASSEMBLY 8080A	MACHINE 8080A (HEX)	COMMENTS
READ A, B, C			
	IN CRD	DB 05	INPUT FROM CARD
	MVI H, ADRH	26 00	
	MVI L, ADRL	2E 40	
	MOV M, A	77	CRD -> MEM - A
	INX HL	23	INCR ADDRESS
	IN CRD	DB 05	
	MOV M, A	77	CRD - > MEM - B
	INX HL	23	
	IN CRD	DB 05	
	MOV M, A	77	CRD -> MEM - C
LET A = A + B - C			
	MVE L, ADRL	2E 40	RESET ADDRESS
	MOV A, M	7E	LOAD ACC <- A
	INX HL	23	
	ADD M	86	ADD ACC <- ACC + B
	INX HL	23	
	SUB M	96	SUB ACC <- ACC - C
	MVI L, ADRL	2E 40	RESET ADDRESS
	MOV M,A	77	ACC -> MEM - A

- Note that each Basic statement translates into 10 or so assembly language instructions and each assembly instruction translates into 1 or 2 words at the machine level.
- No attempt was made to make the assembly program efficient.
 - the intent was to translate directly from the Basic statements (one at a time)

MICROPROGRAMMING DEFINITIONS

Microstore (control store, micromemory)

- The CCU memory (often ROM or PROM) where microprograms are stored.

Microprogram

- A logically related sequence of microinstructions and/or microroutines.

Microroutine

- A sequence of one or more microinstructions which control a functional task (may implement one macroinstruction, for example).

Microinstruction

- The combination of all micro-operations or fields that specify the state of all control lines during a time interval (clock cycle).

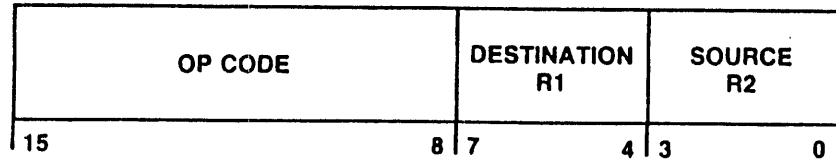
Micro-operation

- The combination of one or more fields to control one functional unit, such as the ALU.

Field

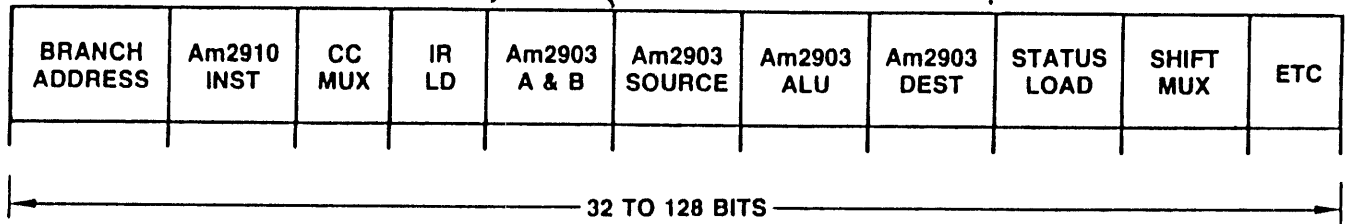
- One or more bits (binary digits) as needed to define a specific hardware activity for a functional unit such as an ALU arithmetic operation.

MACHINE LEVEL INSTRUCTION



MICRO-OPERATION

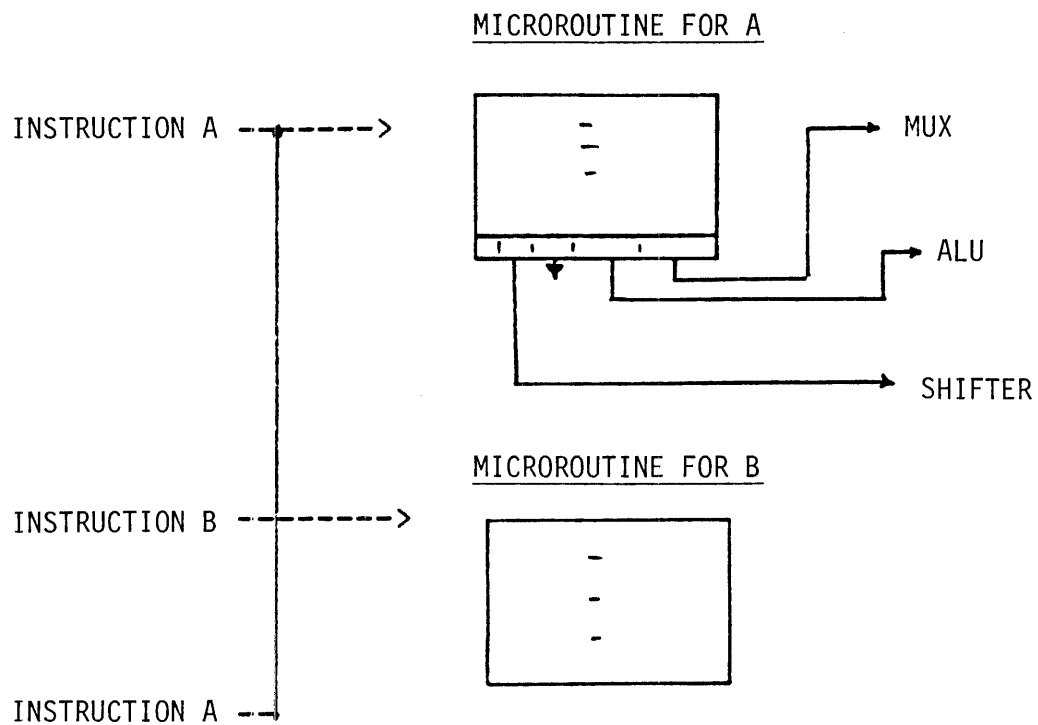
MICRO-INSTRUCTION



INSTRUCTION REGISTER:

MICROPROGRAM

HARDWARE



Each machine instruction causes a specific microroutine to be executed.

MICROINSTRUCTIONS

- The microword is typically very wide (48-128 bits) because of the large number of control signals required to control system resources (functional units).
- The microprogrammer and detailed hardware designer, if not the same person, must work as a team to define the required microword fields (hardware/firmware/software interface fuzzy!)
- The microinstruction format is defined by these individuals.
- There are no fixed rules with regard to format layout or limits on the number of formats permissible. Objectives should include ease of understanding, readability, testing, flexibility and extendability and the associated development of good documentation.

SUGGESTED PRACTICES FOR MICROINSTRUCTION FORMATTING

- Use logical fields to increase readability. Worry about physical layout later. There are development tools to help in implementation.
- Minimize the use of shared or overlapped fields (use horizontal format), as they reduce understandability.
- Group fields as to the hardware functional unit micro-operations which they control for readability and understanding.
- Group all micromemory next address fields at one end of the microword for readability.

**DEVELOPMENT SYSTEMS
FOR AIDING MICROPROGRAM DEVELOPMENT**

- META assembler - converts mnemonics to 1's and 0's. Initially requires a **definition** of microinstruction format and mnemonics (registers, operations). Then a microroutine (**source**) using the specified format and mnemonics is translated into 1's and 0's appropriately.

- Microprogramming shortens the development effort considerably.

- A development system simplifies debugging (error finding)
 - of microcoded routines
 - of hardware functional units and connections

- Aids documentation by producing human readable code
 - "mnemonics"

MICROPROGRAMMED CCU ADVANTAGES REVISITED:

- Speeds comparable to Schottky TTL
- Custom design at an RTL level (mnemonics versus Boolean logic)
- Compact unit (less space) with LSI circuits
- Changes may be "firmware" changes (in PROMs) rather than physical changes
- LSI supports a structured organization
- LSI has better reliability
 - approximately 80% of failures in the field are due to external connection failures (pins, etc)
- Microprogramming the control portion (CCU) allows:
 - hardware and firmware being designed in parallel
 - better documentation (structured microprogramming!)
 - development systems for microprogram development
 - development systems for prototype check-out
- Overall better potential for better documentation
 - understandability
- Potential for better diagnostics
 - separate switchable PROM
 - diagnostic routines on-board the control memory (PROM)

Summary of Design Tradeoffs

ITEM	SSI/MSI HARDWARE	2900 FAMILY FIRMWARE	MICROPROCESSOR FIS MOS SOFTWARE
architecture	any desired	almost any desired	predesigned
instruction	any desired via wiring	any desired via microprogram	predesigned may use software techniques to achieve desired set
word length	any desired	multiples of 4	fixed at 4,8,16,32
execution speed	100-200ns cycle times		0.7 -5us cycle
physical size (controller)	500 dips small packages	50 dips medium size	3-6 dips large packages
design time	long, slow, to do correctly	parallel - fast use aids - development systems	software - fast
documentation	tedious	forced via programming techniques	
upgrades	redesign	change microprogram	change software
design cost	highest	medium	lowest
debug	various aides exist - microprogramming development systems		

IF YOU'RE GOING TO DESIGN ANY MACHINE,
USE INDUSTRY STANDARD PRODUCTS

True LSI!

Am2900 family parts
are 10 to 20 times
as complex as
traditional MSI

The Am2900 family
is designed to be
microprogrammed

"The Am2900 family is
the industry standard
for bipolar LSI"

THE Am2900 FAMILY ELEMENTS

- CPUs (CCU + ALU)
- Microprogram controllers/sequencers
- Bipolar memory (macro and micro levels)
- Interrupt processing devices
- Bus I/O interfaces
- Direct memory access (DMA) devices
- Timing/clocks
- Macroprogram (machine languages) controllers/sequencers
- Multipliers

SOME ELEMENTS OF Am2900 PRODUCT FAMILY

- **High speed microprogrammable registered ALUs**

4-bit slice, 16 registers	Am2901B
Higher speed 4-bit slice, 16 registers	Am2901C
Speed selected version of 2901C	Am2901C-1
Expanded function 4-bit slice, 16 registers	Am2903
Higher speed version of Am2903	Am2903A
Enhancement of Am2903A, including BCD arithmetic	Am29203
16-bit microprocessor for high speed control	Am29116
Multiport, pipelined processor, 8-bit slice	Am29501

- **ALU auxillary circuits**

Carry lookahead	Am2902A
Status and shift control unit for 2901, 2903, 29203	Am2904

- **Register file extensions for ALUs**

16-word by 4-bit two-port register file, for 2903	Am29705
Higher speed version of 29705, for 2903A	Am29705A
16-word by 4-bit two-port register file, for 29203	Am29707

- **Microprogram sequencers**

4-bit sequencer slice	Am2909A
12-bit single-chip sequencer, for up to 4k microwords	Am2910
Speed selected version of Am2910	Am2910-1
Fastest (IMOX) version of Am2910, plus deeper stack	Am2910A
4-bit sequencer slice, compact version of Am2909A	Am2911A
4-bit program control slice	Am2930
4-bit program control slice, compact version of 2930	Am2932
Interruptible sequencer, 31-deep stack, 8-bit slice	Am29112
16-way branch control unit, for 2909A and 2911A	Am29803A
Next address control unit, for 2909A and 2911A	Am29811A

-
- **Clocks**
 - Single-chip clock, microprogrammable cycle lengths Am2925

 - **Interrupt control**
 - Vectored priority interrupt controller, expandable Am2914
 - Priority interrupt expander Am2913

 - **Pipeline registers**
 - Diagnostics register, 8 bits Am29818
 - Multilevel pipeline register, 8 bits Am29520
 - Multilevel pipeline register, 8 bits Am29521

 - **Registered PROMs**
 - Registered PROM, 512 x 8 Am27S25
 - Registered PROM, 512 x 8 Am27S27
 - Registered PROM, 1024 x 8 Am27S35
 - Registered PROM, 1024 x 8 Am27S37
 - Registered PROM, 2048 x 8 Am27S45
 - Registered PROM, 2048 x 8 Am27S47

1-420

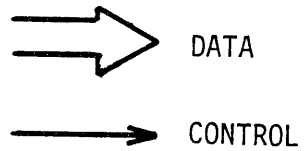
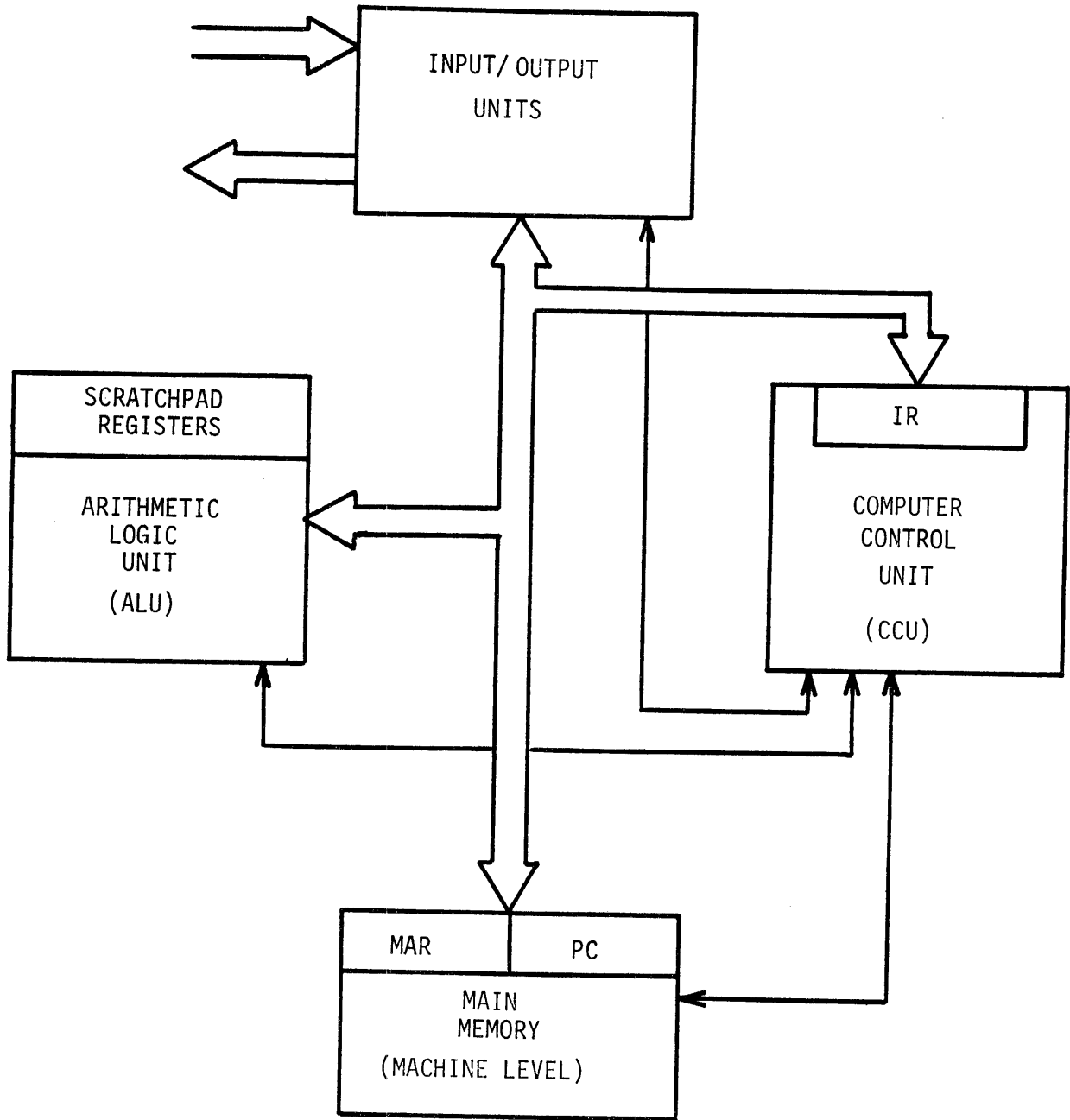
ED2900A

1-420

**ANALYZING AND DESIGNING A
COMPUTER CONTROL UNIT
(CCU)**

DEVELOPMENT OF A COMPUTER CONTROL UNIT (CCU)

- The objective of this section is to develop an understanding of the function and use of a process sequencer. In order to describe the design of a sequencer in a logical manner, something is required for the sequencer to control. While the design concepts are applicable to any kind of process control, examples of a traffic light and a coffee machine will be presented later. Initially, a digital computer macroinstruction sequencer process will be used and an associated computer control unit (CCU) developed.
- The drawing shows the classical Von Neumann/Babbage architecture (5 basic units), with a few buffer-register details. The **arithmetic-logic unit (ALU)** includes some "scratchpad" local storage registers, the **memory unit** includes the memory address register (**MAR**) and the program counter (**PC**), and the **control unit** includes the instruction register (**IR**). This register receives the next machine (macro level) instruction to be executed. It is the function of the CCU to decode the operation code (OP code) portion of the IR value and generate the sequence of control signals needed to direct the ALU, the memory and the I/O portions of the system (i.e. the system resources).

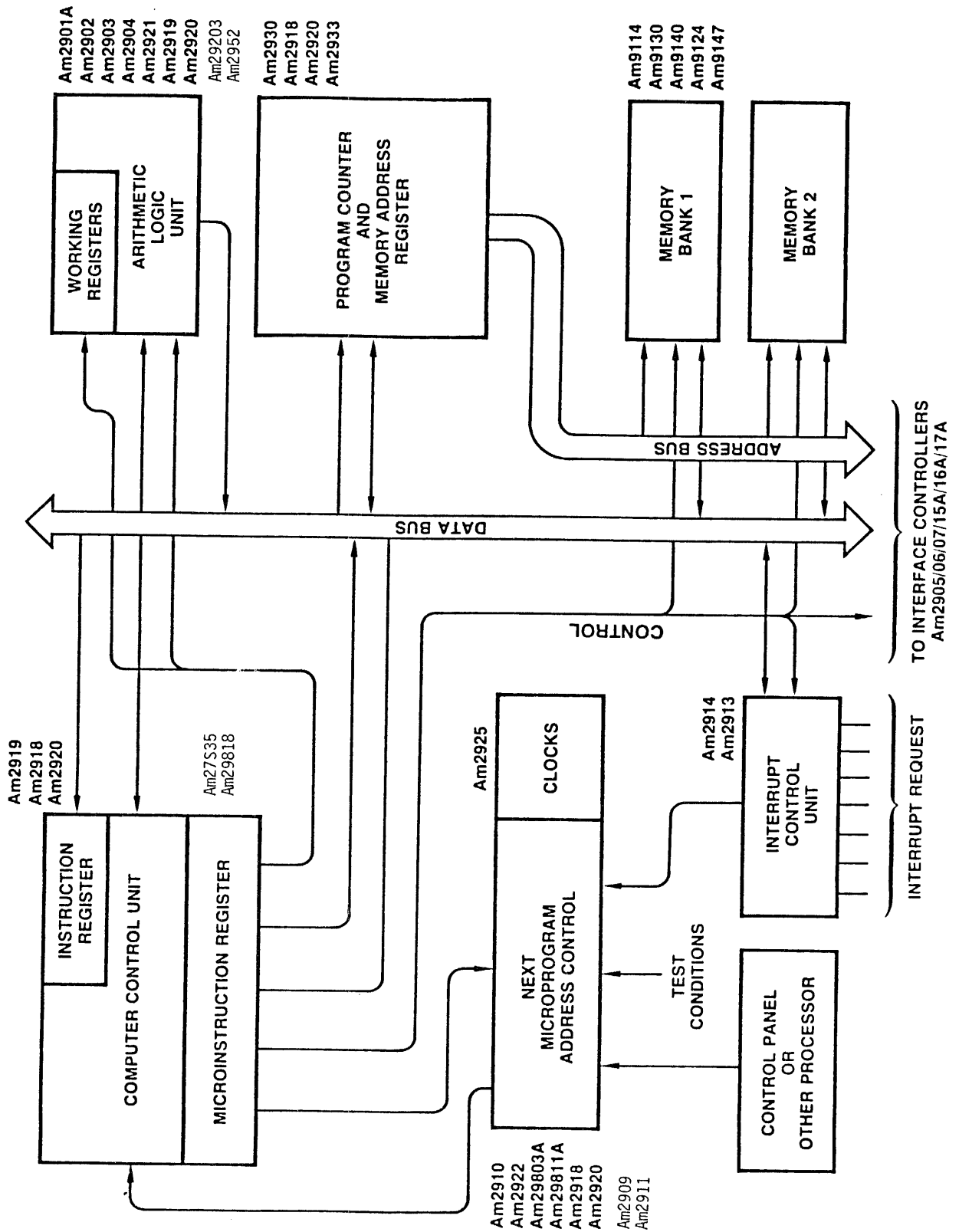


DETAIL VIEW:

- A more detailed view of this architecture shows the level of support provided by the AMD Am2900 family of parts.

- As can be seen, all of the components of a computer are supported with Am2900 chips.

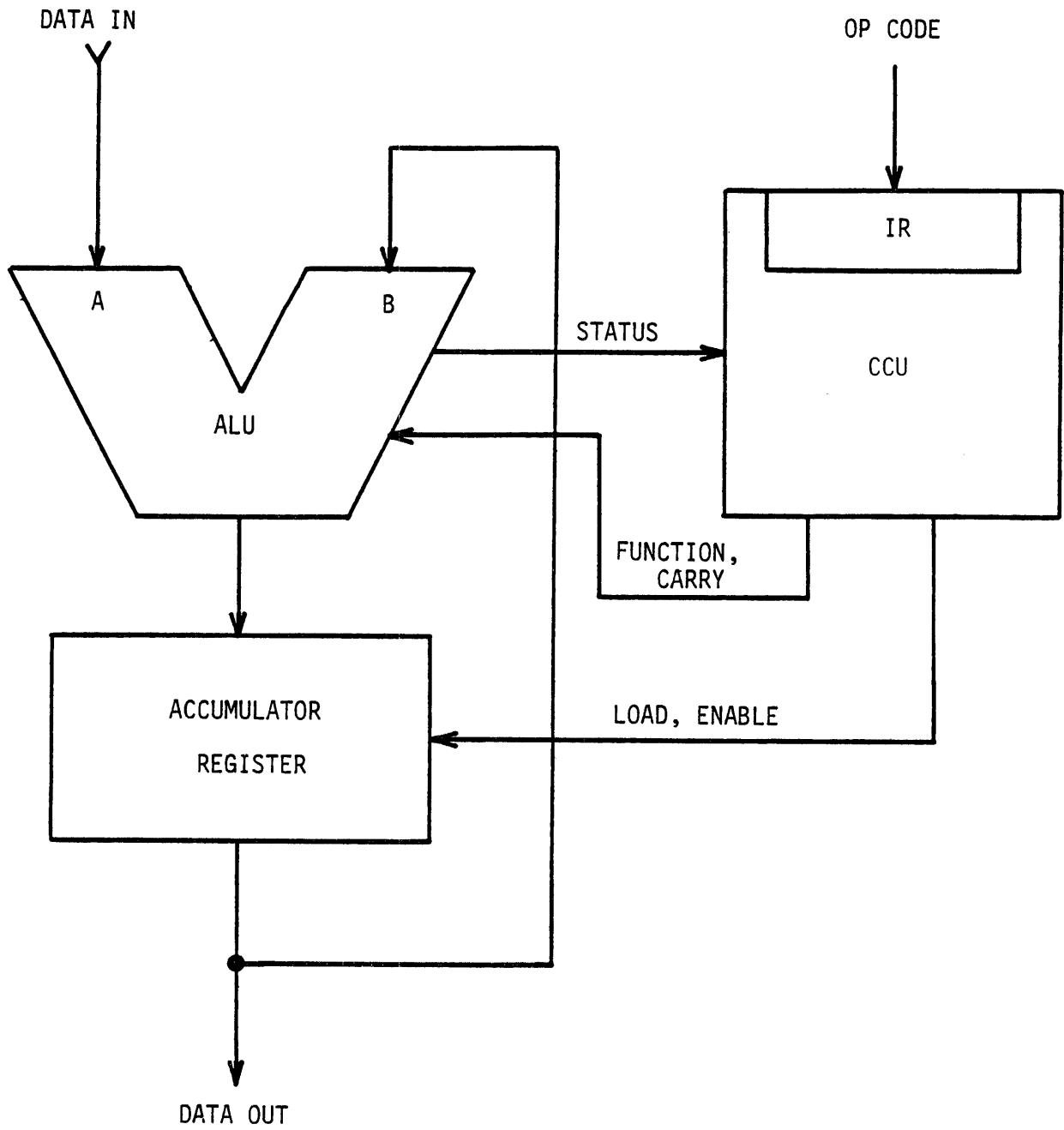
- For most of this discussion the controller portion is emphasized which is shown on the left hand side of this illustration.



SIMPLIFIED SYSTEM:

- In order to initially concentrate on the sequence controller (CCU) the remainder of the computer is simplified to
 - an ALU
 - the accumulator register (ACC)

- This architecture is defined as a single-address structure since the other address (the ACC) is implied. Thus,
 - data comes into only one side of the ALU
 - the accumulator provides the second operand
 - the result of the ALU operation is transferred to the accumulator



CONTROL SIGNALS:

- In order to define the control signals, assume the ALU can perform the functions shown on the next page. Three function control signals are required. Five basic types of instructions can be supported by the ALU, as shown.

- In addition, the ALU needs one bit to provide a 1 or 0 for the **carry-in**. This can be provided by the microword. This carry-in capability can be used in incrementing a register. Note that in a bit slice ALU configuration the **carry-out** of one slice would be connected to the carry-in of the next.

- Outputs from the ALU include the numerical result of the operation, plus various status signals. Examples include
 - carry out

 - zero

 - negative

 - overflow

CONTROL LINES	ALU FUNCTION	
S_2 S_1 S_0	$C_{IN} = 0$	$C_{IN} = 1$
0 0 0	A + B	A + B + 1
0 0 1	B - A - 1	B - A
0 1 0	A - B - 1	A - B
0 1 1	$A \vee B$	"A OR B"
1 0 0	$A \wedge B$	"A AND B"
1 0 1	$\bar{A} \wedge B$	"NOT A AND B"
1 1 0	$A \vee \bar{B}$	"A EXOR B"
1 1 1	$\overline{A \vee B}$	"NOT (A EXOR B)"

MACHINE INSTRUCTION SUPPORTED:

ADD

SUB

OR

AND

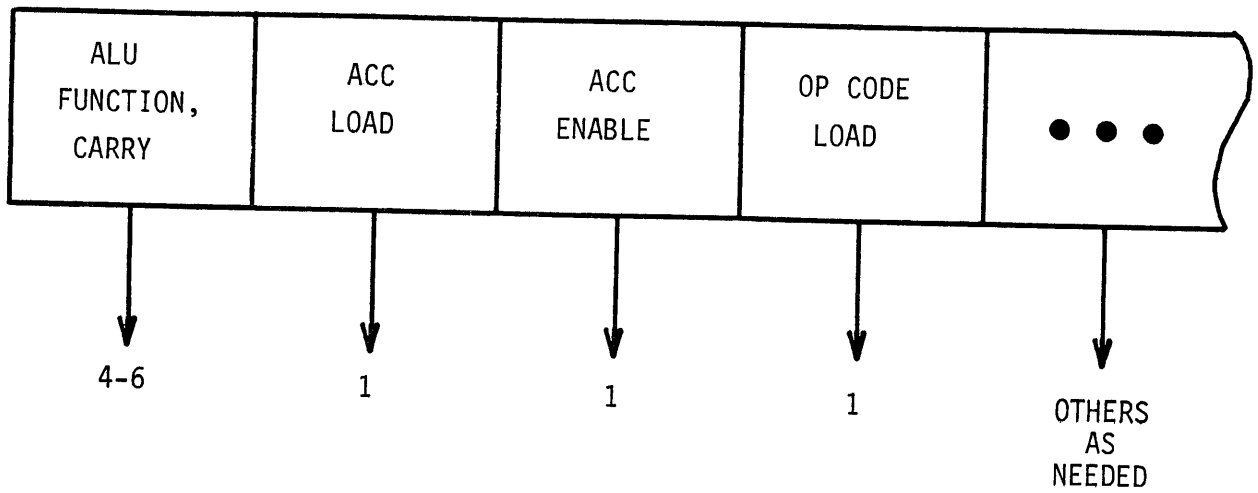
EXOR

MICROWORD FORMAT:

The following page shows the microword format to control

- ALU function select
- Carry-in
- ACC load (input)
- ACC enable (output)
- Load OP code into IR

← MICROWORD FORMAT →



SIMPLE CCU:

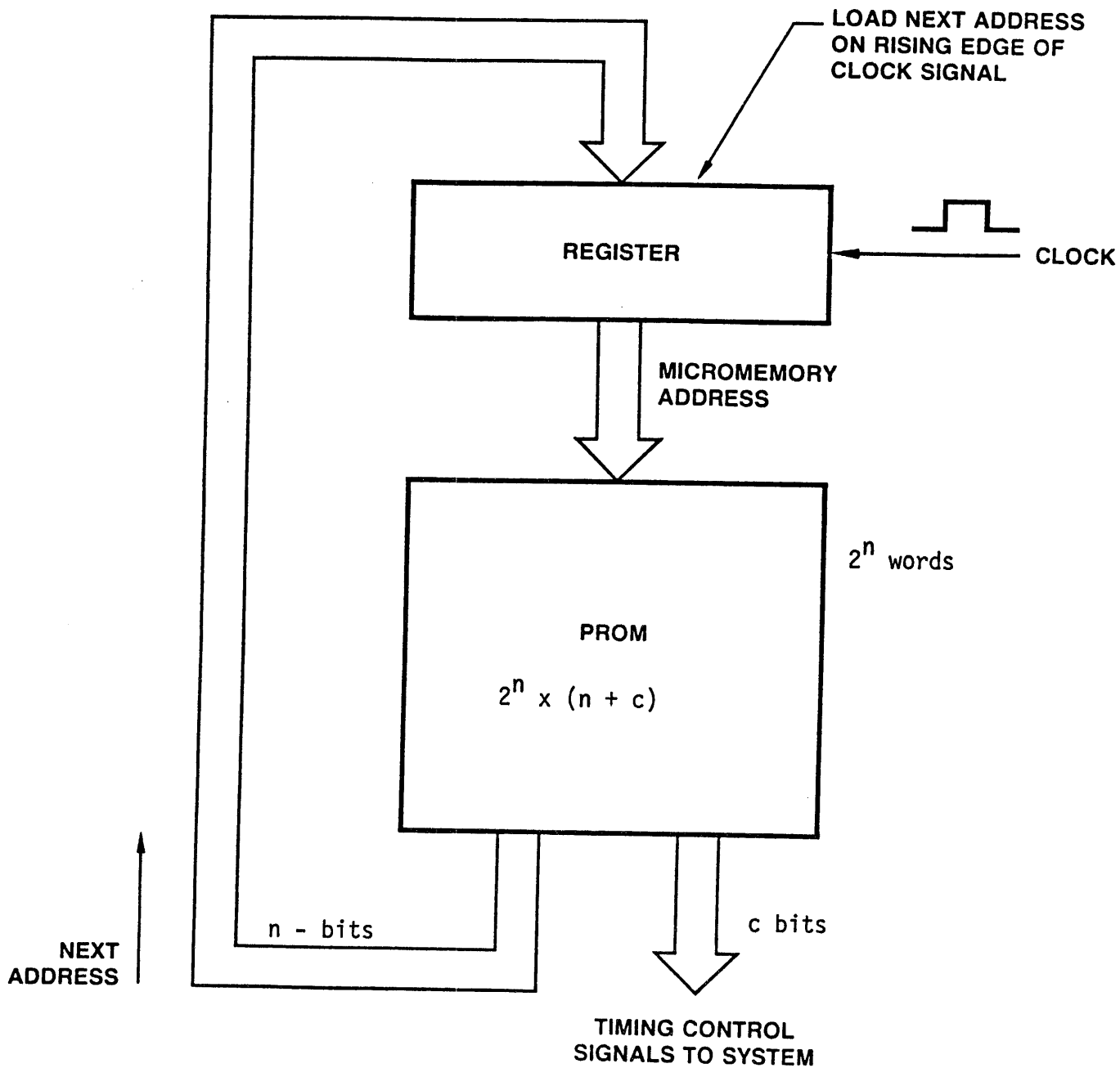
- Each microinstruction contains the address of the next microinstruction to be executed in addition to the fields for the necessary functional unit control signals. The result is a single-sequence controller (i.e. no conditional decisions). Any microinstruction can unconditionally "jump" to any other microinstruction. Usually loops are not created in this addressing mode.

The micro memory in this simple example is 2^n words deep, and m bits wide, where

$$m = a + c$$

microword width (m) = # address bits (a) + # control bits (c)

THE SIMPLEST CONTROL UNIT



TIMING DIAGRAMS

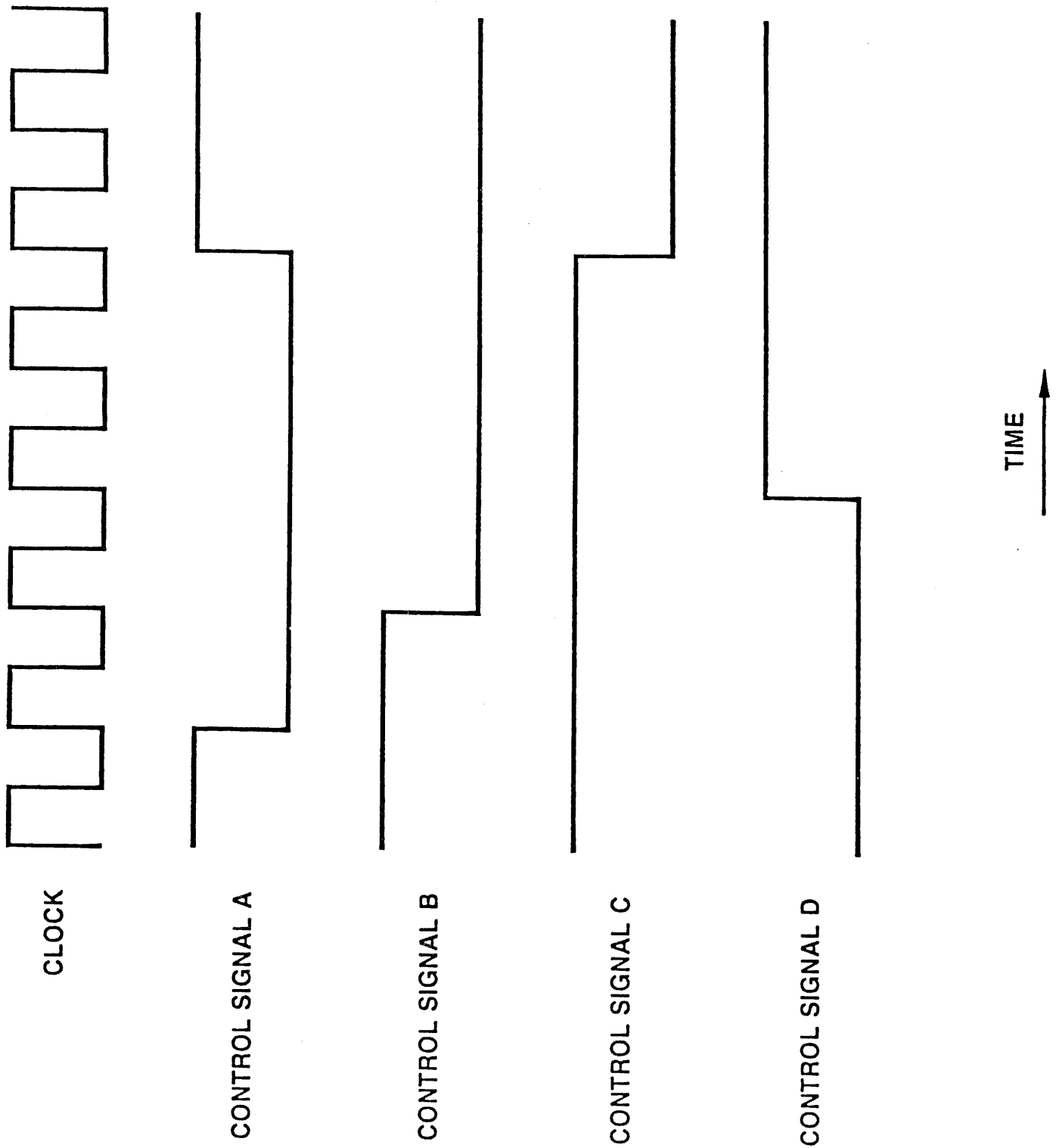
- Now, consider designing at the logic level using timing diagrams that define the desired control signal operation. Specifically consider their binary value based upon a periodic interval (clock).
 - use the rising edge of the clock as a measurement point

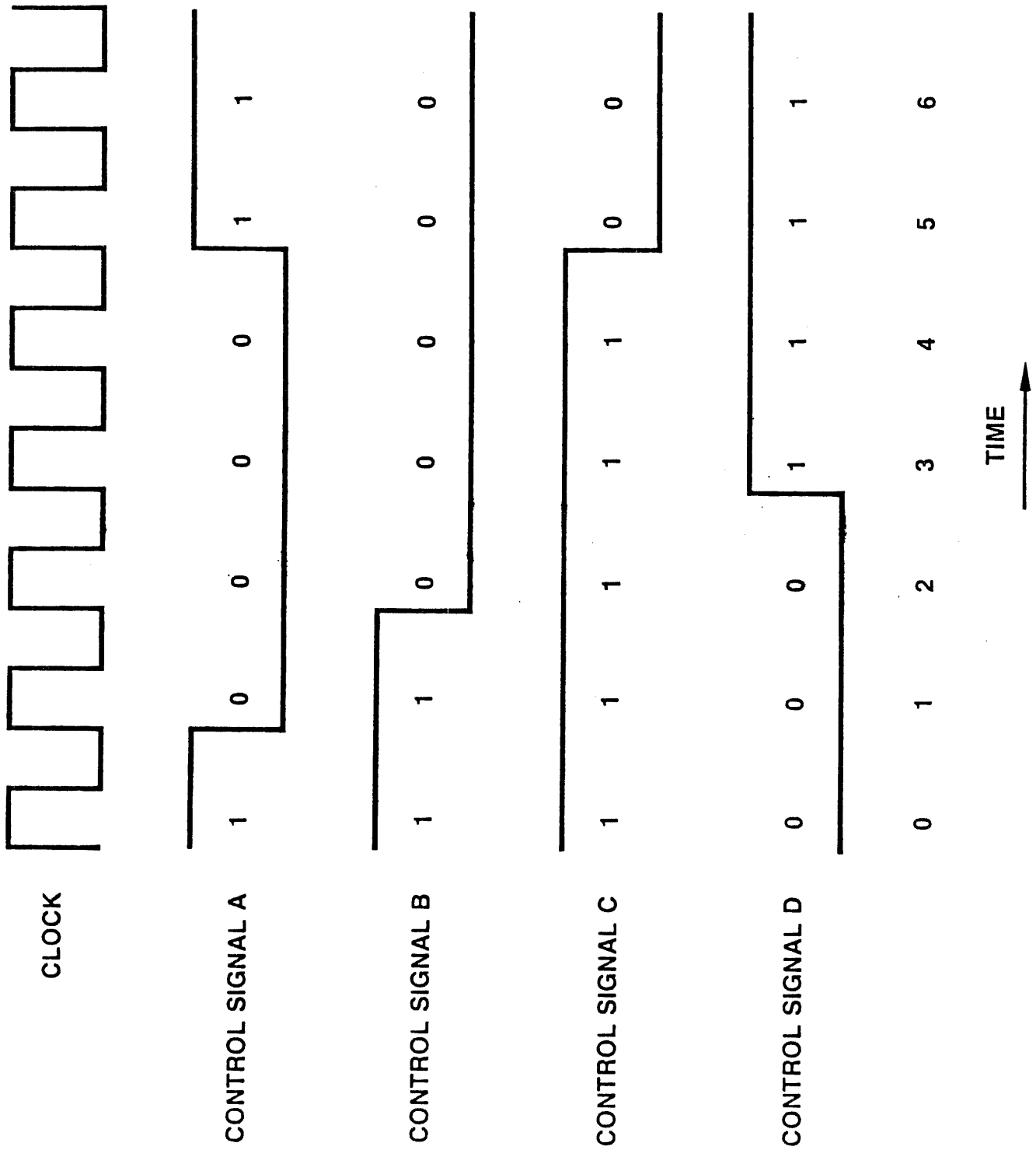
 - the bit pattern formed by the time slice is defined as the microword

- The following three pages present :
 - a timing diagram for a four-signal system

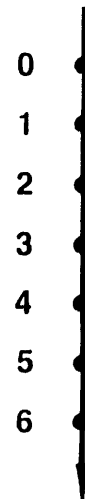
 - the timing diagram digitized on the clock edge

 - the resulting program flow and the clocked microprogram that would generate the desired timing diagram





MICROPROGRAM MEMORY ADDRESS	MICROPROGRAM MEMORY OUTPUTS			
	A	B	C	D
0	1	1	1	0
1	0	1	1	0
2	0	0	1	0
3	0	0	1	1
4	0	0	1	1
5	1	0	0	1
6	1	0	0	1

MICROPROGRAM
FLOW

This is the microcode for sequential execution.

CLASS EXERCISE

Turn to the ED2900A Exercise and Laboratory Manual

Solve the simple traffic light problem by designing at the Boolean level using a state diagram to define the sequenced transitions between each desired light condition. The associated state code of zeros and ones is then used to define the microroutine. This problem could also be solved at the waveform level by initially defining the desired transitions in terms of zero-one transitions for each control signal.

ADDING PROGRAM FLOW CONTROL TO CCU

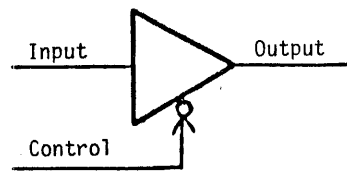
PROGRAM FLOW CONTROL ADDITION (conditional branches)

- Required microprogram flow should have the same characteristics as any computer program, i.e. **sequence** (continue), **iteration** (loop) and **decision** (branch) in order to implement an algorithm. The previous design permitted only sequential flow (a single sequence of microinstructions). Thus, the current CCU structure must be expanded to provide for these additional capabilities.

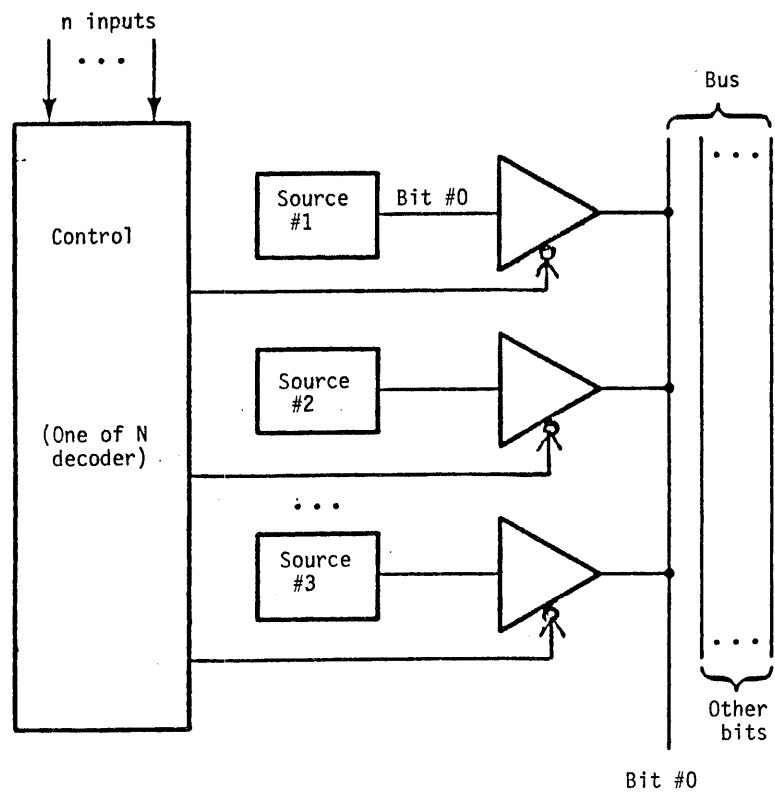
- The controller just described can execute one serial sequence of operations. In order to select from multiple sequences and to allow conditional branching, further addressing hardware is necessary. The current CCU configuration will be enhanced with additional hardware to provide this capability.

- A means must be provided to select from two microaddress sources. Thus, a **tri-state** bus is used. Since only one source may be actively connected to this bus at any time, each source requires an enabling signal to allow it to be selectively enabled and disabled.

Application of Tristate Gates



The Tristate Gate Symbol



Only Bit 0 is illustrated, all other bits would be attached similarly.

PROGRAM FLOW CONTROL (Cont'd)

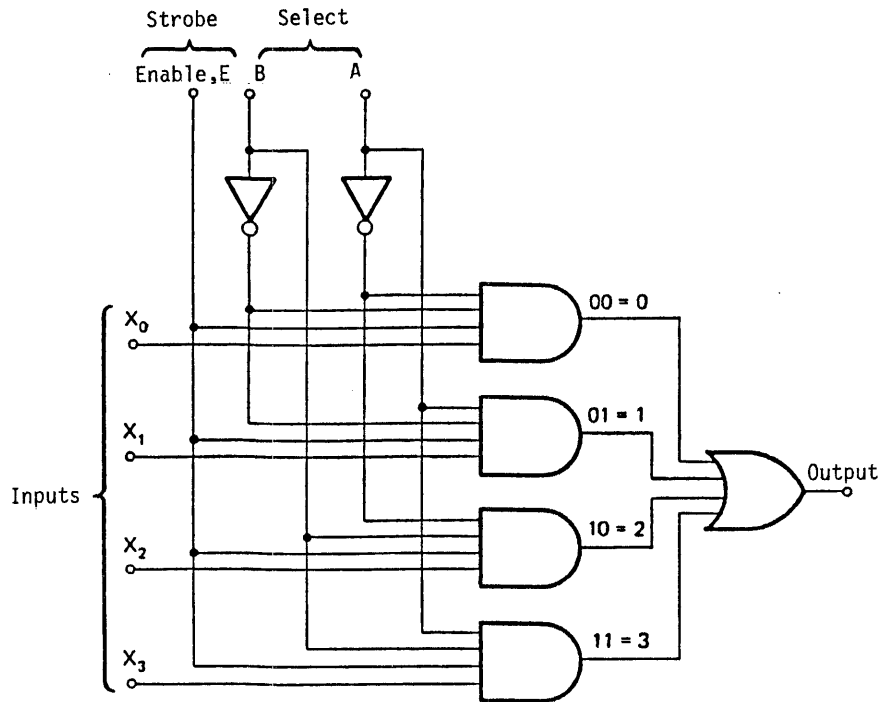
- A "load counter" signal allows the counter to be loaded from one of these tri-state sources or to simply be incremented.

- The block labeled "logic" decodes a 2-bit value from the microword "next address select field" to generate these three control signals. An alternate approach would be to provide three separate bits in the microword for these three signals.

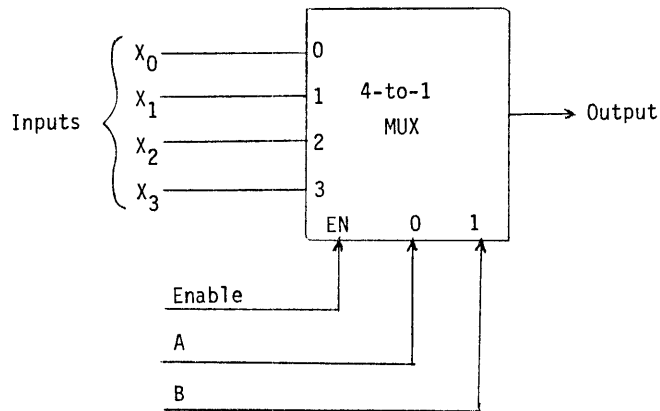
- The **multiplexer (MUX)** and polarity circuits provide the test signals for conditional jumps, and will be developed in more detail later. Likewise, the instruction register (IR) and its associated mapping PROM, which allow the introduction of new micro-addresses, will be developed later.

- Note that in formatting the microword, the microinstruction next microaddress sequence fields are grouped to the left, as previously suggested, in order to provide more structure and readability in the code. Grouping in any manner provides for understandability.

MULTIPLEXER



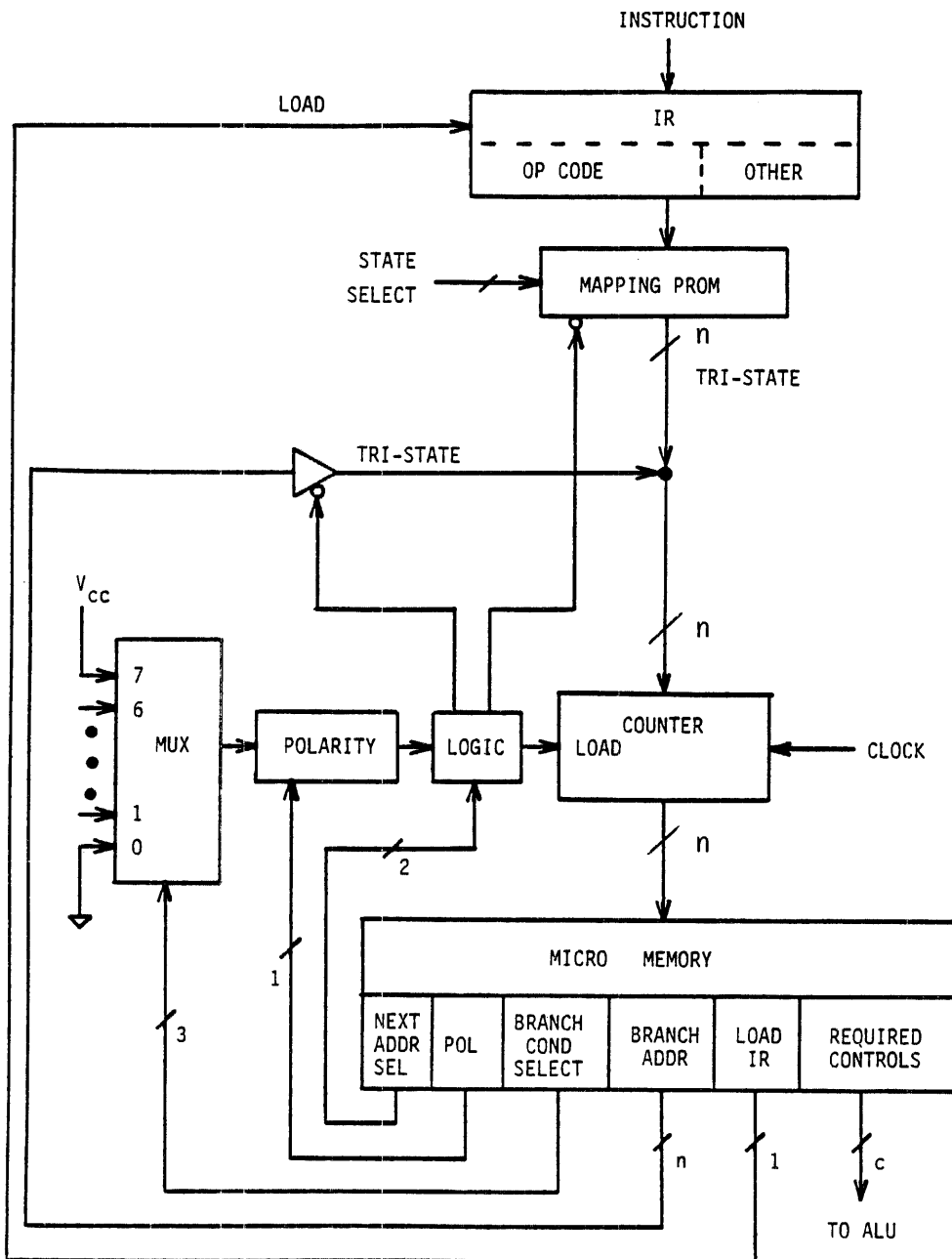
Logic Circuit



Block Symbol

General Computer Control Unit (CCU) Architecture

Each block will now be discussed in terms of its operation associated with sequencing microinstructions.



LINEAR SEQUENCES ("CONTINUE" microinstruction)

- In programming, quite often one instruction follows another. This is true of microprogramming as well. In the CCU, this is facilitated by using a counter register instead of the general register as previously shown. This counter contains the address of the current microinstruction, and can be incremented to the microaddress of a sequential flow is desired.
- The "next address select" field would contain the necessary bit pattern to disable the counter load control, allowing the counter to increment on the next clock pulse. Since three control signals must be generated, two bits would be needed for this encoded field. Whatever the actual bit pattern, the mnemonic "CONT" is assigned for a "continue" microinstruction. The other fields of the microword are not used in this mode, and are mnemonically represented as "X" for "don't care".

For example:

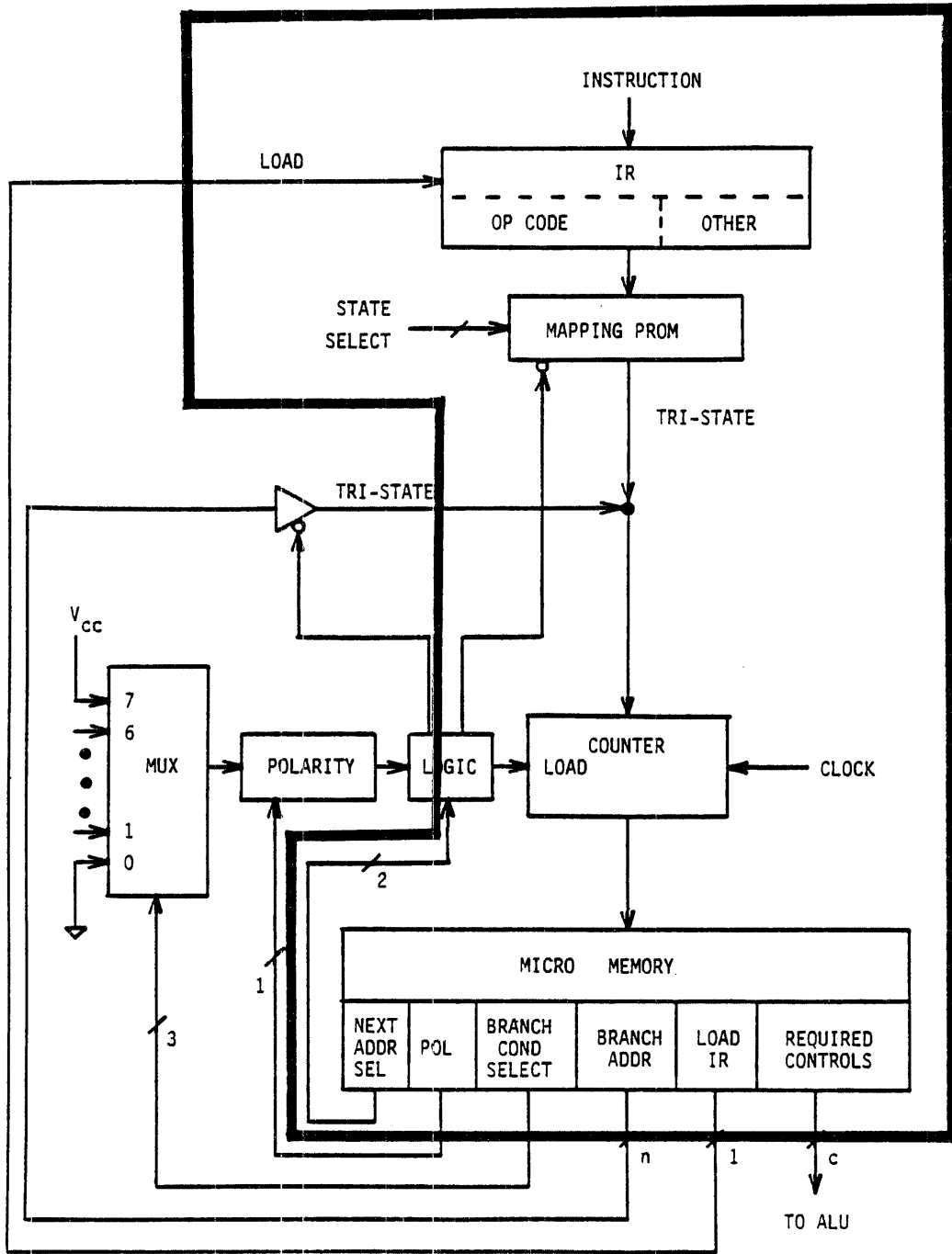
MICROINSTRUCTION SEQUENCER MICROOPERATIONS					
FLOW	NEXT ADDR	POL	COND SEL	BRANCH ADDR	CONTROL
0	CONT	X	XXX	XXXX	* * *
0	CONT	X	XXX	XXXX	* * *
0	CONT	X	XXX	XXXX	* * *

MULTIPLE SEQUENCES (JMAP)

- The controller can still execute only one sequence with the mnemonic "CONT".
- In order to execute multiple sequences, the ability to exit the current sequence is required and a new starting address from some storage location must be provided, i.e. a jump (conditional or unconditional) capability.
- The input to the counter can be used for this purpose (a jump address). Various sources are examined as sources for this address.
- First consider the interpretation of a new macro level instruction. Once the counter is loaded with a new microroutine starting address, each microinstruction in this microroutine sequence could have a "CONT" in the next address select field, except possibly for the last one.

The microinstruction would also contain one bit fields to

- enable the counter load control for external data (address) input
 - enable the tri-state output signal of the mapping PROM which is driven from the macroinstruction register (op-code field).
- The mnemonic "**JMAP**" is used to represent this "jump via the mapping PROM".



MICROMEMORY ADDRESS SOURCE

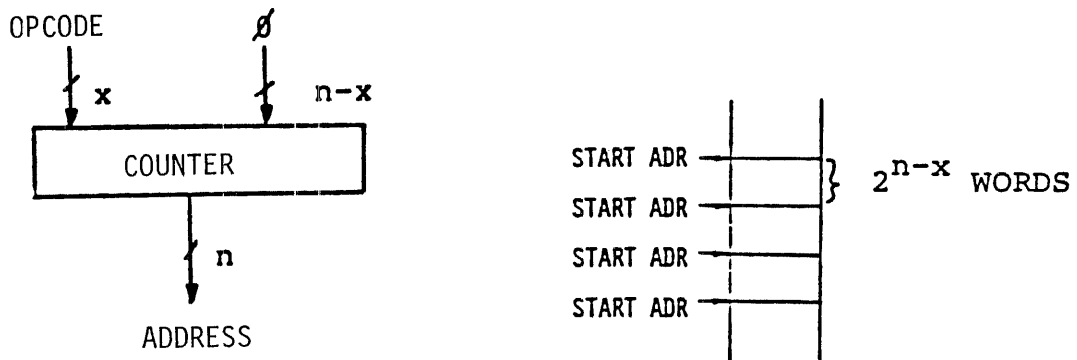
- Consider now the new microaddress source for the counter in more detail.
- In a digital computer, the starting micro-address is dependent upon the current machine (macro) instruction.
- In a controller with no macro level instructions, the starting micro-address is dependent upon the current external "command" which must supply a micro-address.
- The computer control unit (CCU) is used as an example, but the design approach is common to both. The CCU accepts either a control command or a machine instruction (OP code) as directly or indirectly defining a macro-address which lends to a sequence of microinstructions.
- Thus, to be able to control which microroutine is to be executed based upon a macro instruction
 - Add a macroinstruction register (IR)
 - Add the IR "load control" bit to the microword format
 - Gate the opcode portion of the macroinstruction to the counter as the starting address.

Note: A PROM mapper is not used in this simple case. Thus the number of opcode bits cannot exceed the microprogram address width. If it equals the microaddress width, there can only be one microword per macroinstruction (assuming unique opcodes).

OP-CODE MAPPING PROBLEM - There are typically fewer bits in the opcode than in the microaddress for example, let there be x bit opcode and n bit counter where $x < n$

SOLUTION

One approach is to input \emptyset on the remaining least significant microaddress lines:



EXAMPLE

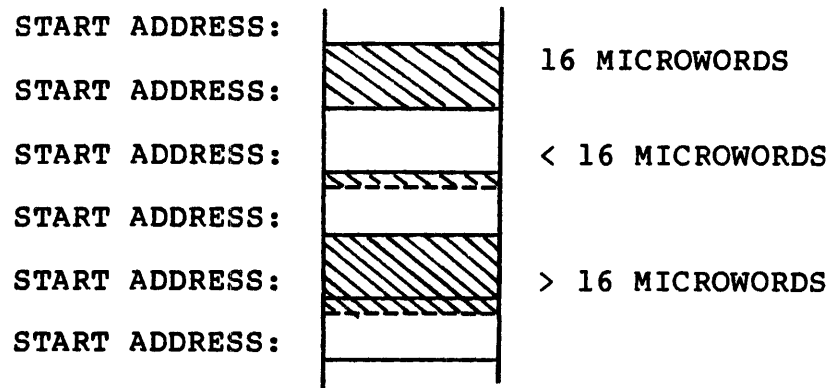
$$x = 8$$

$$n = 12$$

$$n-x = 4$$

This permits 16 microwords ($2^4=16$) per sequence or microroutine.

Examine the micromemory:

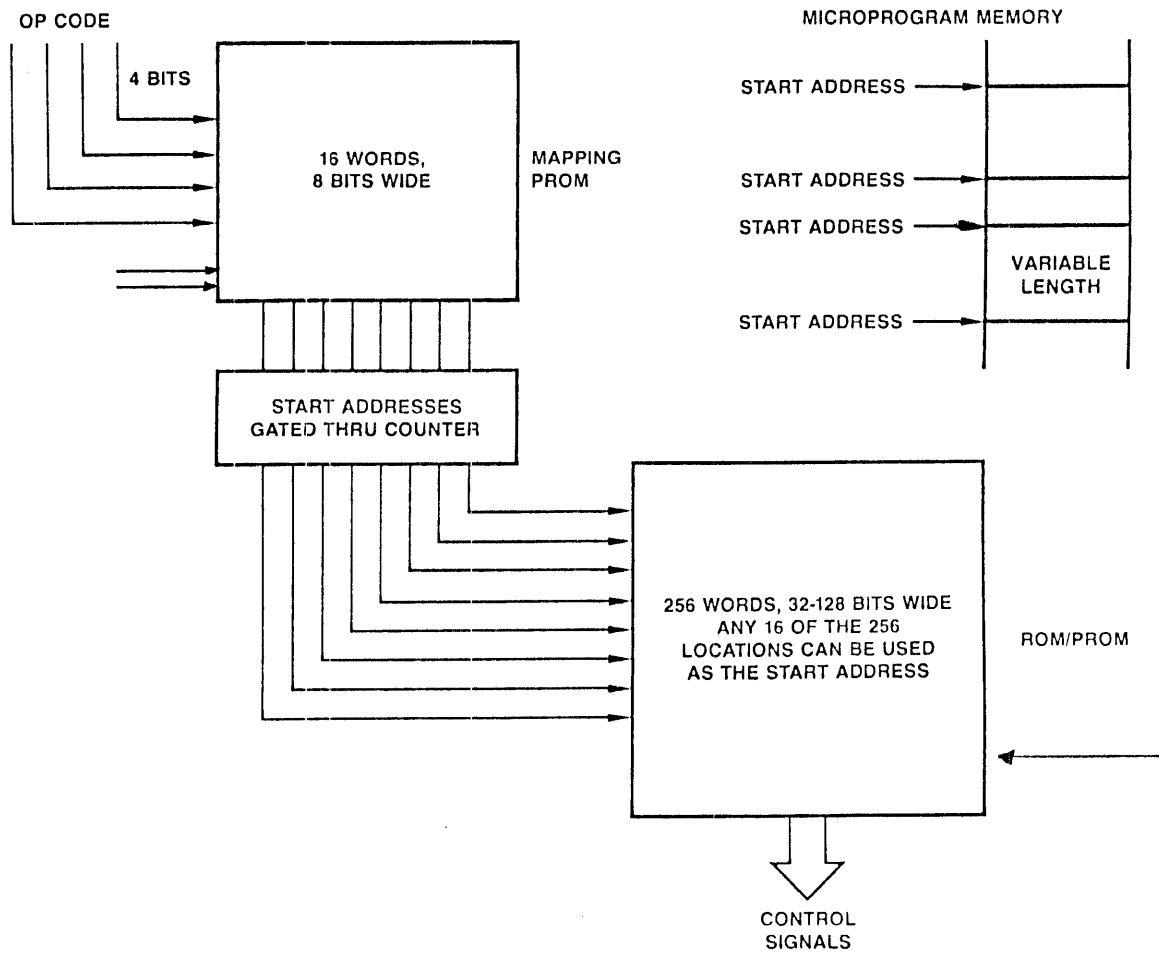


PROBLEM

- What about microroutines of less than 16 microwords?
 - Fragmented control memory
- What about microroutines of more than 16 microwords?
 - Lose starting address and its associated macro OP code

SOLUTION

- Add a micromemory address decoded (mapper)



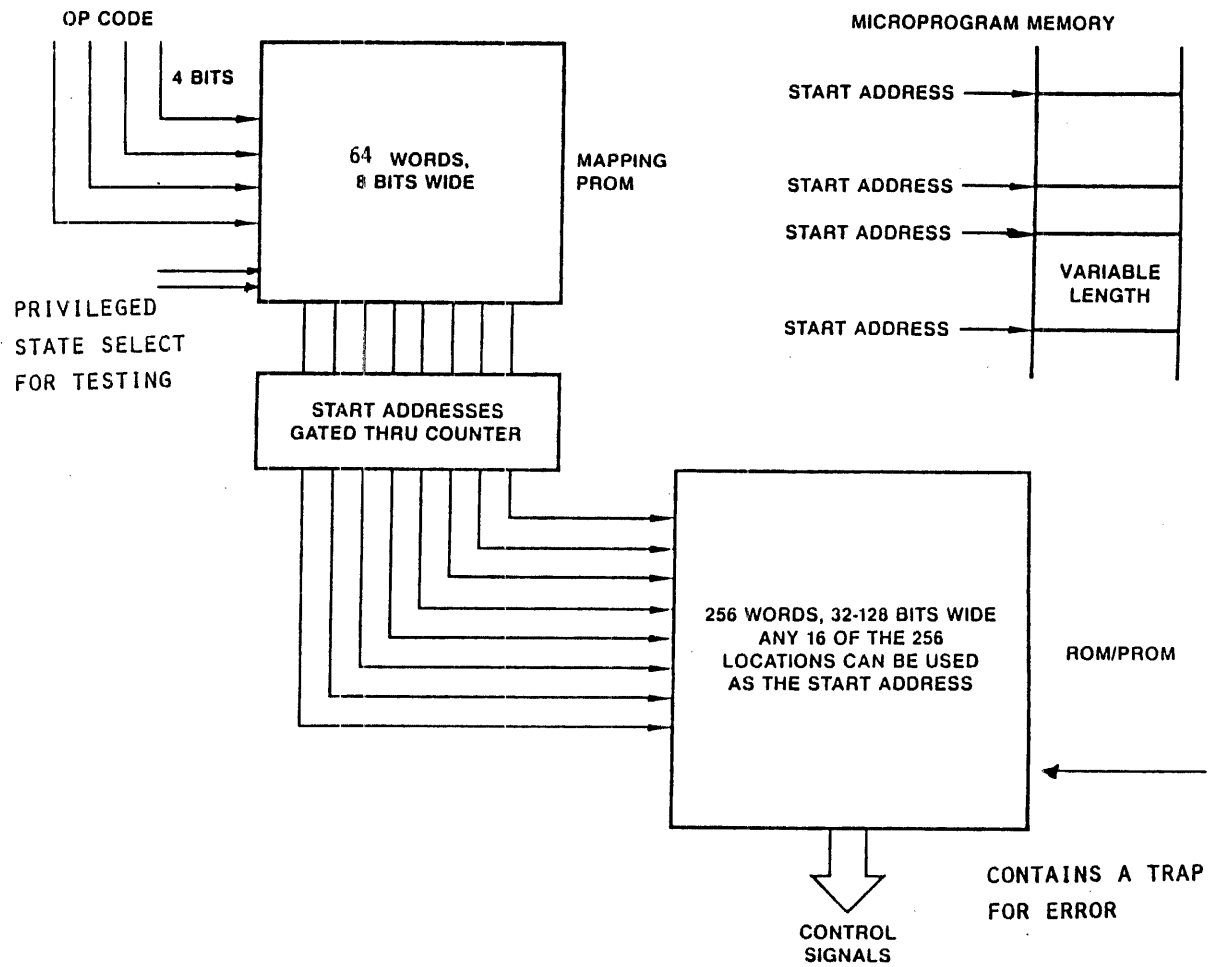
FURTHER SUGGESTIONS:

- Use a larger mapping PROM to provide for privileged macro instruction operation or detection by adding address lines driven by the console switches or the PSW (processor status word -- usually ACC value plus ALU status bits).

- Privileged instructions without the privileged bit set, map into a common "trap" microroutine.

- Provide for more addressing capability than is needed in the initial design.

- Provide for expansion in either of these directions in the initial design.



MICROPROGRAM CONTROL REVISITED

- Structuring of the microprogram can be accomplished with the same conceptual program structures which exist for high level languages. A more extensive list based upon sequence, branch and iteration is:

CONT (sequence)

GO-TO (unconditional branch or jump)

IF-THEN-ELSE (conditional branch)

IF-THEN (conditional branch)

DO X (iteration)

DO UNTIL P = TRUE or DO WHILE P = FALSE (iteration)

On X GO-TO (case statements/conditional branch)

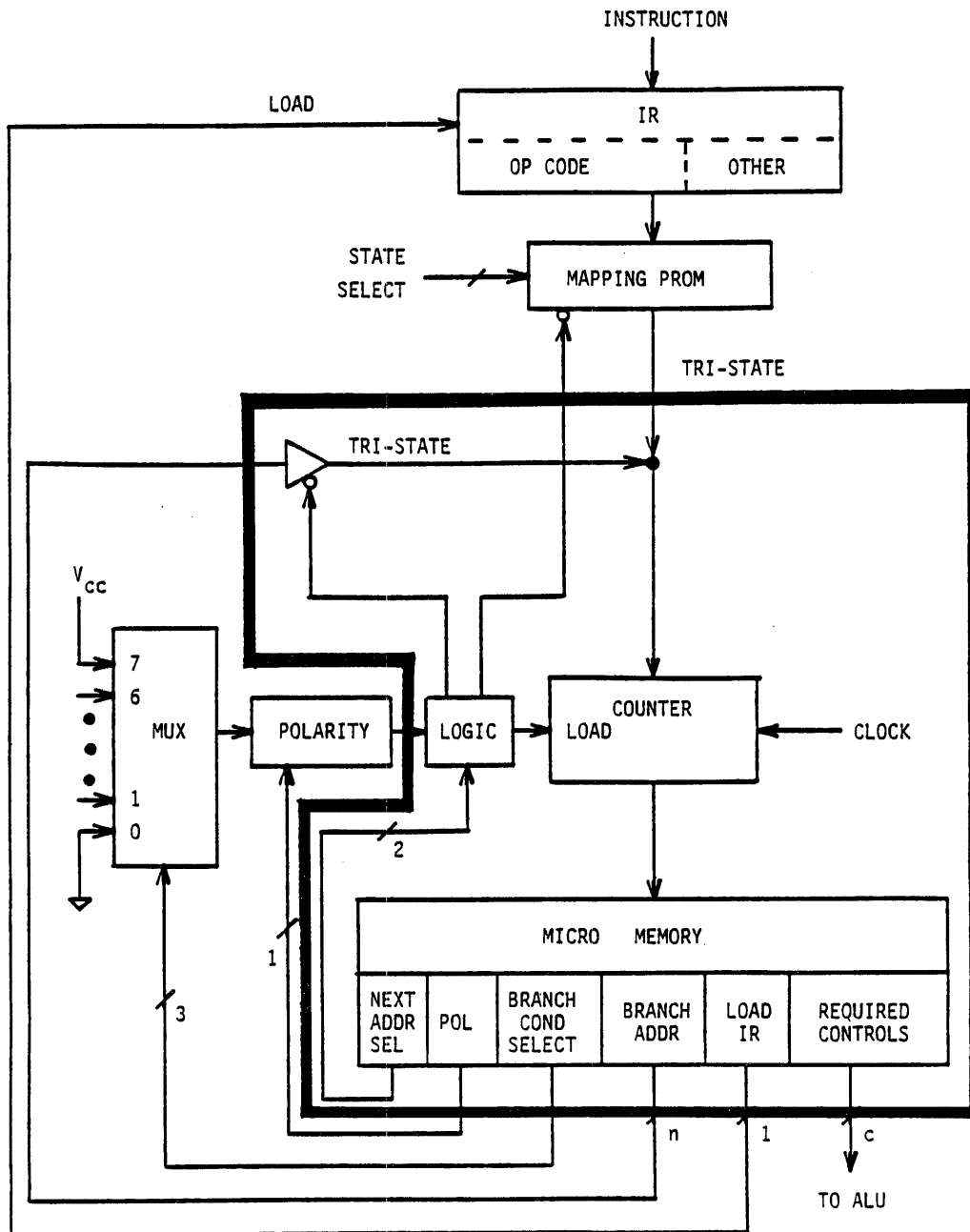
- These various control flow operations are now presented for the previous microsequencer architecture in more detail.

UNCONDITIONAL JUMP (JP)

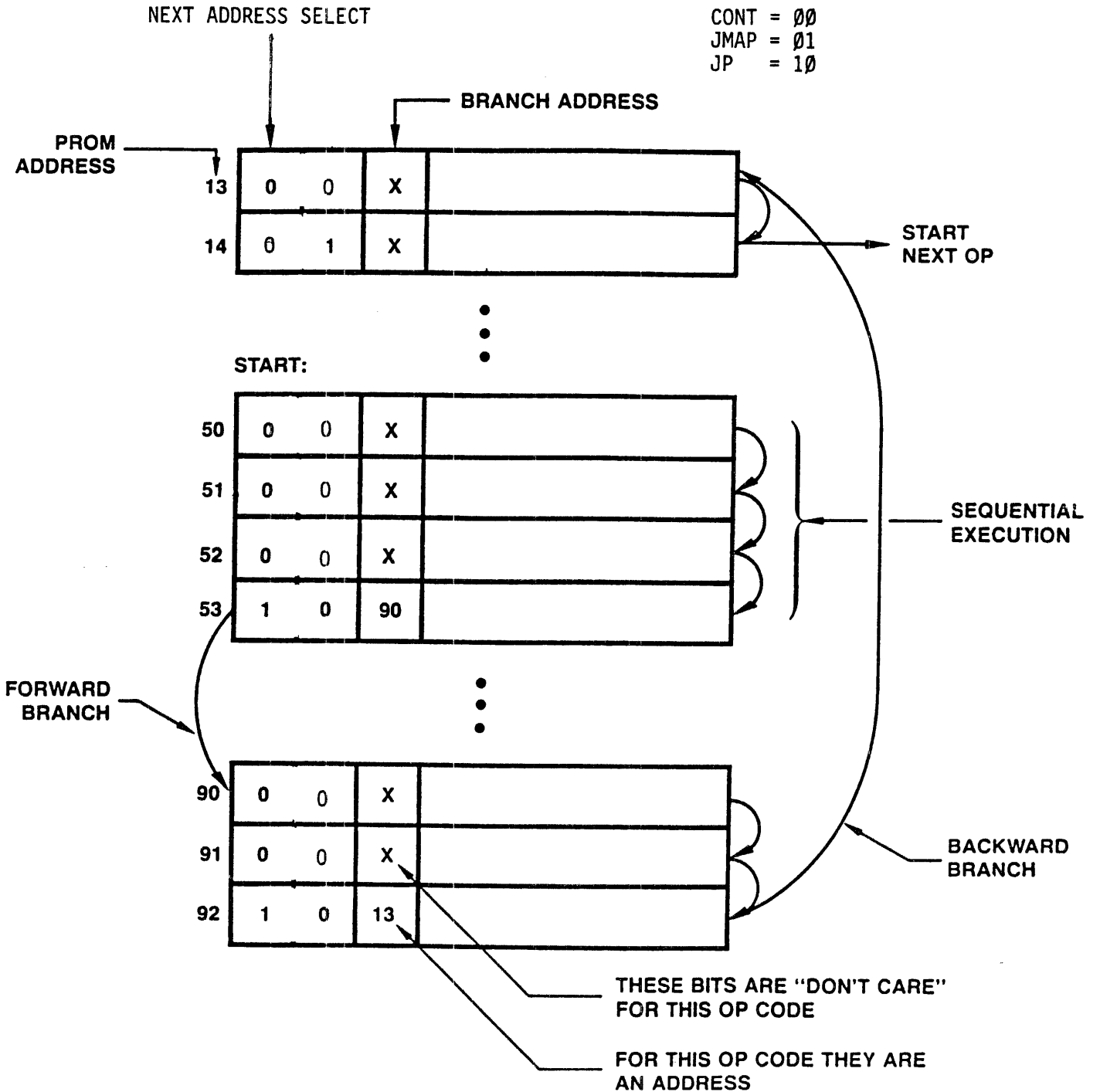
- In order to jump to another microaddress from the middle of a linear sequence, a new address is again required. The input to the counter will be used, but this time the new address will come from the current microinstruction.
- The next address select field would carry a bit pattern to
 - enable the counter load control
 - enable the tri-state gates from the microword
 branch address field
- The mnemonic "JP" is used for this next address operation

For example

FLOW (ADDR)	NEXT ADDR	POL	COND SEL	BRANCH ADDR	CONTROL
51	CONT	X	XXX	XXXX	* * *
52	CONT	X	XXX	XXXX	* * *
53	JP	X	XXX	27	* * *
90	CONT	X	XXX	XXXX	* * *
91	CONT	X	XXX	XXXX	* * *
92	JMAP	X	XXX	XXXX	* * *



EXAMPLE - JP

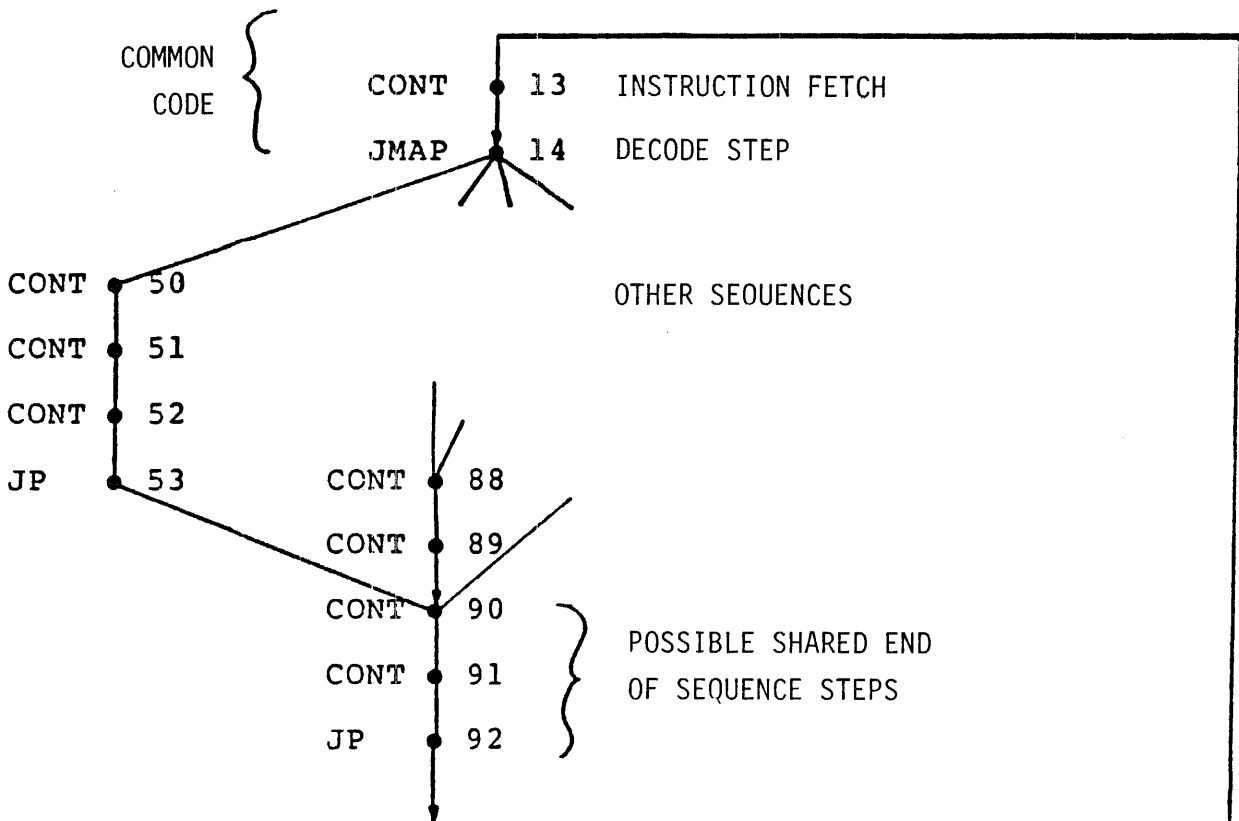


EXPLANATION:

- 50) Start address of routine
 - 50 is an address in the PROM mapping
 - Continue to 51
- 51) Continue to 52
- 52) Continue to 53
- 53) Go to 90 (jump to 90) - JP
 - The branch address is selected to be active and loaded into the counter
 - Note how both fields participate
- 90) Continue to 91
- 91) Continue to 92
- 92) Go to 13
- 13) Continue to 14
- 14) Go to next sequence start address - JMAP
 - Note that the branch address field values are don't care

MICROPROGRAM RETURN FLOW CONTROL

- In a CCU microprogram it is usually required to return to a common (shared) micro instruction sequence before jumping to the next microroutine: This is required in order to get the next macro instruction from main memory, thus the following steps are required:
 - microaddress 13 might be the macro instruction fetch step
 - microaddress 14 would be the op-code decode step to control a microaddress



CONDITIONAL JUMPS OR BRANCHES:

- During execution of certain opcodes, it is often desirable to end a microroutine dependent upon the result of a logic test. For example, a check made on a hardware status line.

For example, Add two numbers and check for

- overflow error - do one microinstruction sequence
- no overflow error - do a different sequence of microinstructions

or, Add two numbers and do

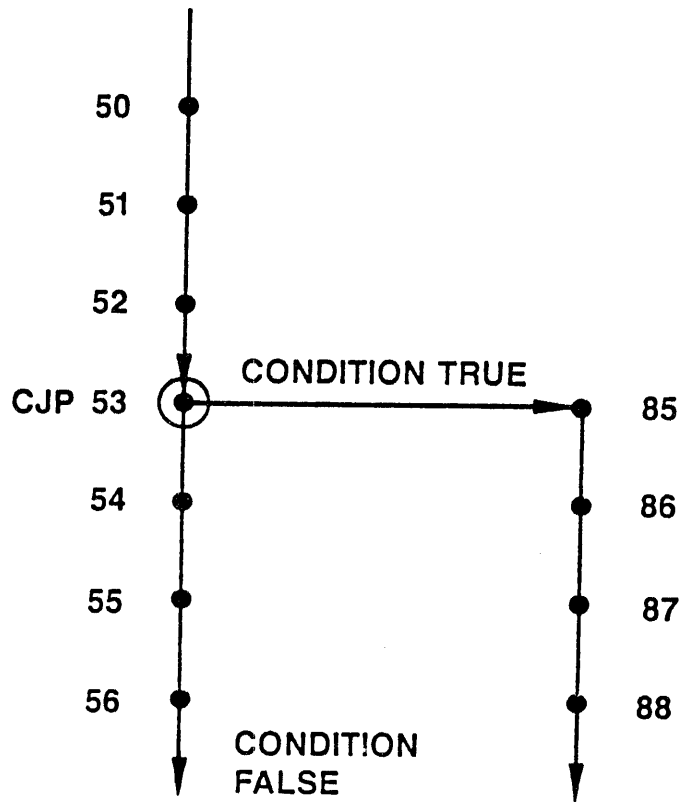
- on carry-out = 1; one microroutine
- on carry-out = 0; a different routine

OTHER TESTABLE CONDITIONS MAY INCLUDE:

<u>logical</u> <u>expression</u>		<u>mnemonic</u>
ACC = 0		ZERO
ACC > 0		SIGN
OVERFLOW		OVR
CARRY = 1		COUT
A > B		GTR
A < B		LESS
interrupt request		IR
error status bit set		ES
invalid instruction bit set		II

- A specific control flow example is shown in the figure where if the condition is true, the CJP next address selection will be microaddress 85. If the condition is false, the next microaddress is 54.

CJP (address)

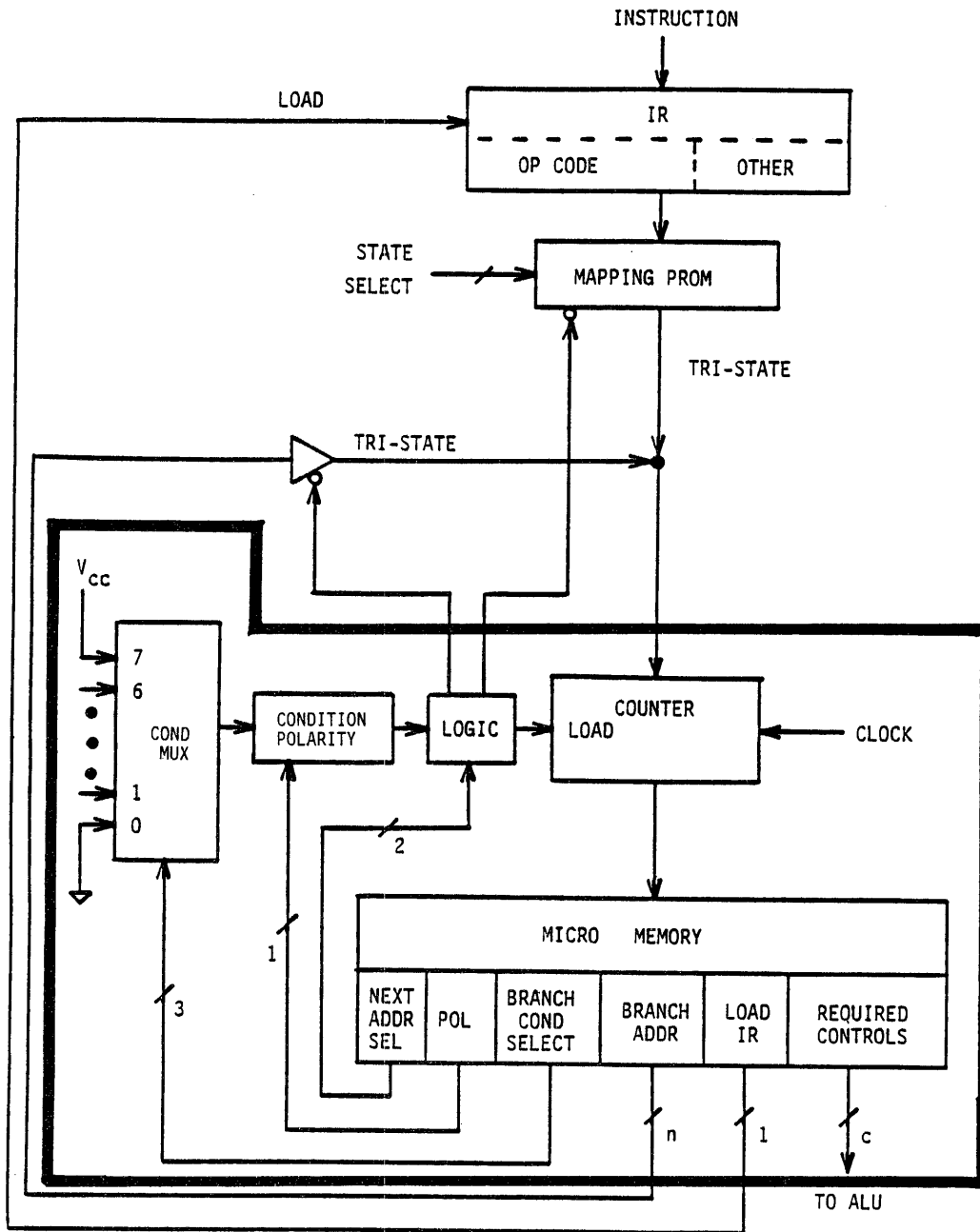


CONDITIONAL JUMP (CJP)

- In this instruction the micro-address is also provided from the microinstruction branch address field (same as JP). The next address select field code would
 - test the condition code input
 - IF the condition code is TRUE, then
 - (1) enable the counter load control
 - (2) enable the tri-state gates from the microword branch address field
 - ELSE (condition code FALSE)
 - (1) disable the counter load control

The mnemonic "CJP" is used.

- In order to allow testing one of several available conditions (overflow, negative, zero, etc.) another multiplexer is used. To allow for testing for either TRUE or FALSE conditions, a polarity selector is used. Both the choice of condition and the choice of polarity is controlled from the microinstruction.
- Note that a constant TRUE and a constant FALSE are shown as inputs to the MUX. This allows an alternate way to do unconditional jumps with a "CJP".



EXAMPLE OF CURRENT CONTROL FLOW OPERATIONS

- The following page provides a sample microroutine (sequence) which demonstrated the four microprogram control flow mnemonics
 - JMAP
 - CONT
 - JP
 - CJP

- There are three fields which are important
 - (next) address select

 - branch condition select including polarity

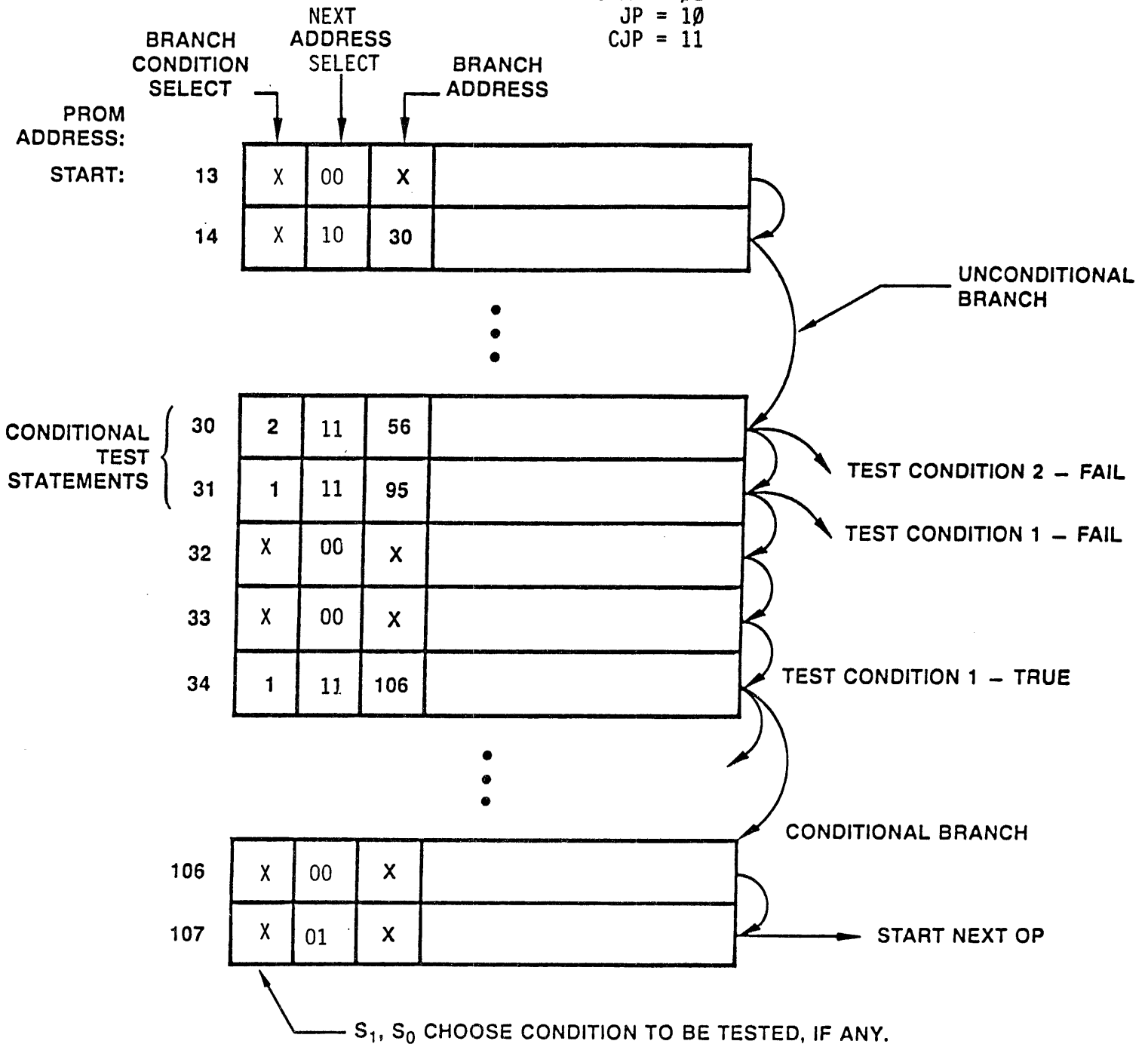
 - (micromemory) address select

 - branch (micromemory) address

- The next address select field determines the microinstruction type.

Example - CJP

CONT = 00
 JMAP = 01
 JP = 10
 CJP = 11

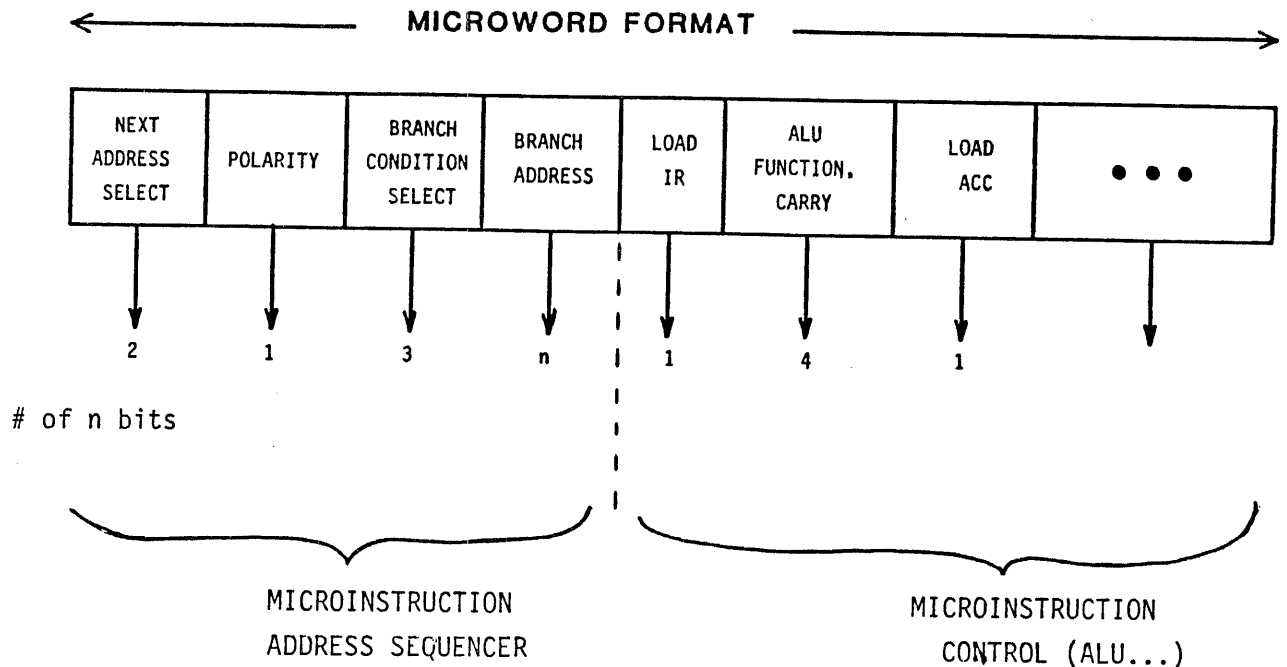


EXPLANATION:

- 13) CONT - first microaddress
- 14) JP - unconditional jump to microaddress 30
- 30) CJP - jump to microaddress 56
 if condition 2 = TRUE
 assume C_2 = FALSE
- 31) CJP on condition 1, "assume FALSE"
- 32) CONT
- 33) CONT
- 34) CJP on condition 1, "assume TRUE", GO TO
 microaddress 106
 this time C_1 = true
 GO TO 106
- 106) CONT
- 107) JMAP - unconditional jump
 select mapping PROM output

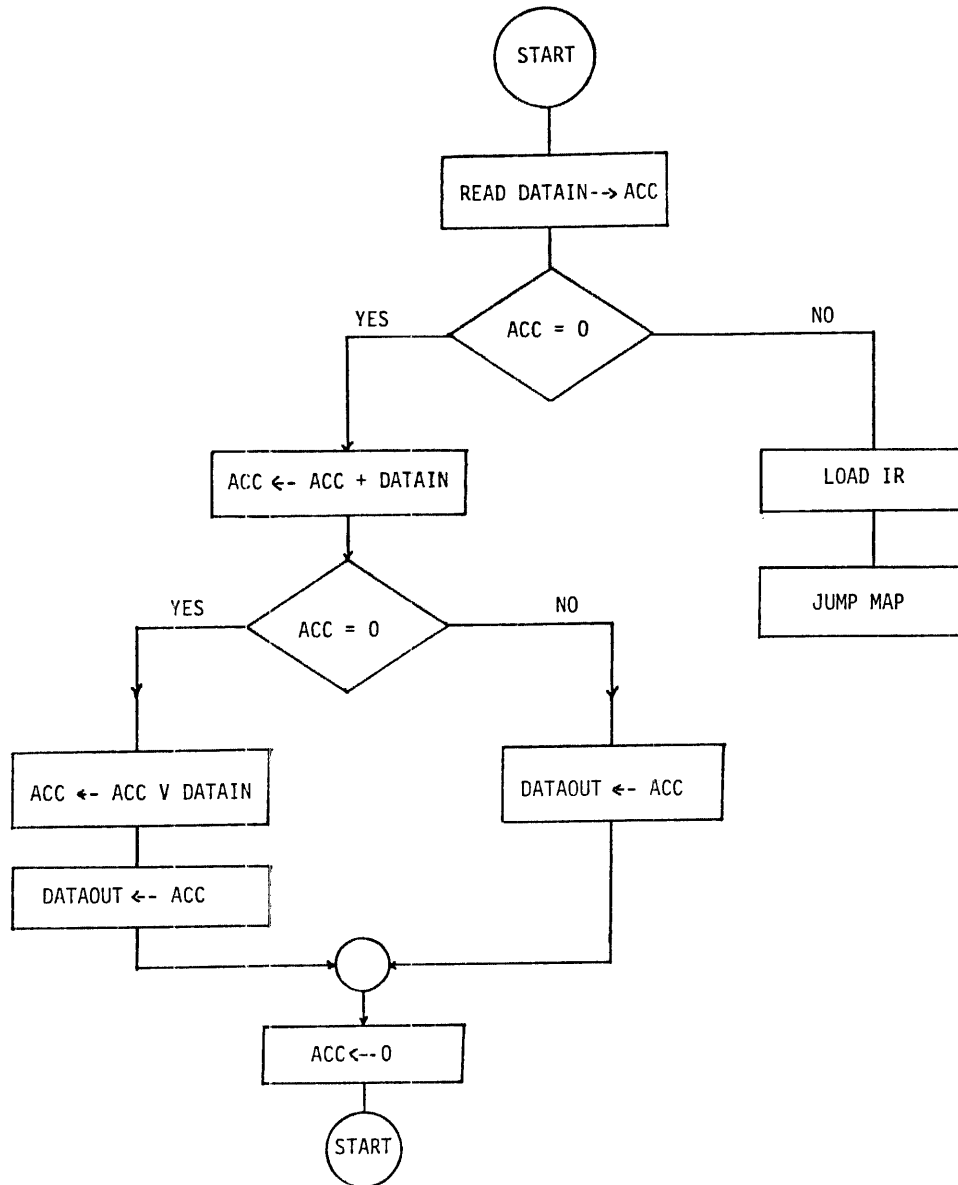
CLASS EXERCISE: MICRO-PROGRAM CONTROL

- The purpose of this exercise is to develop additional understanding of microprogramming architectures through a simple example.
- Consider the simple computer presented at the beginning of this section. With the control fields added, the microword is defined as follows:



NOTE: Another way of stating requirements is through the use of a flow chart defining specific RTL sequential operations.

- For the structured flowchart on the next page, write the microcode for the sequencer portion of the microinstructions. Define mnemonics where needed.



This flowchart does not represent a real-world algorithm, but is useful as a pedagogical example.

ENCODING OF MNEMONICS (bit patterns are arbitrary examples)

CONT = 00; continue

JMAP = 01; jump map

JP = 10; unconditional jump

CJP = 11; conditional jump

TRUE = 1; condition true

FALSE = 0; condition false

ZERO = 000; test for ALU result = 0

SOLUTION

<u>FLOW</u> <u>(MM ADDR*)</u>	<u>NEXT</u> <u>ADDR*</u>	<u>POL</u>	<u>COND</u> <u>SEL</u>	<u>BRANCH</u> <u>ADDR*</u>	<u>CONTROL</u>	<u>(REGISTER TRANSFER</u> <u>LANGUAGE)</u>
1	CJP	FALSE	ZERO	6	DATAIN --	ACC
2	CJP	FALSE	ZERO	4	ACC --	ACC + DATAIN
3	CONT	X	XXX	XXX	ACC --	ACC V DATAIN
4	CONT	X	XXX	XXX	DATAOUT --	ACC
5	JP	X	XXX	1	ACC --	0
6	JMAP	X	XXX	XXX	LOAD	IR**

* All addresses are micromemory addresses

** Assume macroinstruction prefetch

1-940

ED2900A

1-940

1-950

ED2900A

1-950

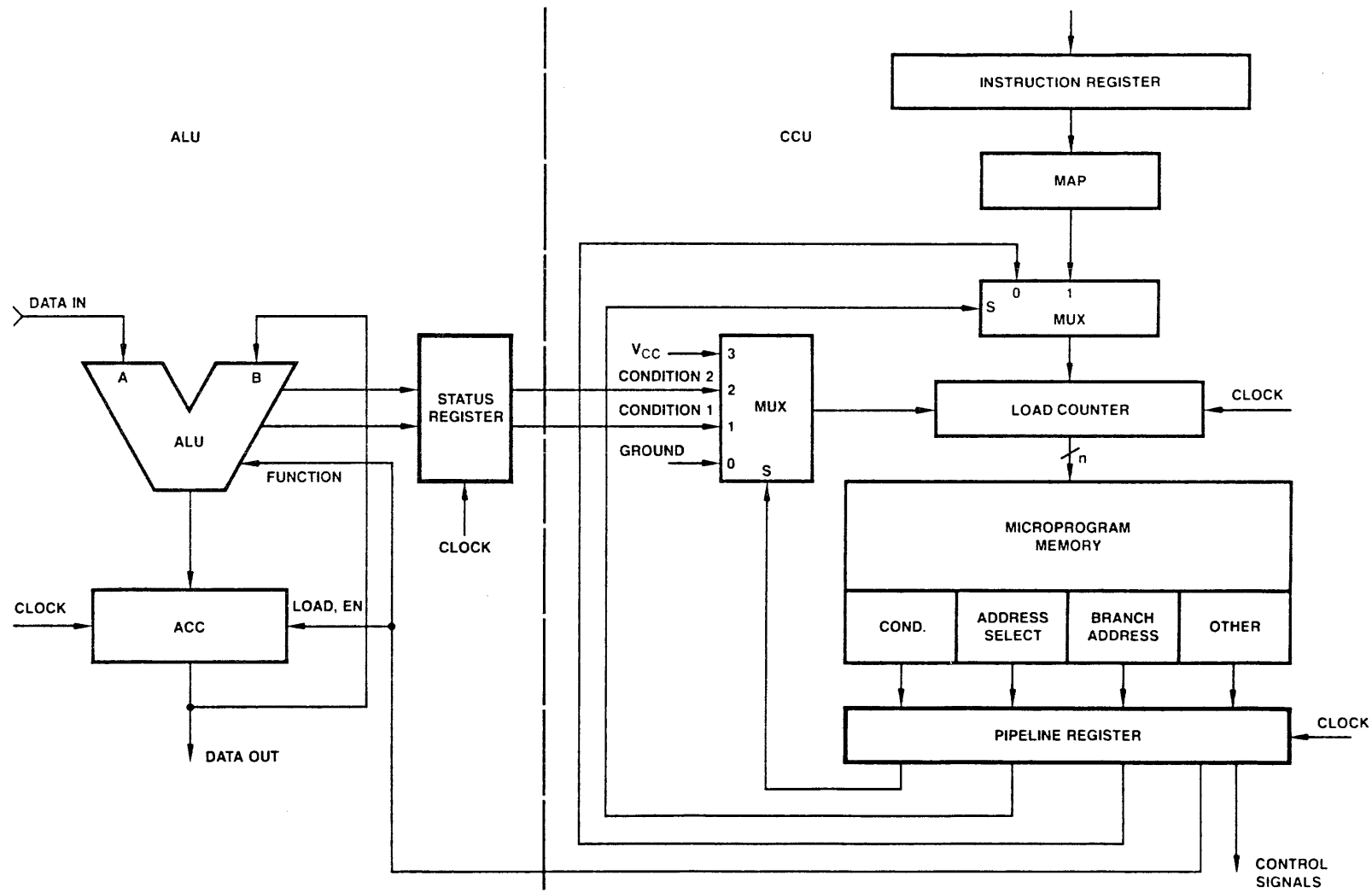
IMPROVING CCU SPEED

TIMING CONSIDERATIONS

- Consider the CCU with the ALU attached as shown in the figure on the next page. Note that the condition code MUX and address logic are combined into one block.
- Note also the addition of a **status register** between the ALU and the condition code multiplexer. This allows a test on the result of the previous operation, and increases speed as will be seen later.
- In order to determine the clock period, it is necessary to time the signal flows from the time they leave a register until they are ready to be clocked into another register. This must be done for all such paths. The slowest register-to-register path determines the lower bound on clock speed (microcycle).
- For example, the main path delays in the CCU itself are:
 - clock to output of the counter
 - read-access time of micromemory
 - set-up time for the counter (except for CONT)
 - in parallel with the above, time through the MUX and set-up time for the counter load
$$t_{CP} = t_{CL \text{ to output}} + t_{\text{read access}} + t_{\text{set-up}}$$

since $t_{\text{mux delay}} + t_{\text{setup}}$ is shorter
- In order to examine speed improvements in the CCU, consider the timing paths including the ALU.

ADD THE ALU



TIMING COMPUTATION

- The timing for this implementation is computed by examining all sequential paths. Two of these are of interest in developing our CCU:

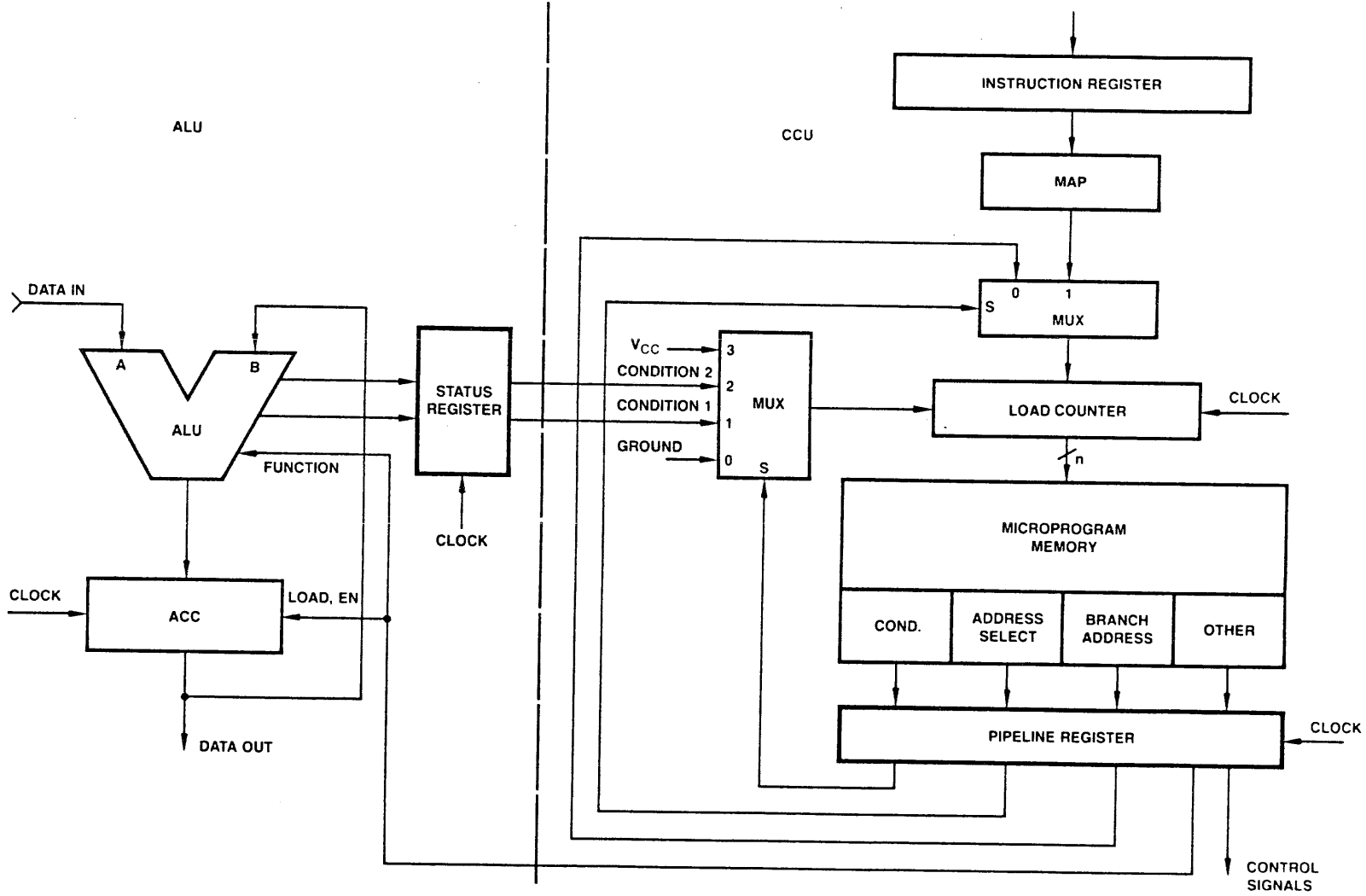
First:

1. Clock to output of counter	15ns
2. Fetch instruction	50 ns
3. ALU to status line	95 ns
4. Status register set-up	5 ns
	Total = 165ns

And second, in parallel:

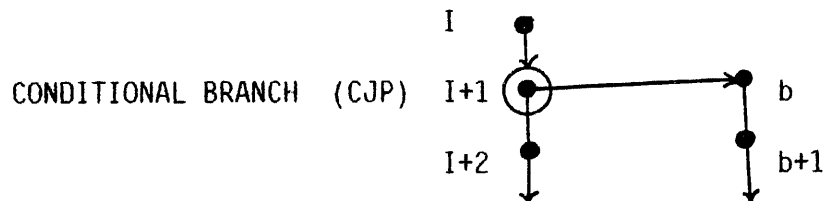
Steps 1. and 2.	65 ns
3a. ALU instr to output	120 ns
4a. ACC set-up	5 ns
	Total = 190 ns

- The minimum microcycle required is the time of the longest path $\underline{C}_p = \text{CNTR}(15) + \text{MEMORY}(50) + \text{ALU}(120) + \text{ACC}(5) = 190 \text{ ns}$



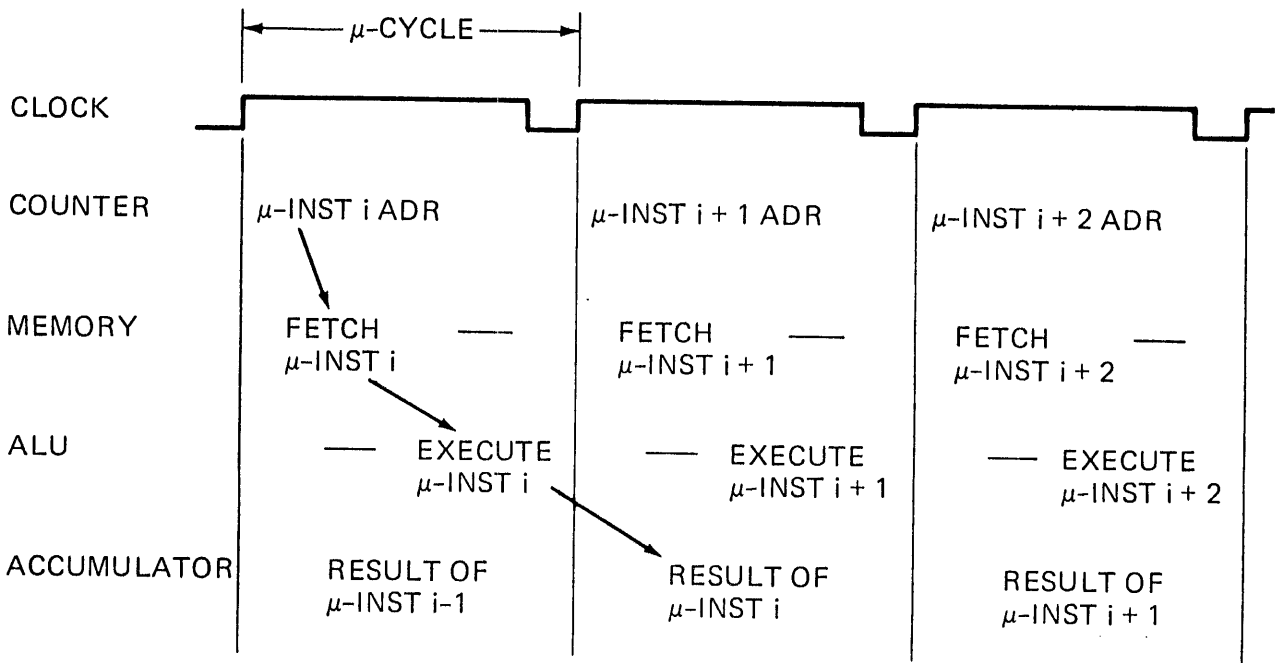
CONTINUING EVOLUTION OF SEQUENCER

- A fairly powerful sequencer has evolved in terms of the instruction set (next address selection) it can support. However, speed is another criteria. Some additional improvements can be made to increase speed of operation.
- For this development, the execution of a conditional branch is analyzed, both with the branch taken and with the branch not taken.

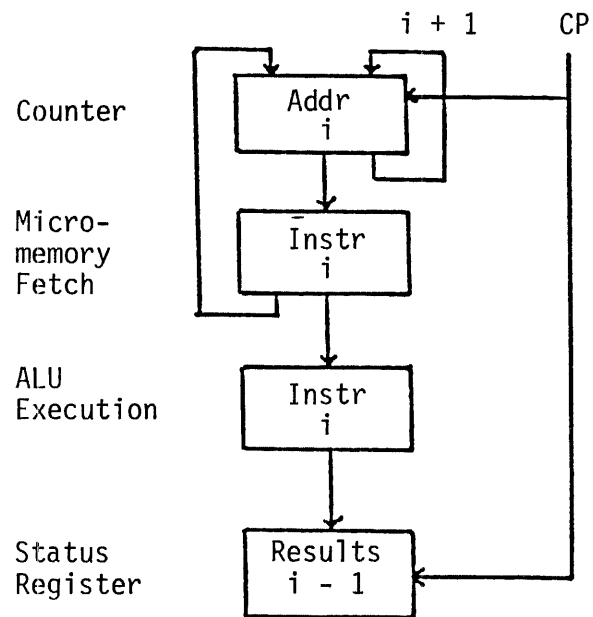


- Note that although several things seem to take place "simultaneously" during a single microcycle, some of them actually occur sequentially within a microcycle due to asynchronous nature (non-clocked logic delays) of the hardware.
- Note also that there is no difference in flow when the branch is taken as shown in the BRANCH TAKEN diagram.

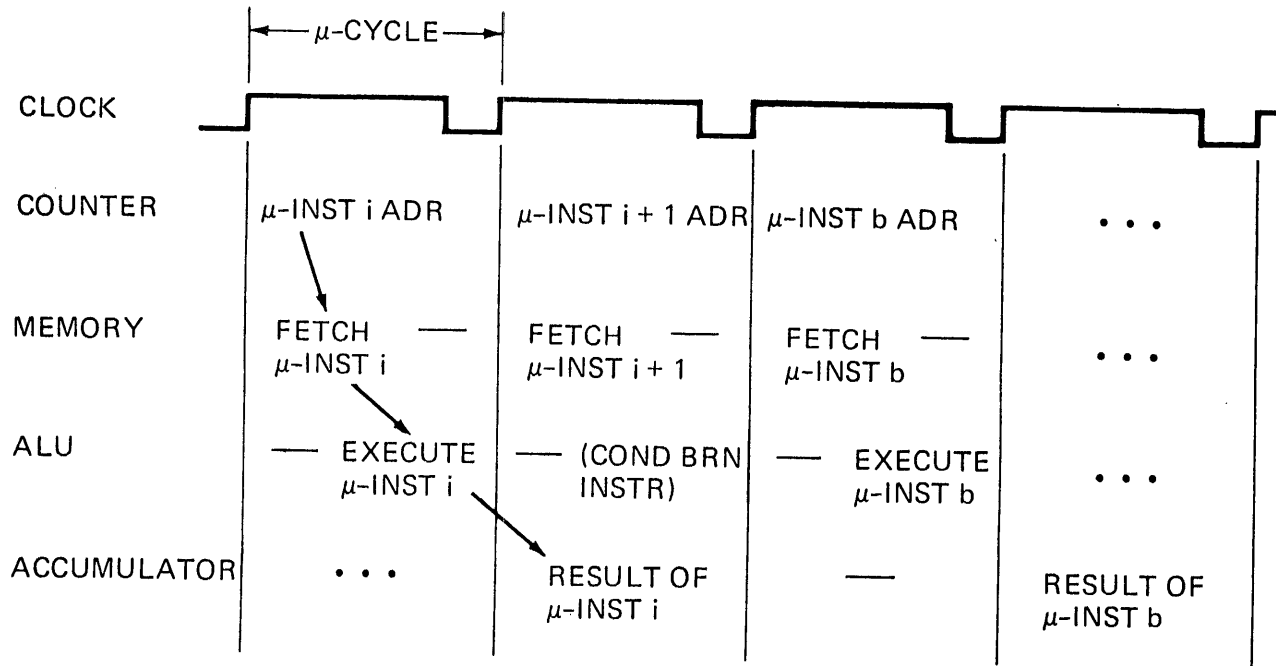
No Branch



Current Instruction Flow
(No Branch)

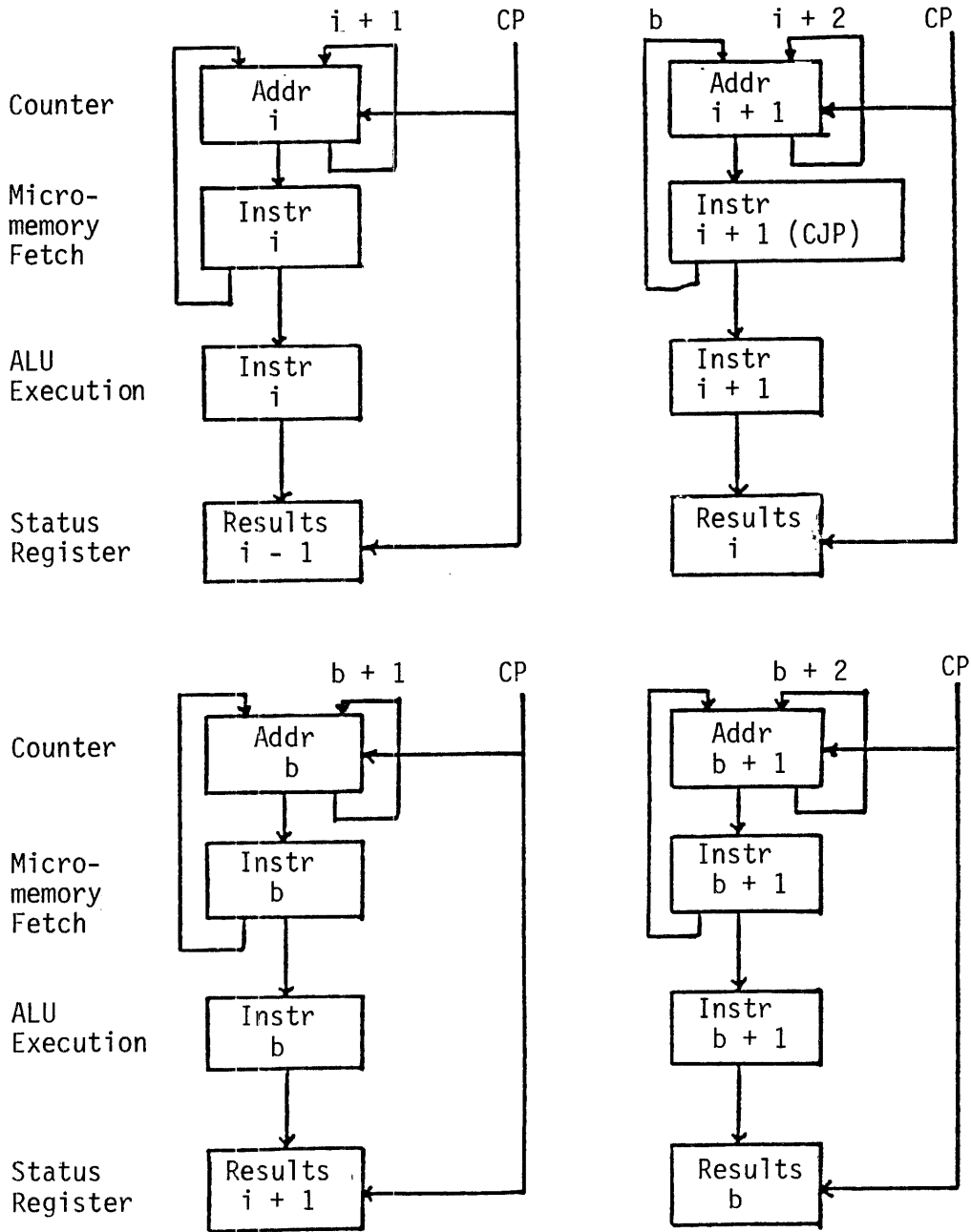


Branch Taken



Branch on result of previous instruction.

Current Architecture with Branch Taken



PROBLEM WITH NONPARALLEL USE OF FUNCTIONAL UNITS

- Memory fetch idle during ALU execute

- ALU idle during memory fetch

- Wide or long micro-cycle (relatively slow)

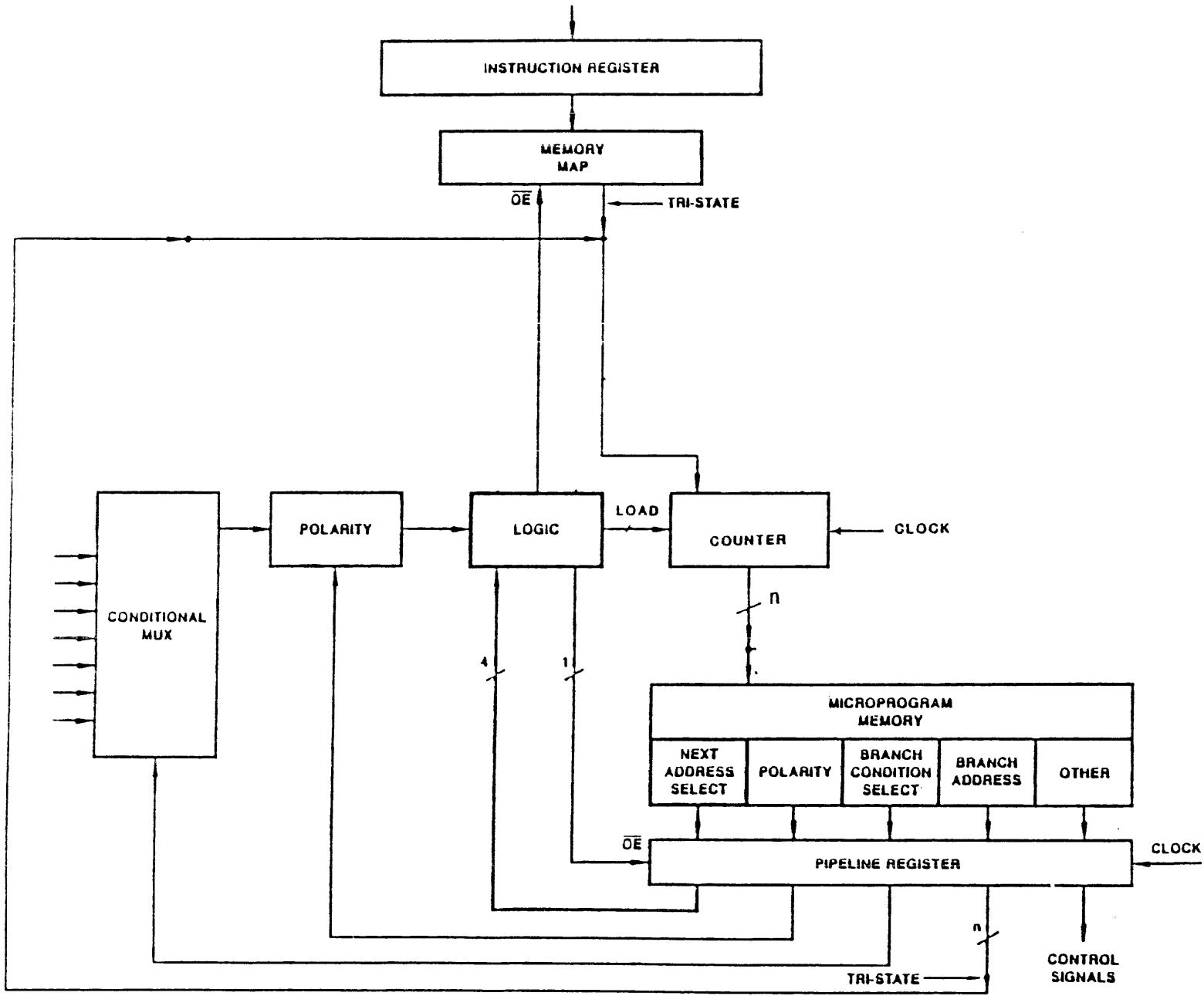
A SOLUTION

- Add a pipeline register (buffer) at the output of the ROM (PROM). The pipeline register then buffers the "flow" of data in the logic (pipe) so that independent functional units can act in parallel (concurrent operation) for reduced microcycle timing.

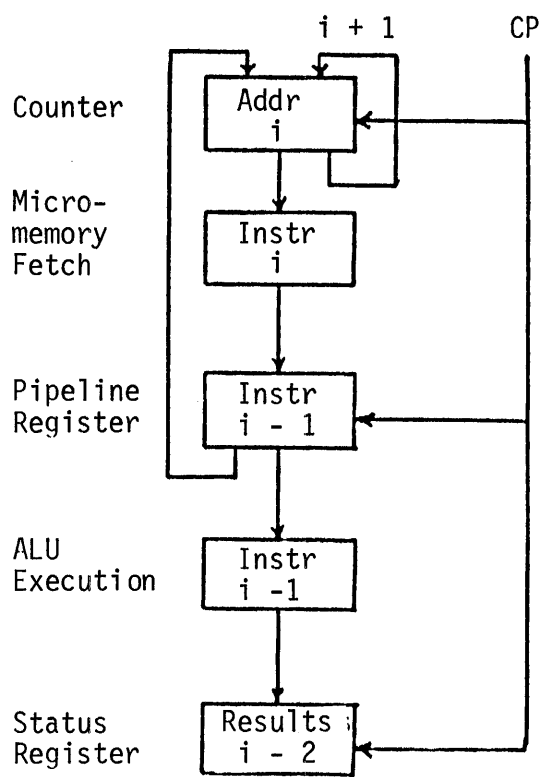
A two-level pipeline results in the current design with:

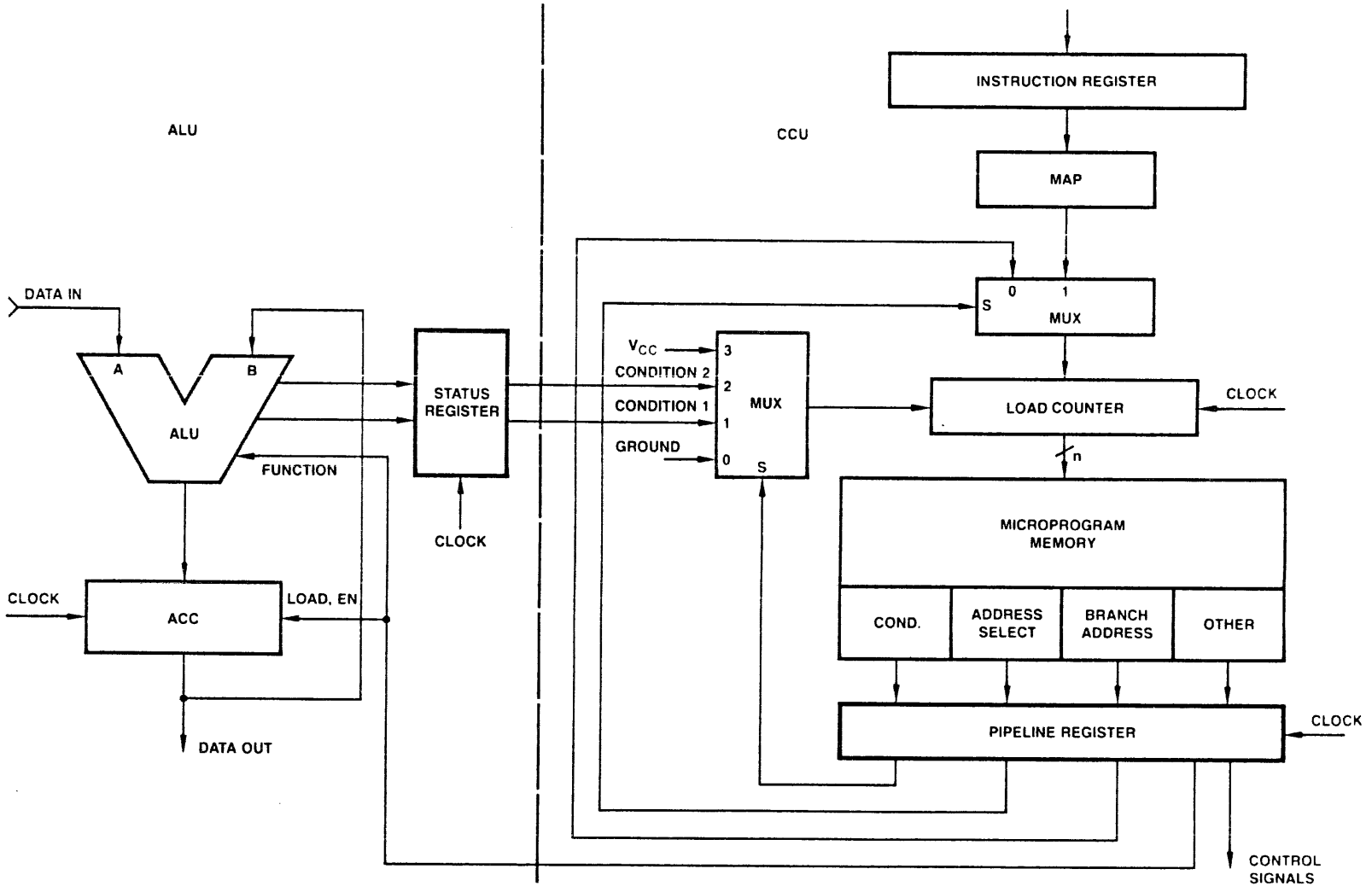
- 1) counter register

- 2) pipeline register

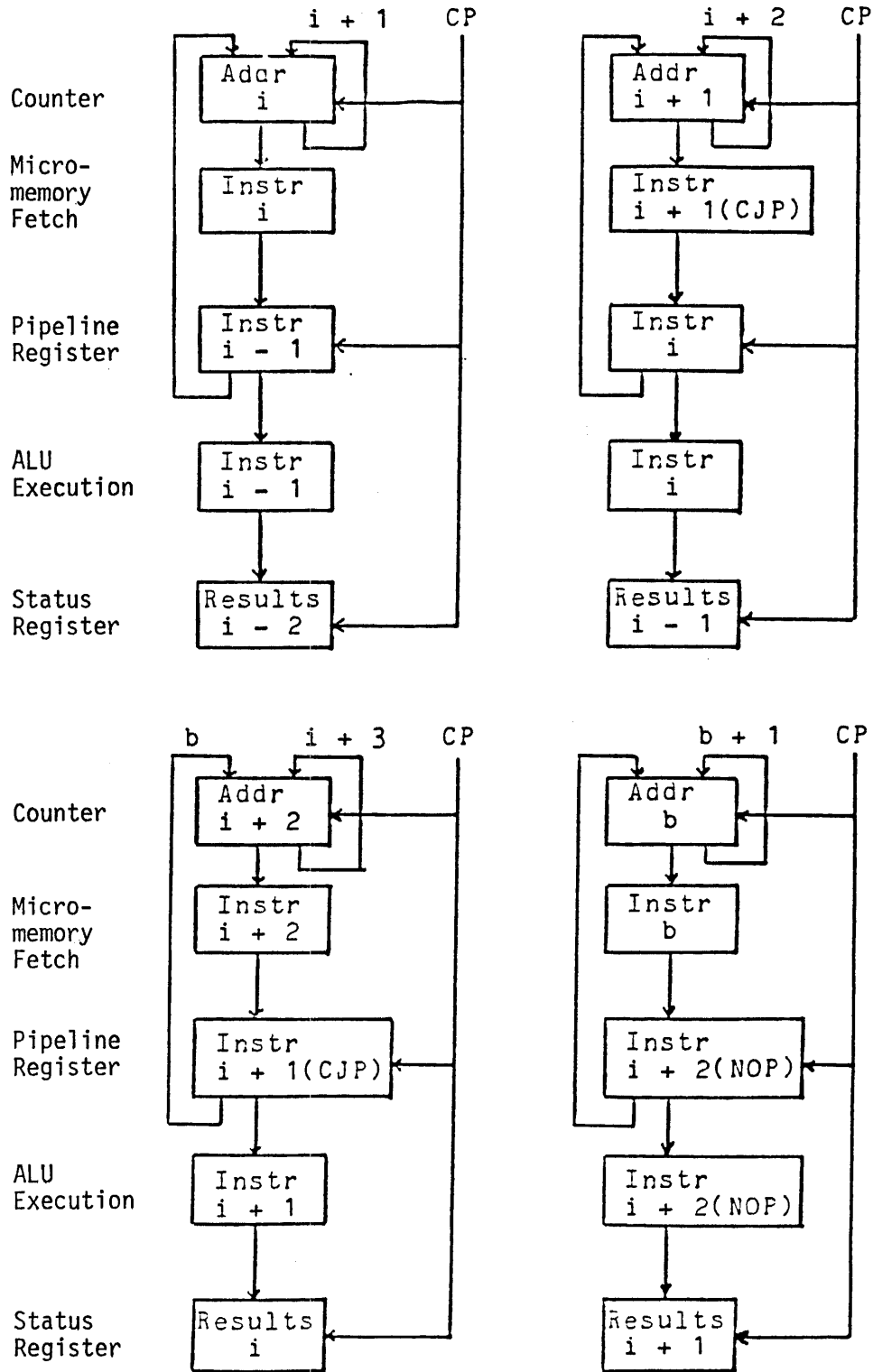


Pipeline Concept

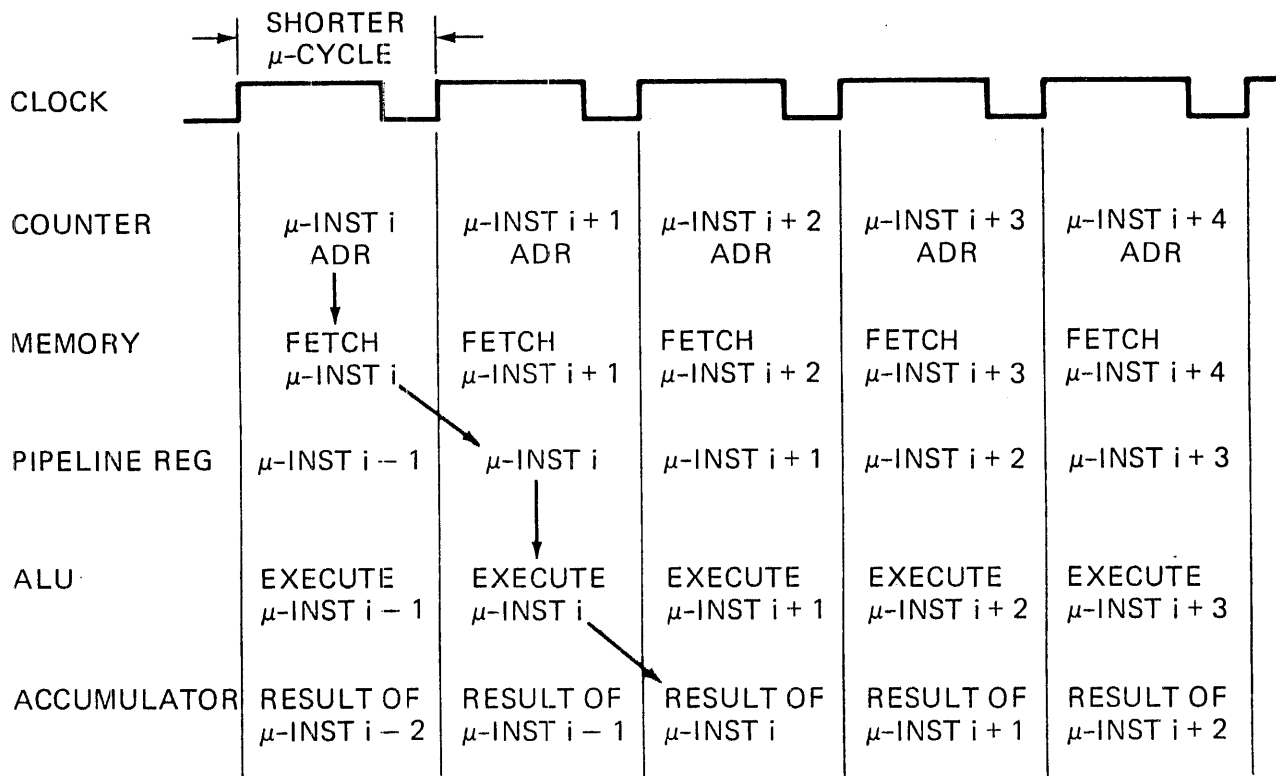




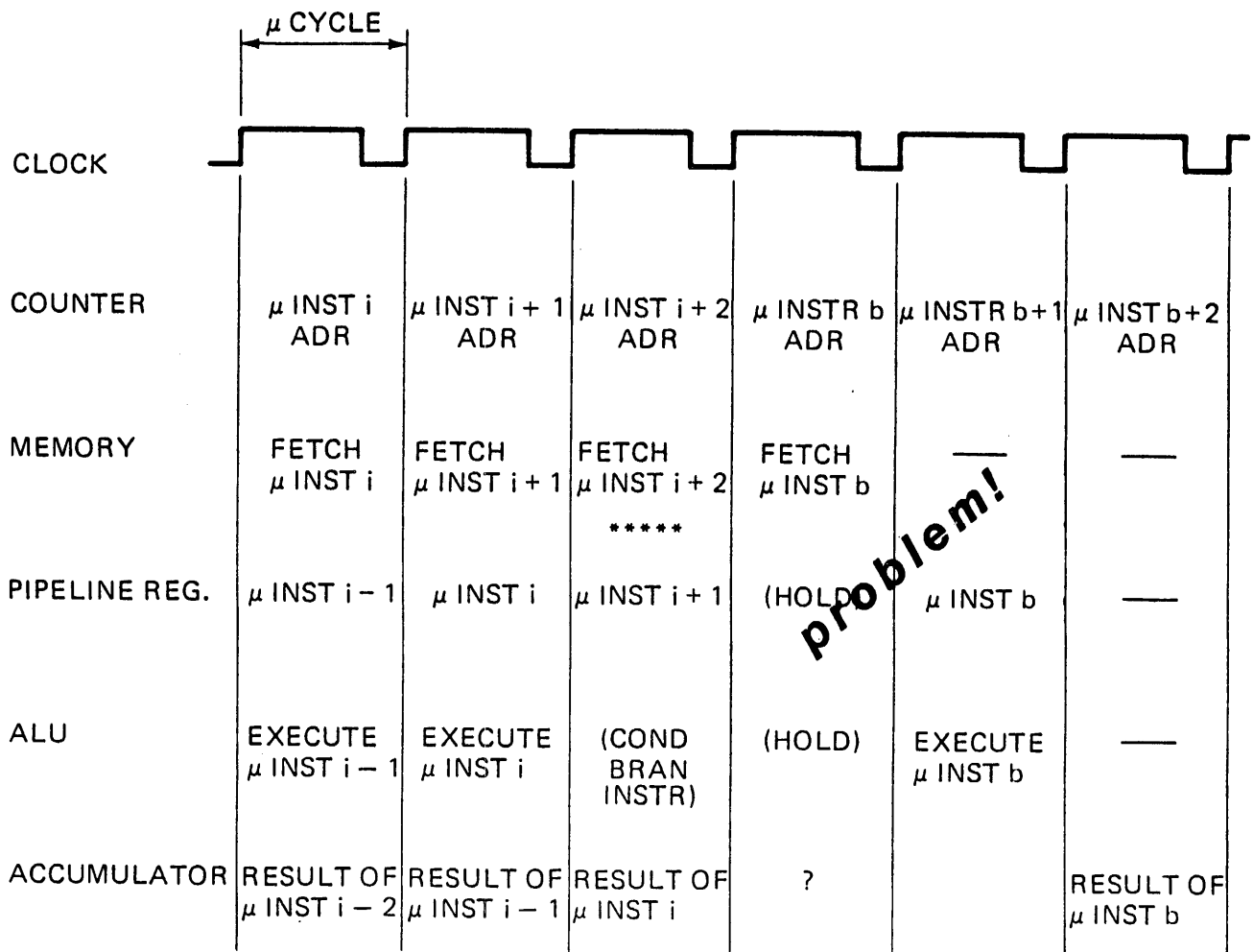
Pipeline with Branch Taken



No Branch

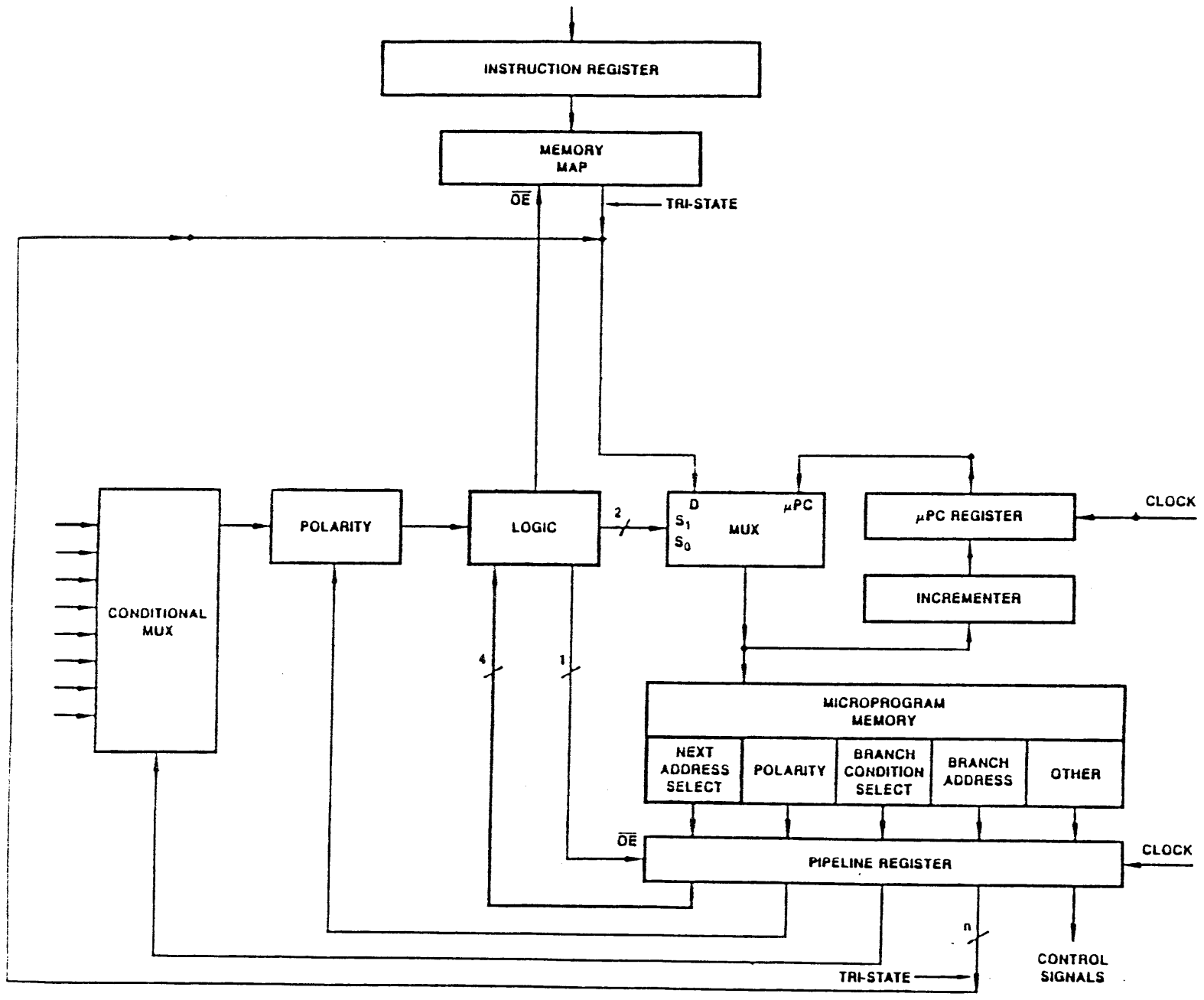


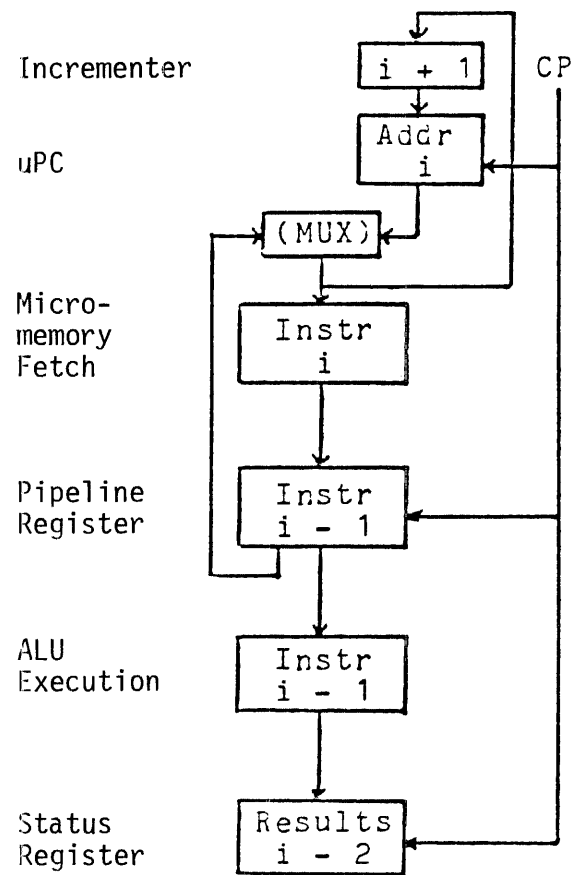
Branch Taken



ADDITIONAL ARCHITECTURAL IMPROVEMENTS

- Further improvement can be made by moving the counter out of the path of the branch address, and replacing it with a combinatorial logic **incrementer** and a **microprogram counter register (uPC)**. The incrementer generates the next sequential address during the clock cycle with only a gate delay.
- A multiplexor is added to allow either the micro PC register or the tri-state bus to be selected as the address source to the micro memory.
- Note that the tri-state output on the pipeline is for the branch address field only.
- This architectural change eliminates the problem of a lost cycle when the branch is taken and allows the controller to run at full speed all the time as shown in the following diagrams:



Pipeline Concept with Incrementer

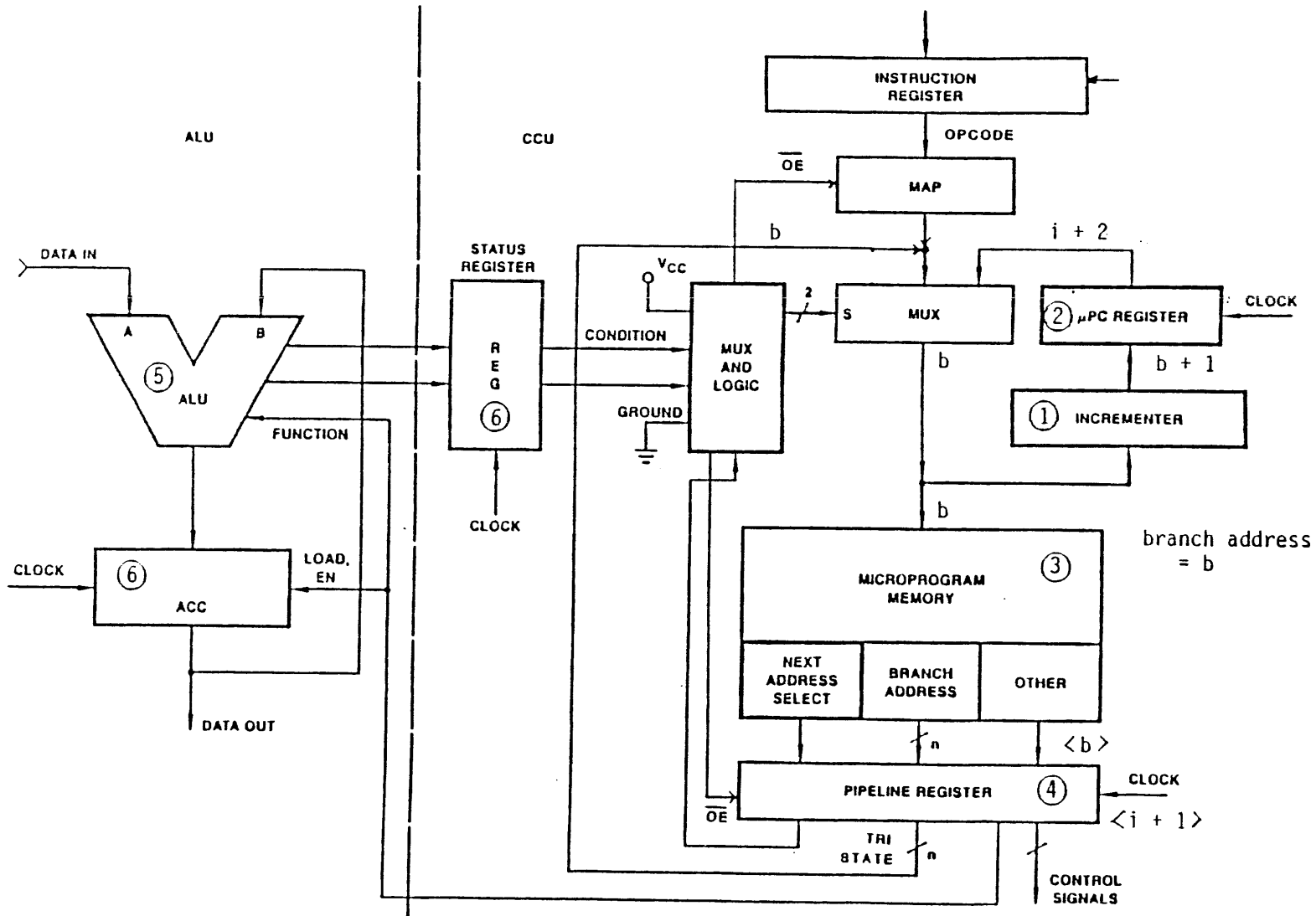
(revised)

1-1150

ED2900A

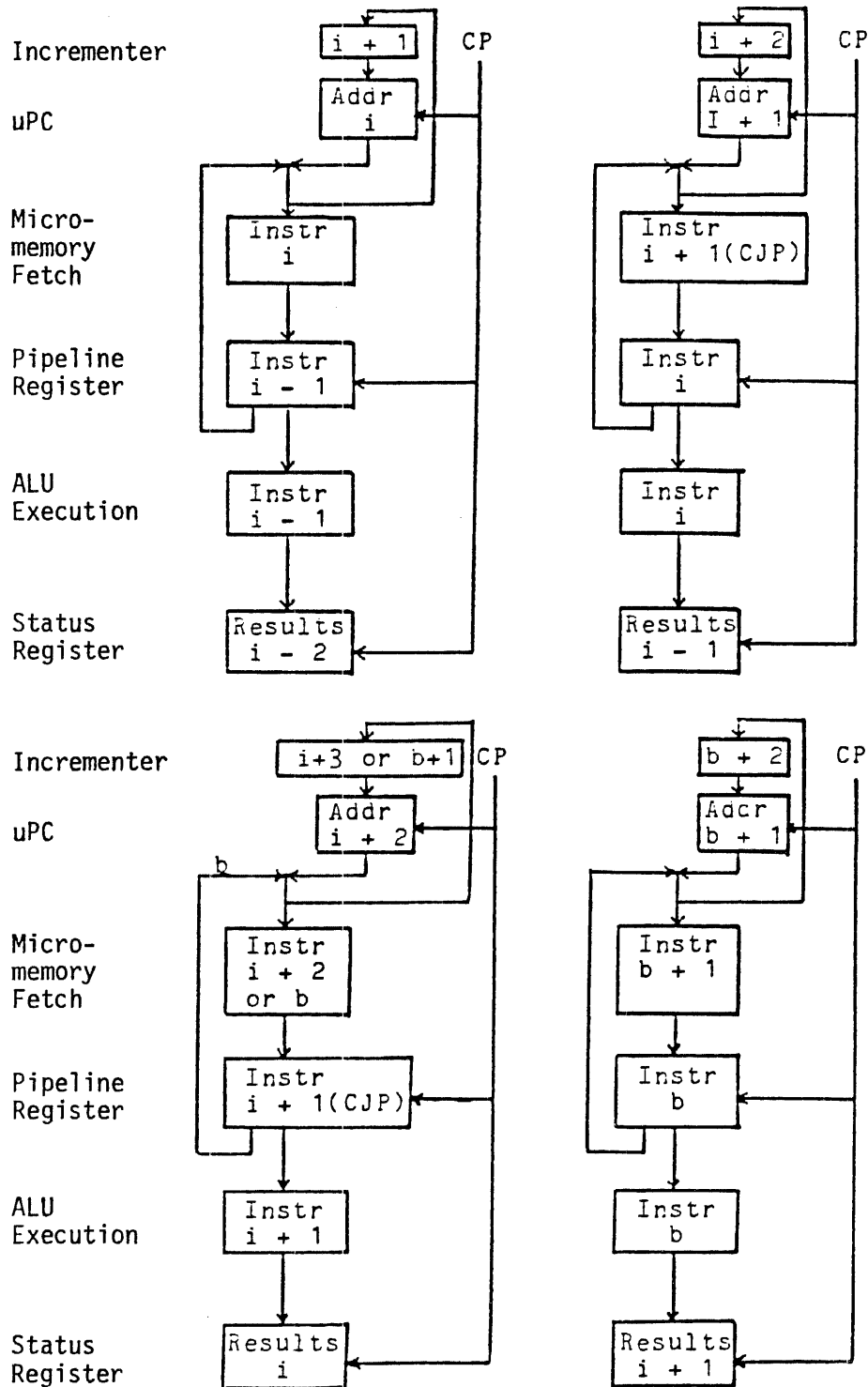
1-1150

ADVANCED MICRO DEVICES 

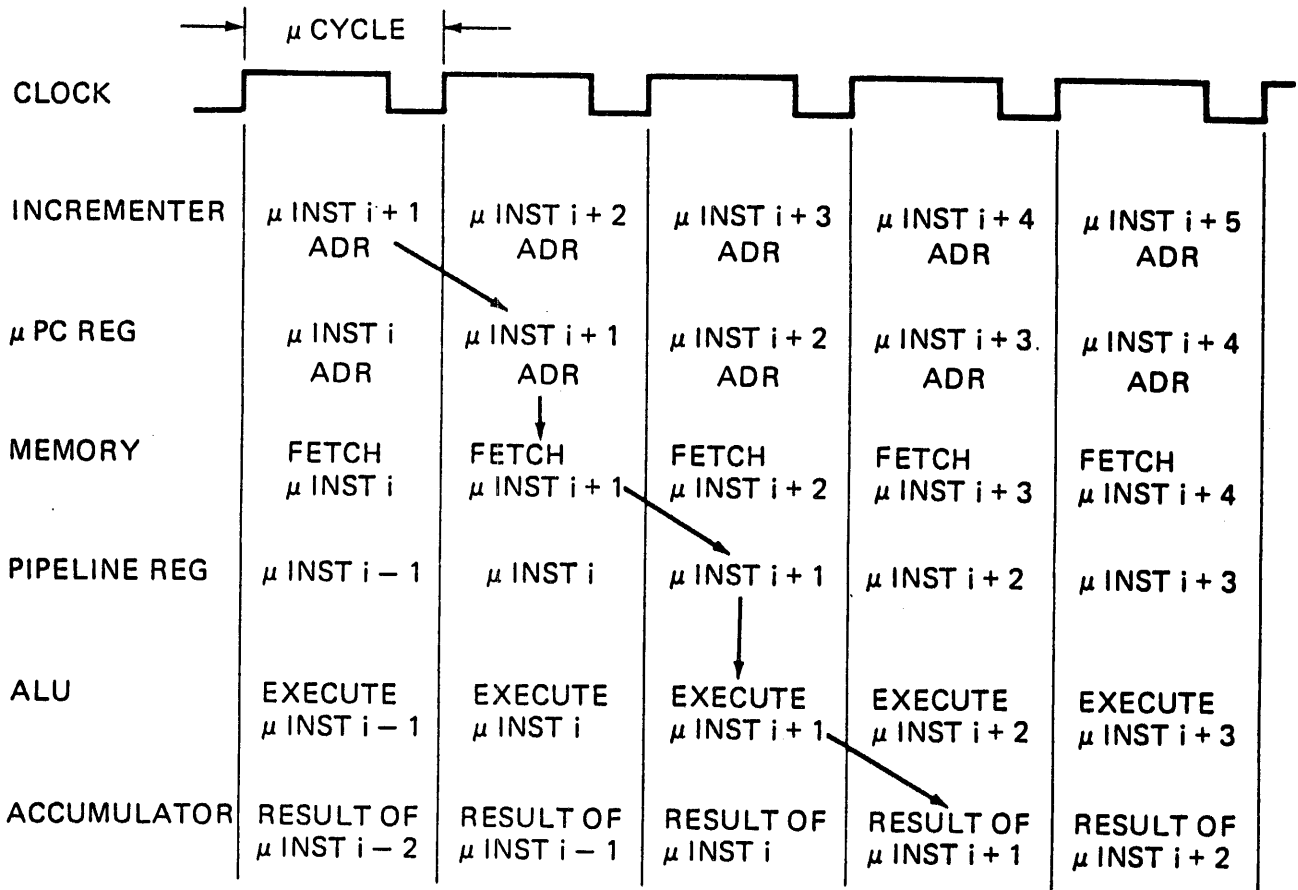


TIMING

Pipeline with Incrementer - Branch Taken

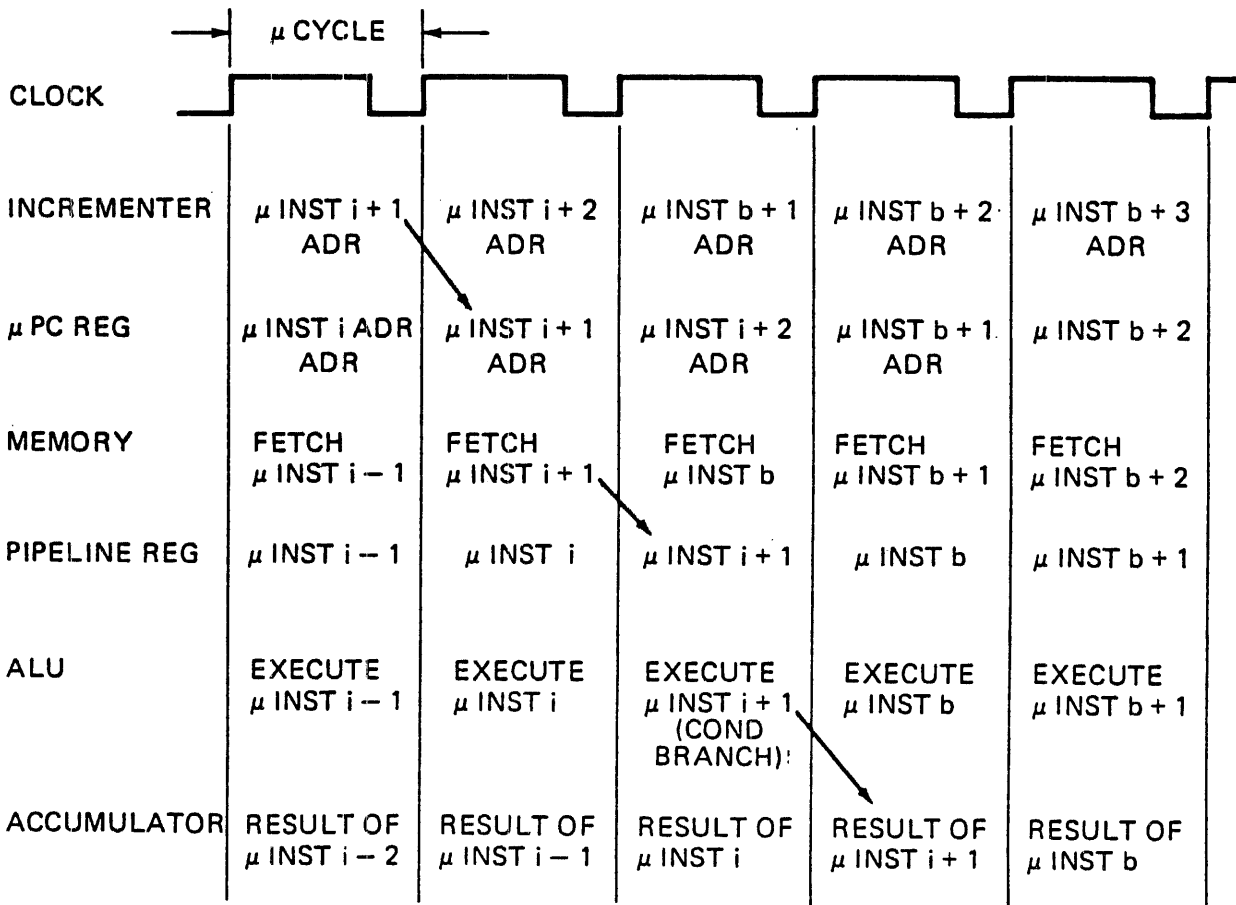


No Branch



Final Version Architecture

Branch Taken - No Penalty



FURTHER IMPROVEMENTS IN MICROPROGRAM CONTROL

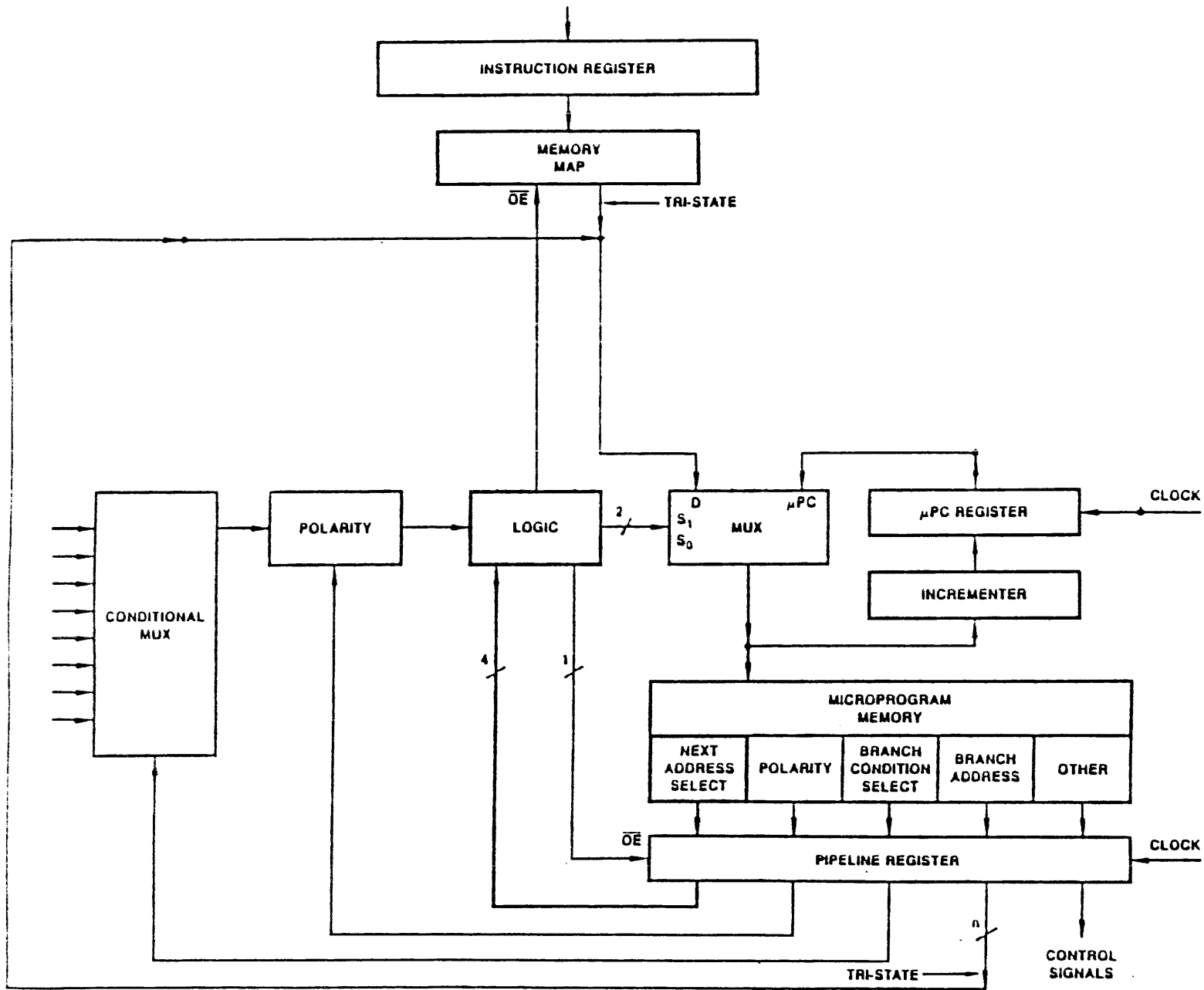
1-1200

ED2900A

1-1200

SUBROUTINE CONTROL FLOW (branching)

- There are cases where a branch to a routine and then a return to the main microprogram flow upon the routine's completion is desired. It may be desired to do this branching from several different places in the main program.
- Subroutine organizations, as used in other programming languages, provide a structured way of accomplishing this task.
- The ability to perform nested subroutines is also desired, that is, where one subroutine can call another subroutine and so forth.
- Subroutines support structured programming concepts, especially the implementation of modular code and functionality.
- To facilitate these features, the following capabilities are required to perform a subroutine (a branch and return sequence):
 - a stack to save the micromemory address
 - a top-of-stack (TOS) pointer
 - a means of accessing the top of the stack through another input to the micromemory address MUX
 - logic to control the stack operations

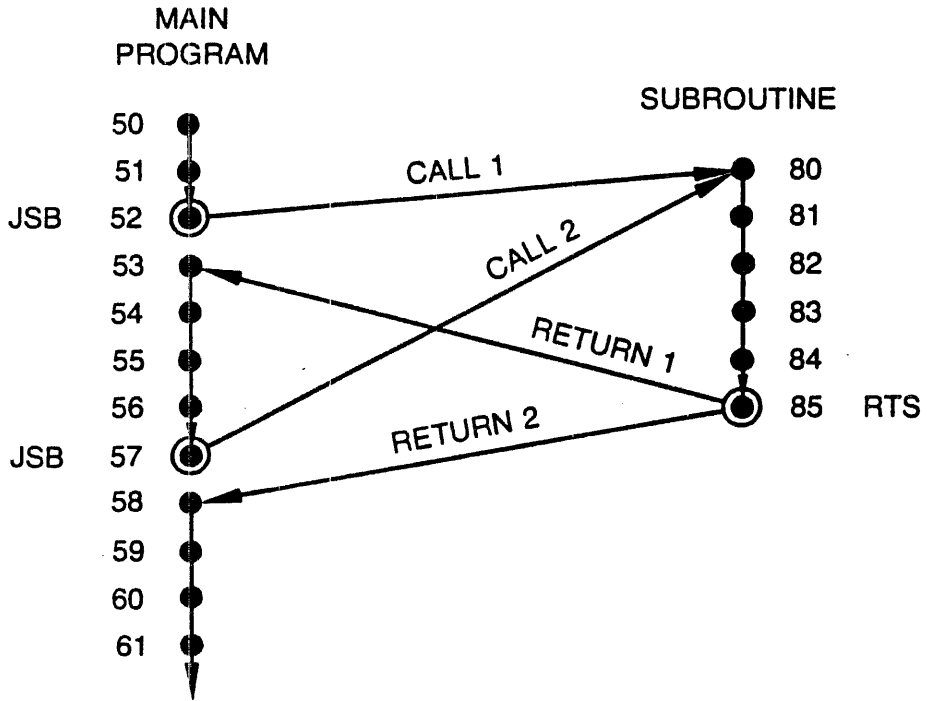


SUBROUTINES:

- Subroutines should be callable from anywhere in the microprogram.
- As with jumps/branches, subroutine calls can be conditional or unconditional.
- At the completion of the subroutine, control returns to the main macroprogram statement following the calling statement. This is an unconditional return.
- A return can be permitted prior to the completion of the subroutine based on some logical condition. This would be by definition a conditional return.
- Defined mnemonics are:
 - CJS** - conditional jump subroutine
 - CRTN** - conditional return
- Assume forced TRUE conditions will be used to implement unconditional calls and returns.
- The "logic" will control **PUSHing** the return micromemory address onto the stack and **POPping** the stack on return. The POP operation logically connects the value (microaddress on the top of the stack) to the S input on the microaddress MUX.

UNCONDITIONAL JSUB (CJS-PASS)

UNCONDITIONAL RETURN (CRTN-PASS)



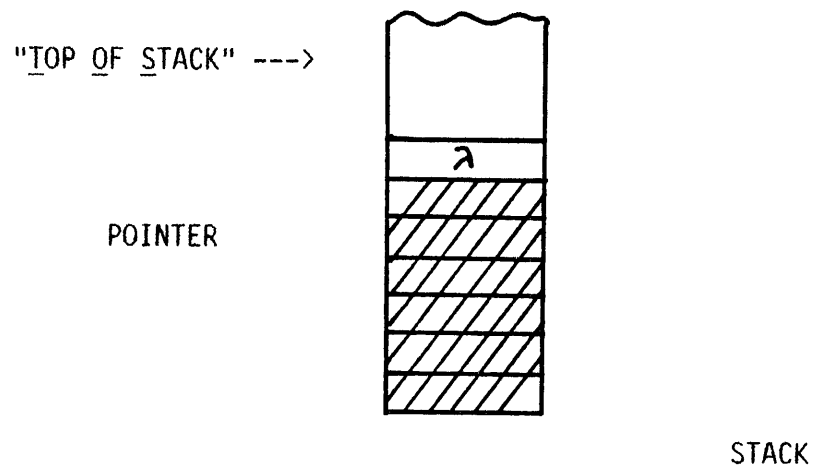
JSB: JUMP TO SUBROUTINE
 RTS: RETURN FROM SUBROUTINE

	RETURN ADDRESS	STACK	CONTENTS
START			λ
AFTER 52	53		
AFTER 85			λ
AFTER 57			58

λ means "undefined"

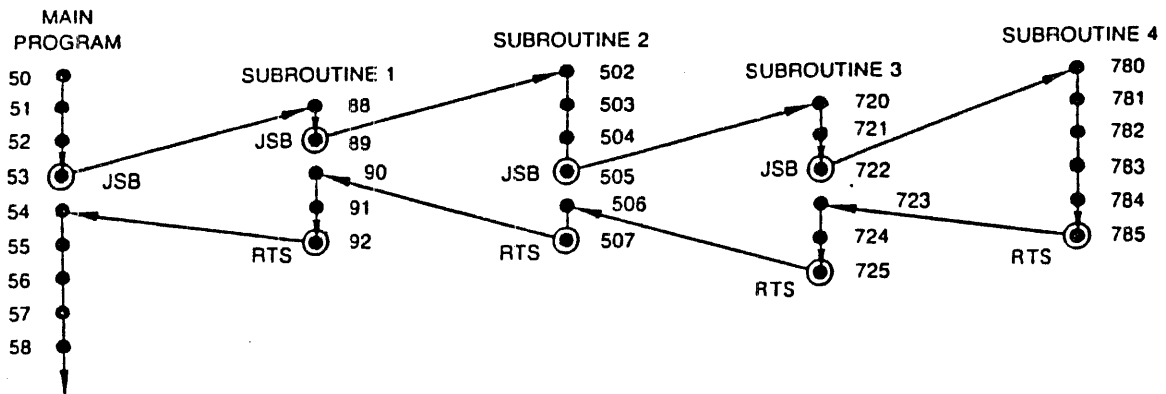
NESTED SUBROUTINES

- Occur where one subroutine calls another
- The best way to handle multiple return addresses is via a last in, first out stack and a top of stack (TOS) pointer

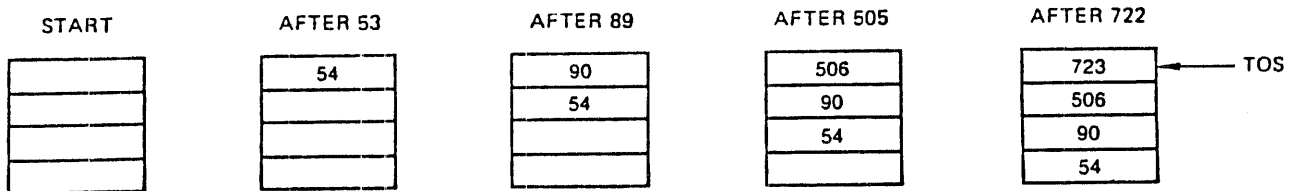


"PUSH" an address on the stack $TOS = TOS + 1$

"POP" an address off the stack $TOS = TOS - 1$



JSB : JUMP TO SUBROUTINE
 RTS: RETURN FROM SUBROUTINE



LIFO STACK CONTENTS

EXAMPLE:

- The following microroutine demonstrates a subroutine call and return:

	NEXT ADDR SELECT	P O L	COND MUX SEL	BR ADDR	

STRT:	31 CONT		X	X	
	32 CJP		TEST	L1	
	32 CONT		X	X	
	33 JP		X	L2	
L1:	104 CONT		X	X	
	105 CONT		X	X	
L2:	106 CJS		TEST	L3	
	107 CONT		X	X	
	108 JMAP		X	X	
L3:	547 CONT		X	X	
	548 CONT		X	X	
	549 CONT		X	X	
	550 CRTN		PASS	X	"unconditional return"

Possible next address controls for our CCU so far:

CONT	Continue
JP	Go to branch address
CJP	If condition true then go to branch address
JMAP	Go to mapping PROM output (start address)
CJS	If condition true then go to subroutine address
CRTN	If condition true then go to <TOS>

LOOPS (ITERATION)

- There are many algorithms that require one or more statements to be repeated for X number of times (DO loop)
- One way to implement a X-times loop is via a loop starting address and a decrementing counter.

- **Example -**

BEGIN LOOP:

REGISTER <--- START ADDRESS

COUNTER <-- X - 1 note counter is 1 less
than times loop is executed

END LOOP:

IF COUNTER = 0 GO TO <uPC> (leave loop and continue)

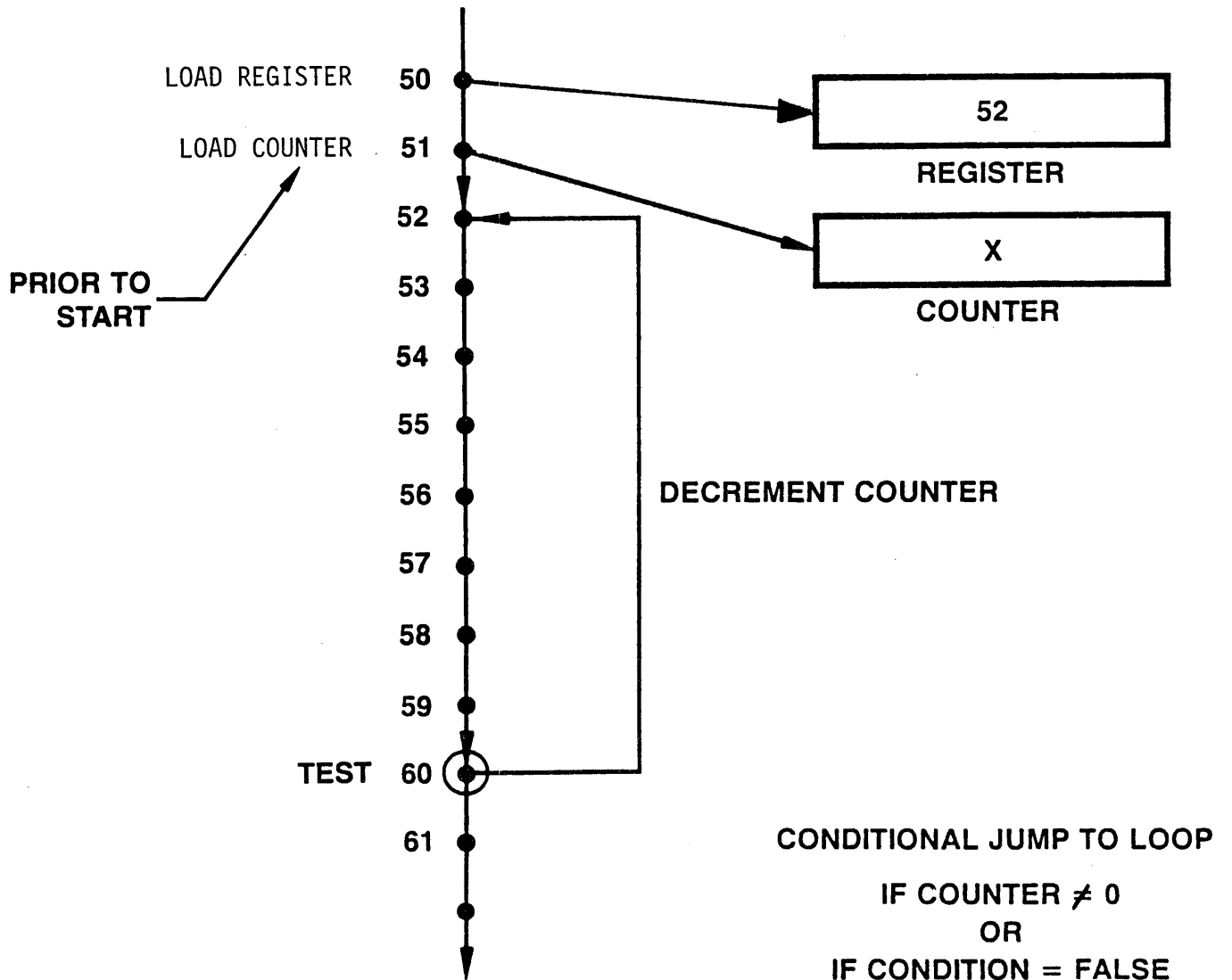
IF COUNTER ≠ 0 GO TO <REGISTER> (loop again at
START ADDRESS)

- Note that loop's starting micromemory address could also be stored in the branch address field at the last microinstruction in the loop instead of the register (an additional required storage location).
- A loop may also occur where one or more statements are repeated until some condition exists or event occurs (referred to as **DO-WHILE** or **DO-UNTIL** loops).

IF TEST = TRUE GO TO <uPC>

IF TEST = FALSE GO TO <REGISTER>

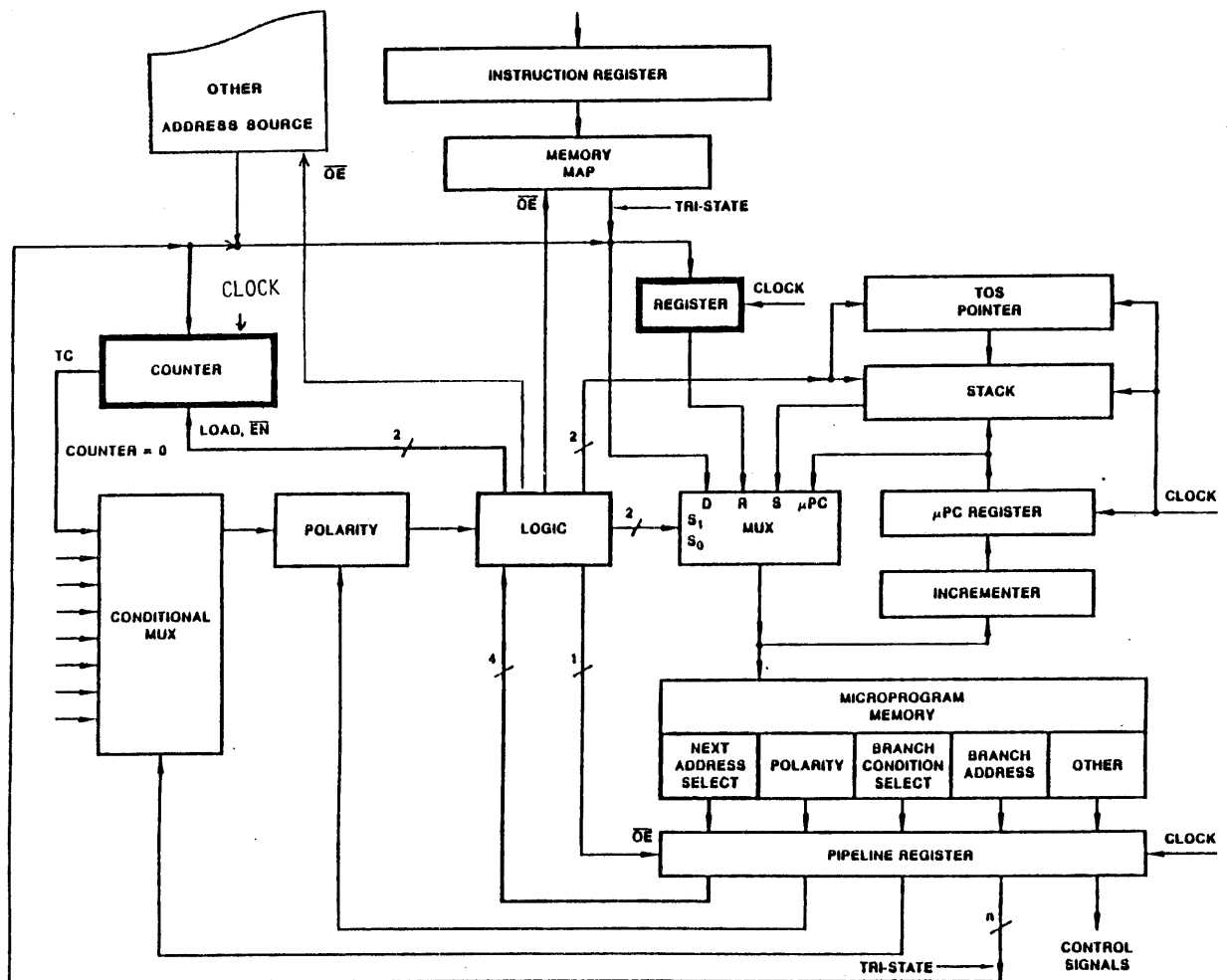
LOOPS



MODIFIED SEQUENCER STRUCTURE FOR LOOP ITERATION:

- A counter was added to hold the loop count. A source is needed to hold the original value of the count for transfer to the counter. Another field in the microword could be added. However, an overlapped or shared field could be used.
- A **shared field** is a field that has one meaning for some operations and another meaning for other operations. Often an extra bit is added to the microword to indicate which meaning is being used, but in this case the next address select field does the job.
- Sharing fields (also called **vertical microprogramming**) should be used with care. However, the example under consideration is commonly used with Am2900 parts.
- The branch address field (which is only used during jump or CJS instructions) is "overlapped" with the counter value field. Note that the count is thus limited to n bits.
- Some type of next address select code is needed that will determine the location (register, microinstruction, stack) of loop starting address.
- Finally, an extra tri-state enable is added for flexibility for selecting other external microaddress values. In this development, it will be used for enabling interrupt vectors.

Complete CCU



SEQUENCER NEXT ADDRESS CONTROL

- The next figure presents the logic block diagram for next address control. The following signals are thus defined:

- **Inputs:**
 - Next address select from pipeline (microword) - assume 4-bits will suffice

 - Condition code; - output of condition code MUX

 - OE (low); - allows all outputs to be tri-stated

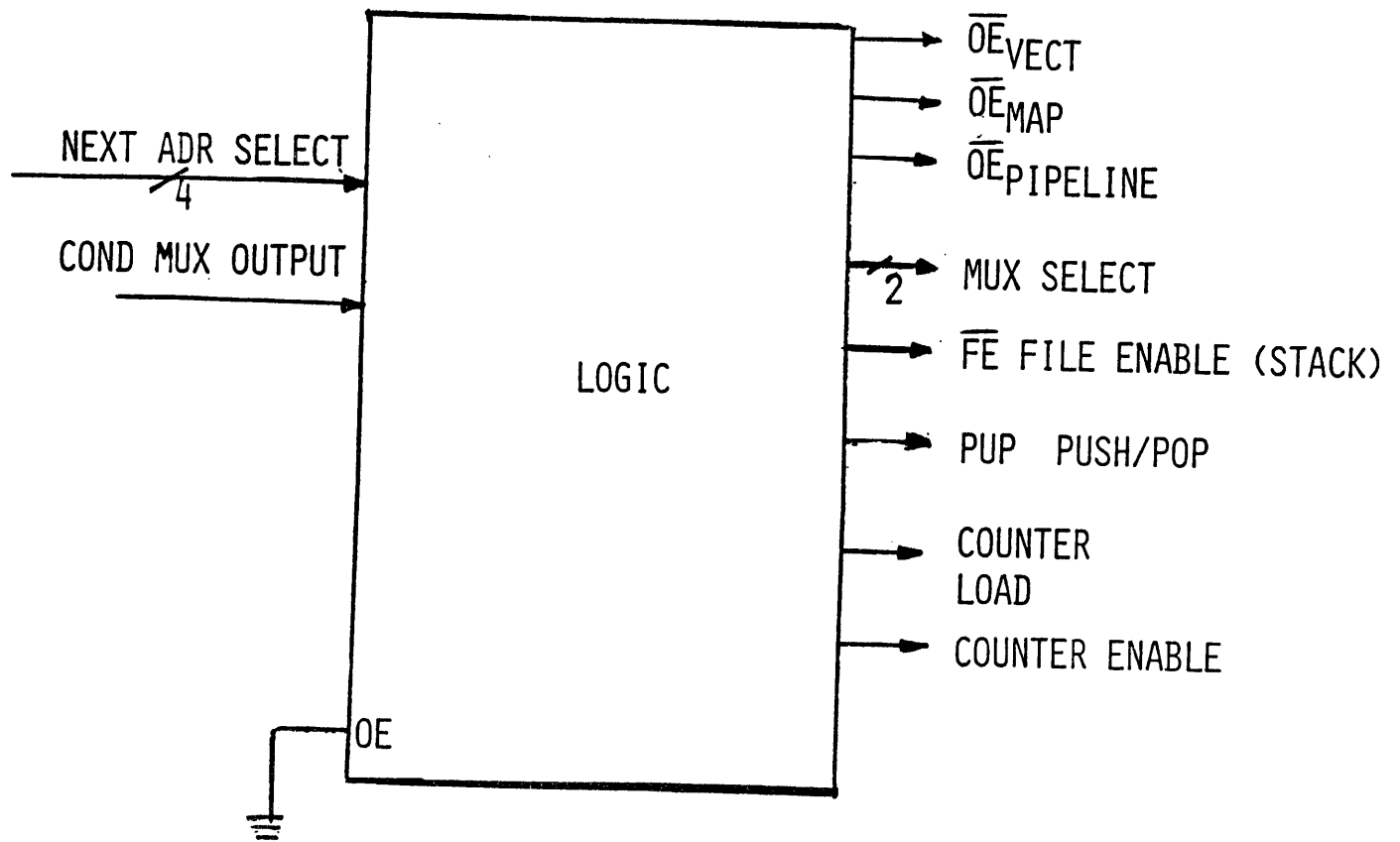
- **Outputs:**
 - Three output enables for tri-state sources
 - MAP - for mapping PROM
 - PIPELINE - for pipeline branch address field
 - VECT - extra (intended for interrupt vectors)

 - MUX select for control of the micromemory address MUX

 - Counter load and enable for loop counter control

 - FE file enable causes a stack operation

 - PUP determines stack push or pop

Summary of Next Address Control**Logic Block**

1-1340

ED2900A

1-1340

CCU IMPLEMENTATIONS
USING Am2900/Am29100 FAMILY PARTS

MICROSEQUENCER SELECTION

- There are three choices of Am2900 chip sets available for implementing a control unit.
- The first consists of the Am2910 microprogram controller.
- The second is the Am29112 microprogram controller.
- The third consists of the Am29811 next address control unit with either the Am2909 or Am2911 microprogram sequencer (bit slice).

PRIMARY DIFFERENCES BETWEEN APPROACHES

Am2910

- The Am2910 is a single package, containing sequencer, next address control logic, and a combined counter/register.
- The Am2910 is not a bit-slice, but has a 12-bit micromemory address output (4K micromemory addressing).
- The Am2910 includes vector-enable output

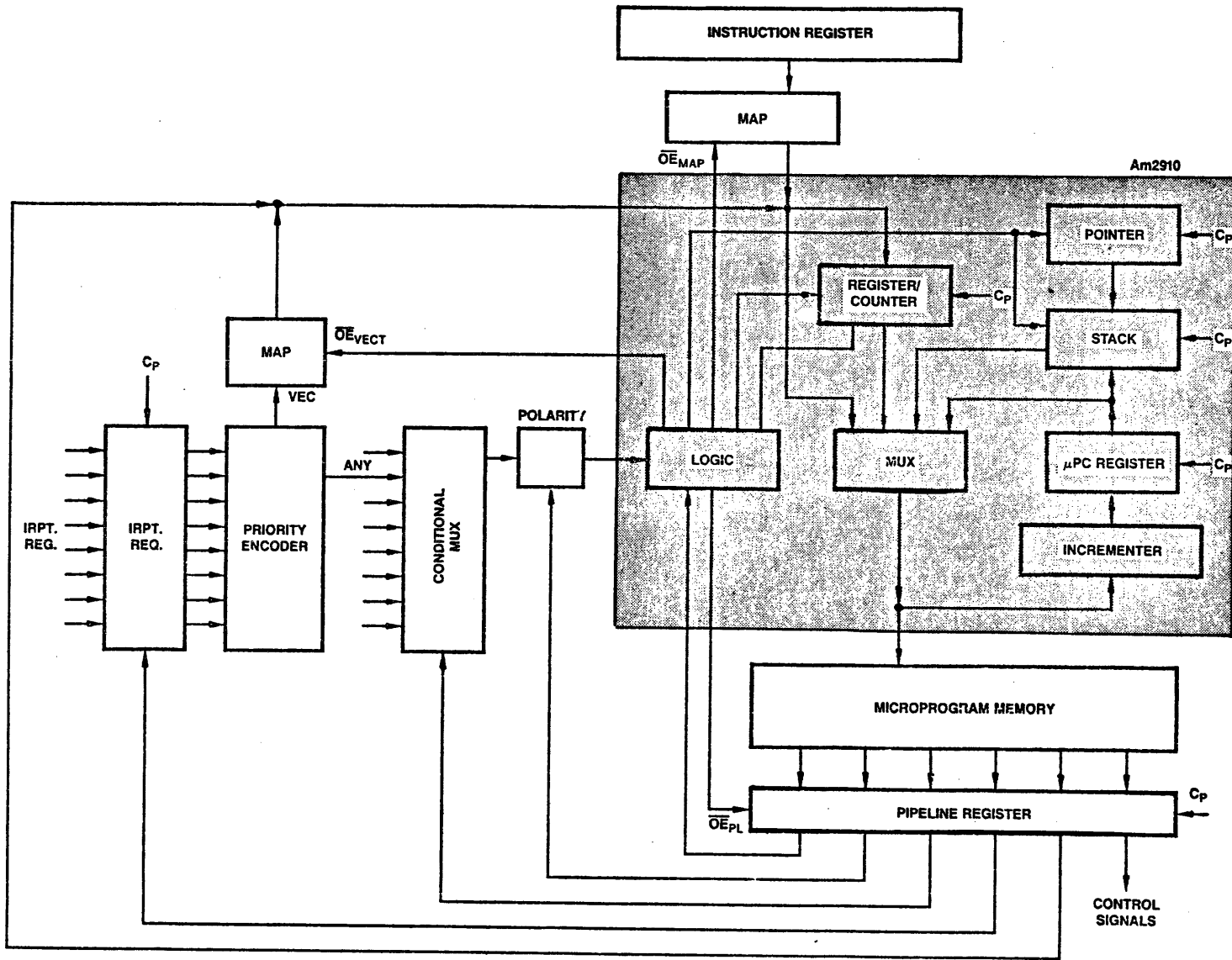
Am29112

- The Am29112 is similar to the Am2910 in general structure, but is an 8-bit slice expandable to two for addressing 64K of micromemory.
- The Am29112 stack is 33 registers deep.
- The Am29112 also features direct, multiway, relative and program-counter-relative addressing modes, along with vectored interrupts.

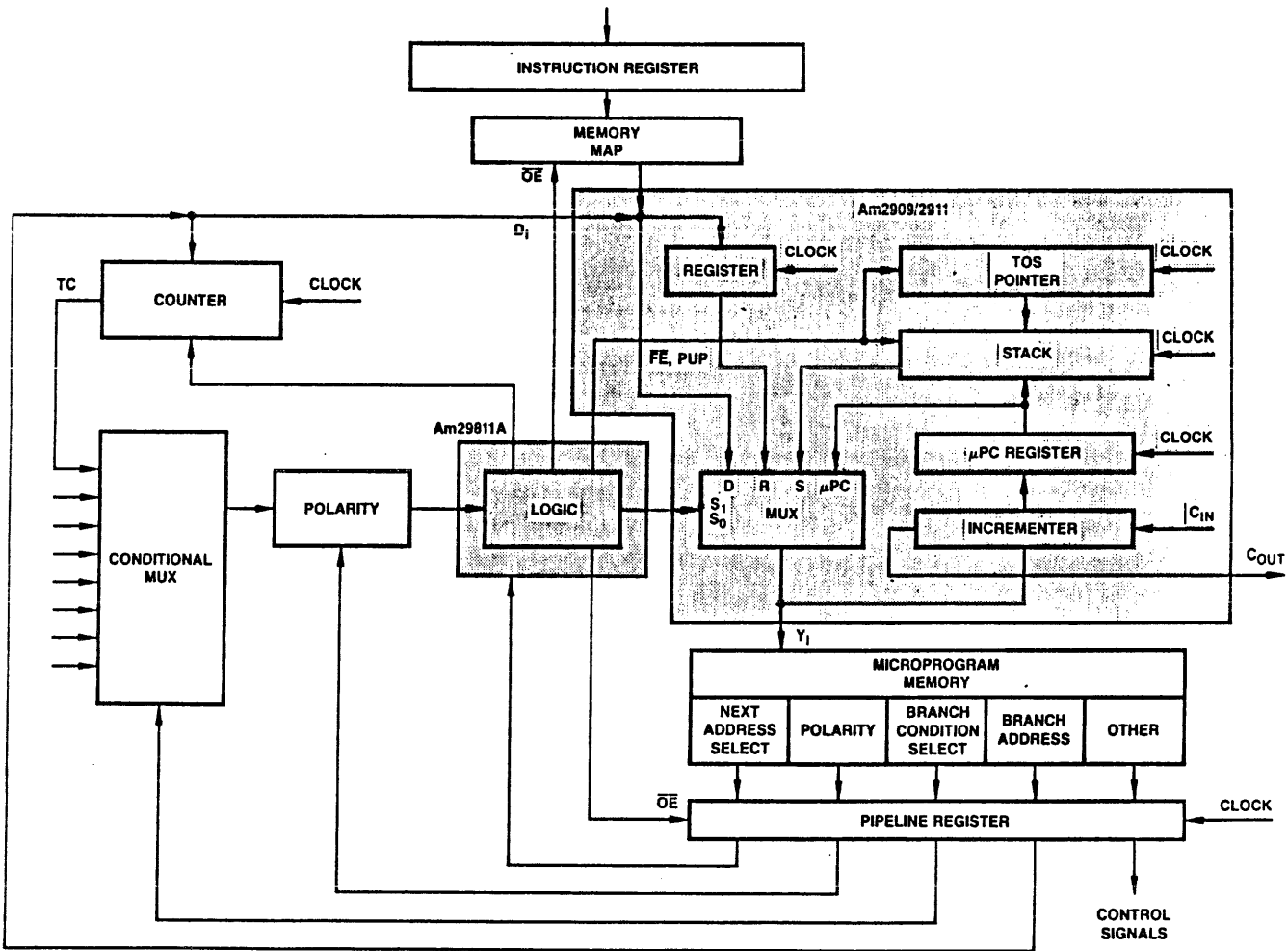
(The Am2910 will be emphasized with possible alternate capabilities discussed with the Am29811 and the Am2909/2911 and the Am29112)

Am2909/2911 SEQUENCERS

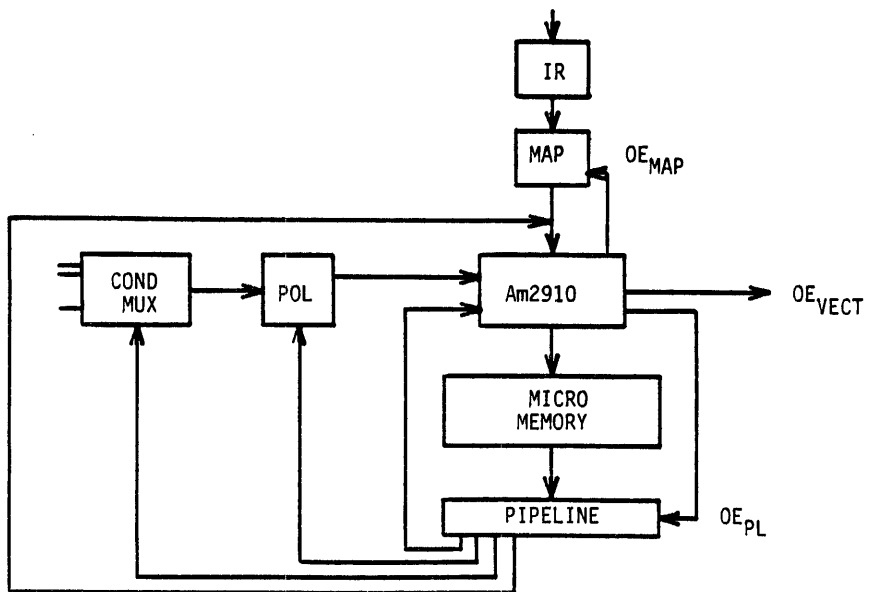
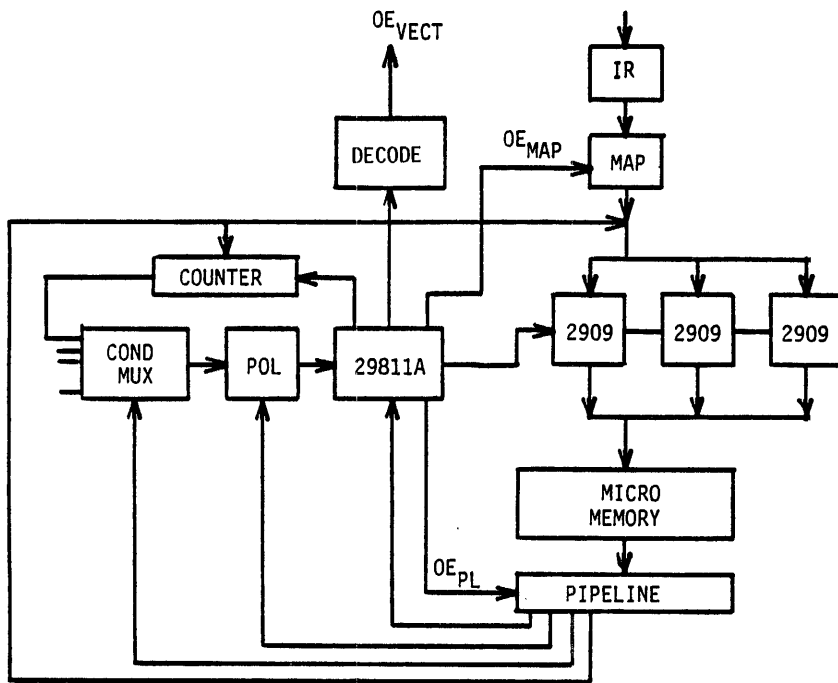
- The Am2909/2911 is a 4-bit sequencer slice, allowing any width of microprogramming addressing and requires next address control logic.
- The Am2909 has four input bits OR'ed with its output for use with the Am29803 for doing 16-way branches (case statement).
- The Am29811 next address control logic has the same instruction set as the Am2910 except for the Am2910's three-way-branch.



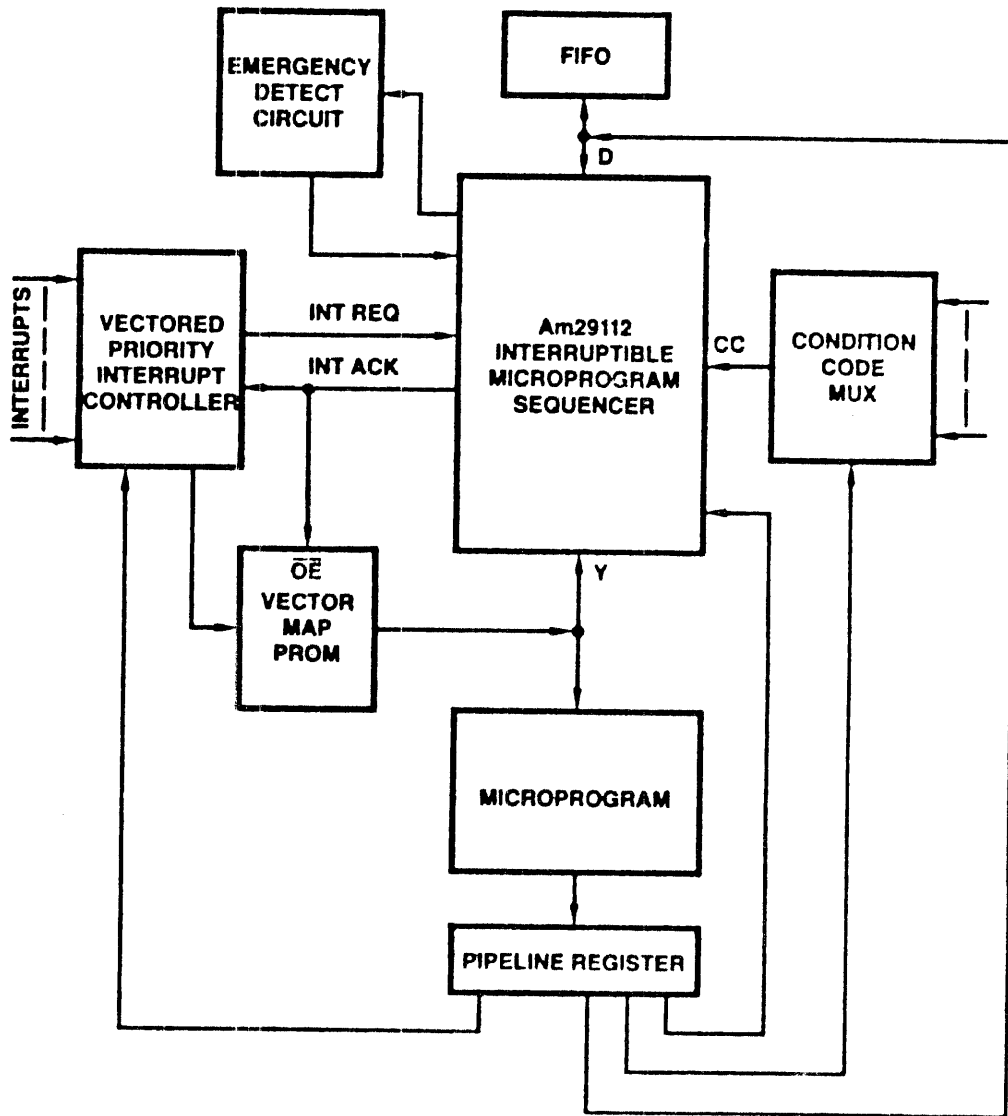
Am2910



Am2909/Am2911



Am29112 in a Single Pipelined System



1-1430

ED2900A

1-1430

SUPERSEQUENCER

Am2910

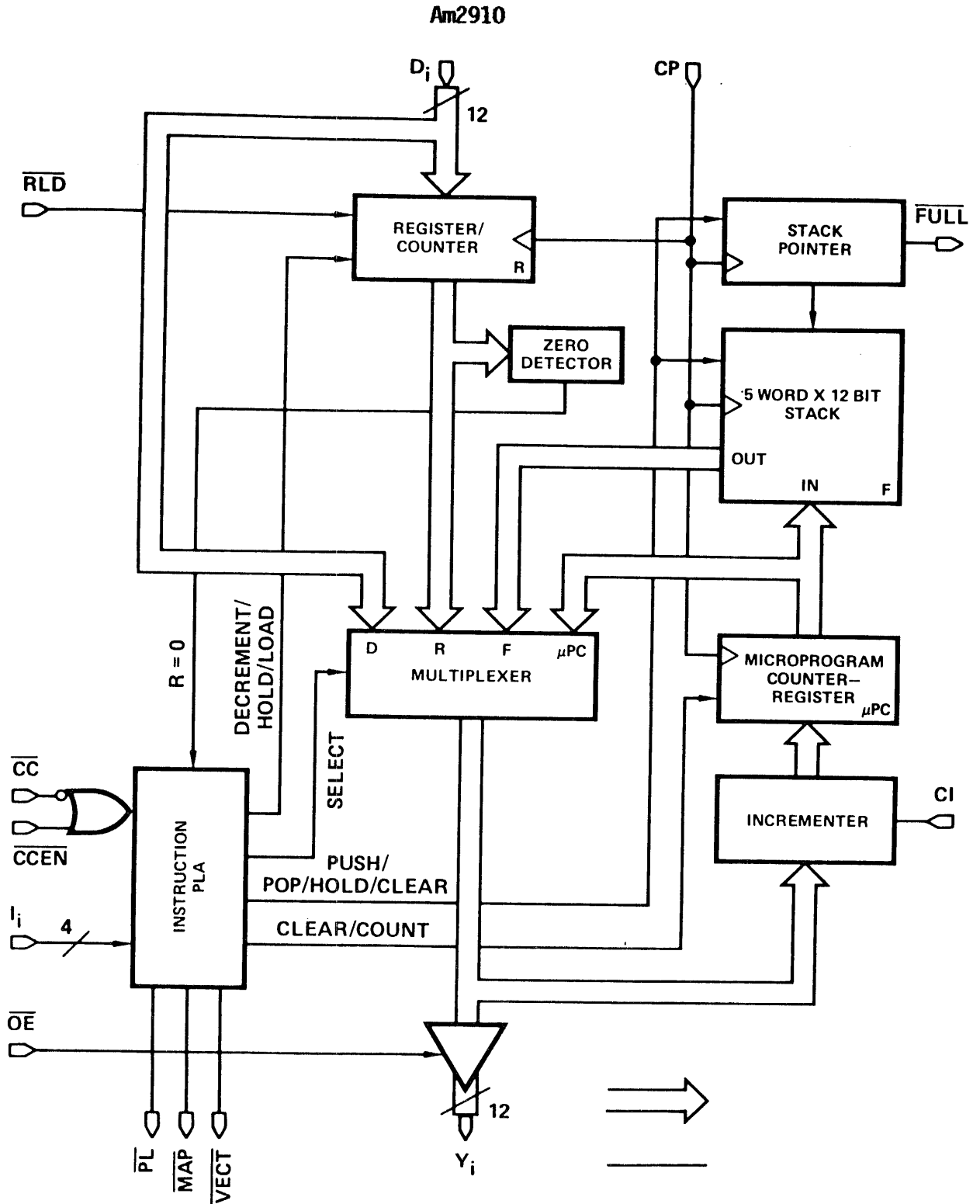
Am2910 DISTINCTIVE CHARACTERISTICS

- Twelve bit address output
- Four address sources - D, R, File (Stack output), uPC
- Internal loop counter
- Five deep subroutine stack - Am2910, nine deep - Am2910A
- Conditional test input
- Sixteen powerful microinstructions
- \overline{OE} for three next address jump sources
- Fast microprogram execution
- Additional control pins
(discussed in detail later)

RLD - register latch

CCEN - for forced pass

CI - for inhibiting incrementer



Am2910 INSTRUCTION SET SUMMARY**START:**

JZ Jump Zero (Reset)

SEQUENCE:

CONT Continue

BRANCH:

JMAP Jump Map

CJP Conditional Jump to Pipeline

CJV Conditional Jump to Vector

JRP Conditional Jump Register or Pipeline

CJPP Conditional Jump to Pipeline and POP Stack

SUBROUTINE:

CJS Conditional Jump to Subroutine (CJP and PUSH)

JSRP Conditional Jump to Subroutine where Start Address
is the Register or Pipeline

CRTN Conditional Return

LOOPING:

LDCT Load Counter and Continue

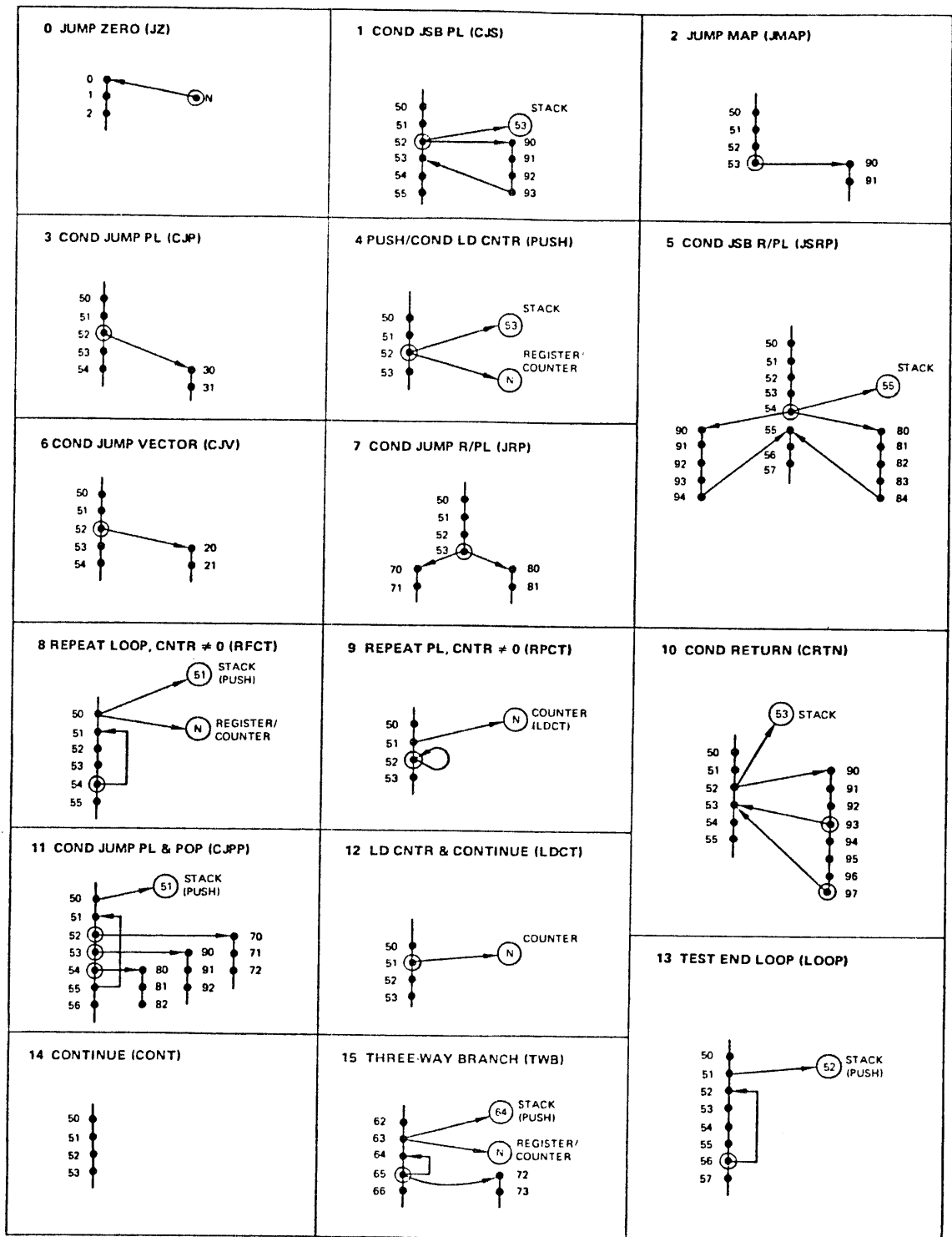
PUSH Push Micro-PC on Stack, Conditional Load Counter
and ContinueRPCT Repeat Loop if Counter = \emptyset , Start Address on Stack

LOOP Repeat Loop until TEST = TRUE, Start Address on Stack

TWB Repeat Loop if TEST = FALSE and Counter = \emptyset ELSE IF TEST = FALSE and COUNTER = \emptyset , Go to Pipeline

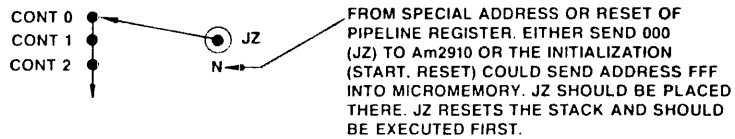
ELSE IF TEST = TRUE Continue

Am2910



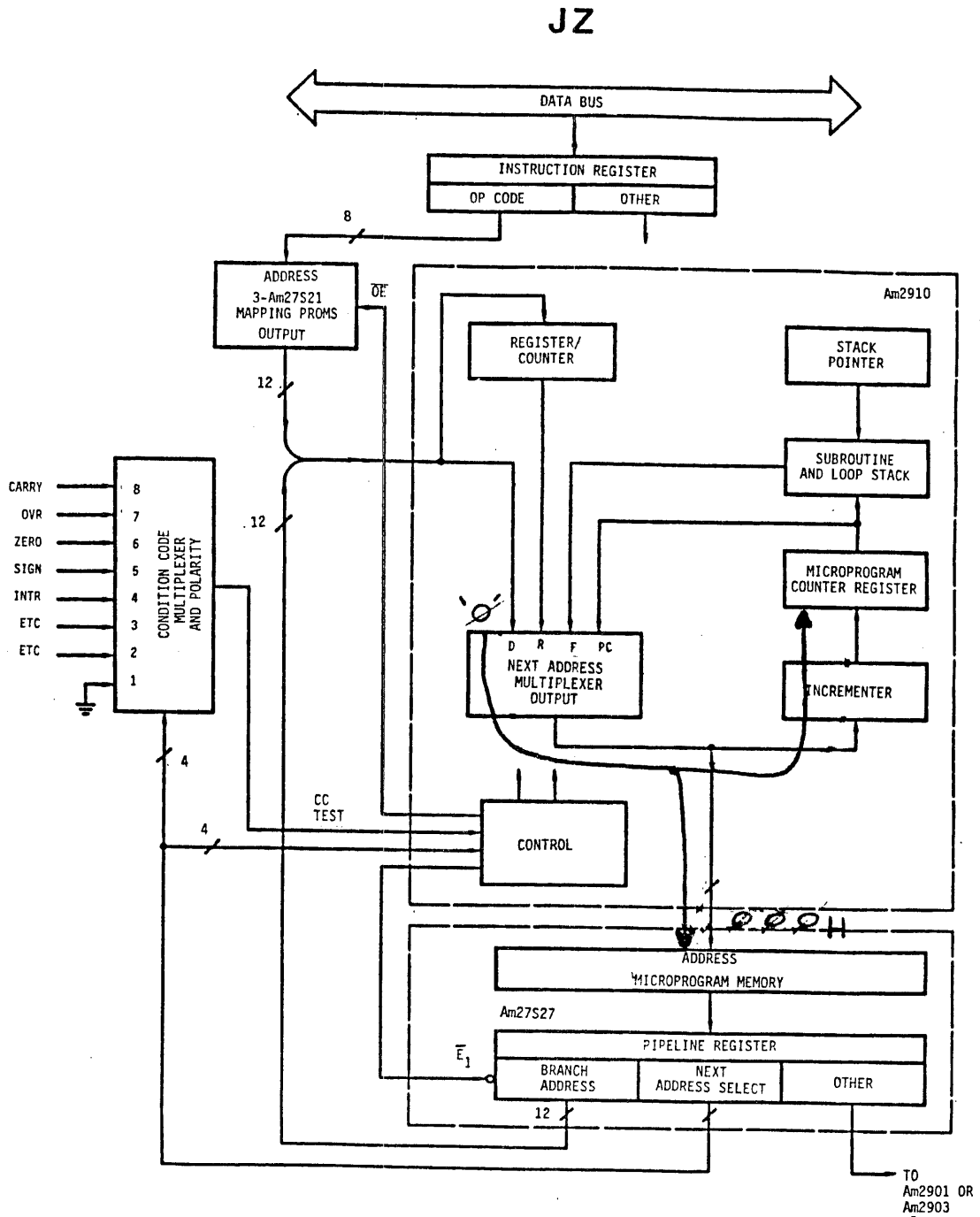
JZ Jump to Address Zero

ADDRESS (HEX)	LABEL	2910 INSTR	COND MUX	BRANCH ADDRESS
0	START:	CONT	#	# <----- Start Address
1		CONT	#	#
2		CONT	#	#
.				
.				
.				
n		JZ	#	# <----- Hardwired Start Address



\overline{CC}	COUNTER = 0 LINE	STACK	ADDRESS SOURCE	REGISTER COUNTER	\overline{OE}
X	X	CLEAR	0	NC	PL

Figure 4-9. Jump zero (JZ, 0).



CONT Continue to Next Instruction in Sequence

ADDRESS (HEX)	LABEL	2910 INSTR	COND MUX	BRANCH ADDRESS
50		CONT	#	#
51		CONT	#	#
52		CONT	#	#
53		CONT	#	#

Sequential Program Flow



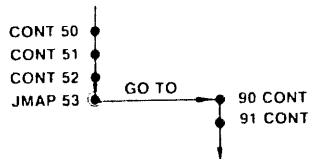
\overline{CC}	COUNTER = 0 LINE	STACK	ADDRESS SOURCE	REGISTER/ COUNTER	\overline{OE}
X	X	NC	μ PC	NC	PL

Figure 4-10. Continue (CONT, E).

JMAP Jump to Start Address (Enable Mapping PROM)

ADDRESS (HEX)	LABEL	2910 INSTR	COND MUX	BRANCH ADDRESS
50		CONT	#	#
51		CONT	#	#
52		CONT	#	#
53		JMAP	#	#

Address supplied by Map

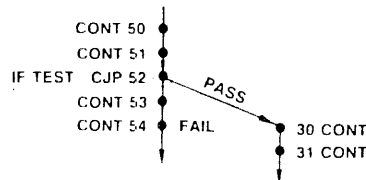


\overline{CC}	COUNTER = 0 LINE	STACK	ADDRESS SOURCE	REGISTER/ COUNTER	\overline{OE}
x	x	NC	D	NC	MAP

Figure 4-11. Jump map (JMAP. 2).

CJP Conditional Jump to Branch Address (Pipeline)

ADDRESS (HEX)	LABEL	2910 INSTR	COND MUX	BRANCH ADDRESS
30	LABELA:	CONT	#	#
31		CONT	#	#
.				
.				
.				
50		CONT	#	#
51		CONT	#	#
52		CJP	TESTA	LABELA
53		CONT	#	#
54		CONT	#	#

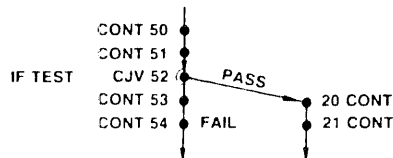


\overline{CC}	COUNTER = 0 LINE	STACK	ADDRESS SOURCE	REGISTER. COUNTER	\overline{OE}
PASS FAIL	X	NC	D μPC	NC	PL

Figure 4-12. Conditional jump pipeline (CJP. 3).

CJV Conditional Jump to Vector Map Output

ADDRESS (HEX)	LABEL	2910 INSTR	COND MUX	BRANCH ADDRESS
20		CONT	#	#
21		CONT	#	#
.				
.				
.				
50		CONT	#	#
51		CONT	#	#
52		CJV	ANYI	# <--- Branch Address from Vector Map
53		CONT	#	#
54		CONT	#	#

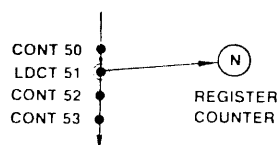


\overline{CC}	COUNTER = 0 LINE	STACK	ADDRESS SOURCE	REGISTER/ COUNTER	\overline{OE}
PASS FAIL	X	NC	D μPC	NC	VECT

Figure 4-13. Conditional jump vector (CJV. 6).

LDCT Load the Register/Counter and Continue

ADDRESS (HEX)	LABEL	2910 INSTR	COND MUX	BRANCH ADDRESS
50		CONT	#	#
51		LDCT	#	VALUE-1
52		CONT	#	#
53		CONT	#	#



\overline{CC}	COUNTER = 0 LINE	STACK	ADDRESS SOURCE	REGISTER/ COUNTER	\overline{OE}
X	X	NC	μ PC	LOAD	PL

Figure 4-14. Load counter and continue (LDCT, C). This instruction must be executed before a loop instruction or a jump which used the register.

JRP Conditional Jump to Register or Branch Address (Pipeline)

ADDRESS (HEX)	LABEL	2910 INSTR	COND MUX	BRANCH ADDRESS
27		LDCT	#	REGADR <--- Load Address into Register:
.				
.				
50		CONT	#	#
51		CONT	#	#
52		CONT	#	#
53		JRP	TESTB	PIPEADR <--- If True Go To PIPEADR:
.				
.				
70	REGADR:	CONT	#	#
71		CONT	#	#
.				
.				
80	PIPEADR:	CONT	#	#
81		CONT	#	#

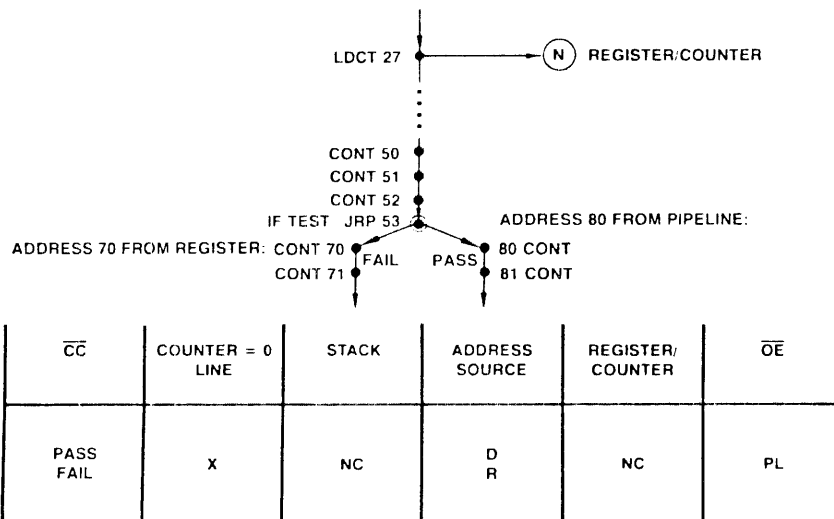
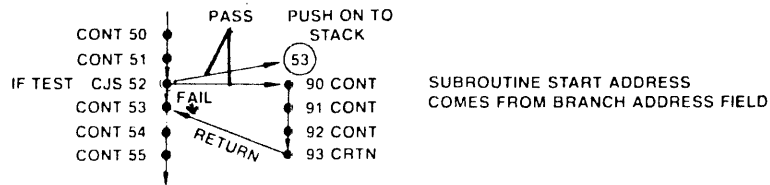


Figure 4-15. Conditional jump register/pipeline (JRP, 7). LDCT must have been executed somewhere ahead of JRP.

CJS Conditional Jump to Subroutine Address

ADDRESS (HEX)	LABEL	2910 INSTR	COND MUX	BRANCH ADDRESS	
50		CONT	#	#	
51		CONT	#	#	
52		CJS	TESTC	SUBADR	<--- GOSUB if True
53		CONT	#	#	<--- Where SUB Returns
54		CONT	#	#	
55		CONT	#	#	
.					
.					
.					
90	SUBADR:	CONT	#	#	
91		CONT	#	#	
92		CONT	#	#	
93		CRTN	PASS	#	<--- Unconditional Return



\overline{CC}	COUNTER = 0 LINE	STACK	ADDRESS SOURCE	REGISTER- COUNTER	\overline{OE}
PASS FAIL	X	PUSH NC	D μ PC	NC	PL

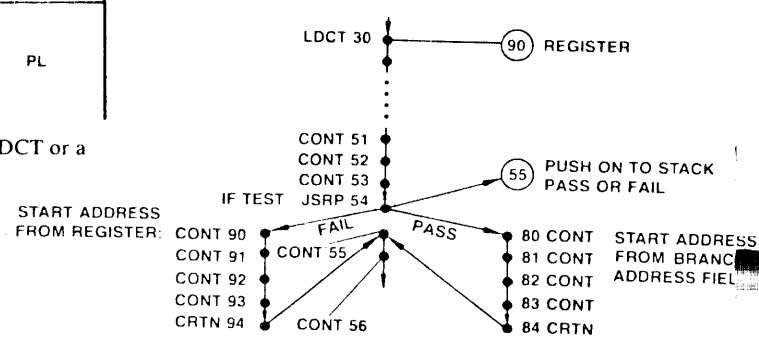
Figure 4-16. Conditional jump subroutine from pipeline (CJS, 1).

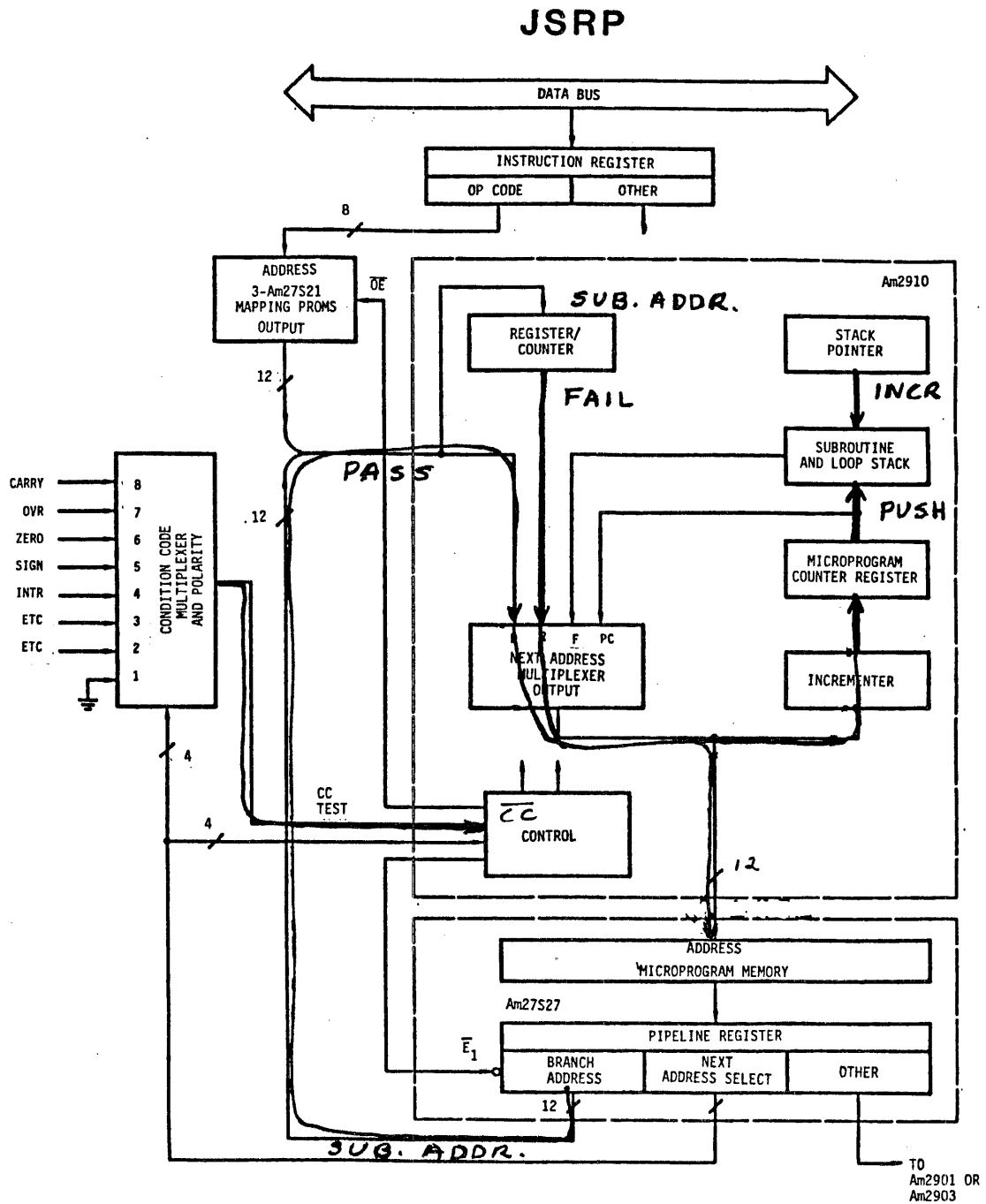
JSRP Conditional Jump to Subroutine (Register or Pipeline)

ADDRESS (HEX)	LABEL	2910 INSTR	COND MUX	BRANCH ADDRESS	
30		LDCT	#	SUBADRF	<--- Load Register with Subroutine Address
.					
.					
51		CONT	#	#	
52		CONT	#	#	
53		CONT	#	#	
54		JSRP	TESTE	SUBADRT	<--- If TRUE, go to SUBADRT
55		CONT	#	#	<--- Where Subroutine Returns
.					
.					
80	SUBADRT:	CONT	#	#	
81		CONT	#	#	
82		CONT	#	#	
83		CONT	#	#	
84		CRTN	PASS	#	<--- Unconditional Return
.					
.					
90	SUBADRF:	CONT	#	#	
91		CONT	#	#	
92		CONT	#	#	
93		CONT	#	#	
94		CRTN	PASS	#	<--- Unconditional Return

\overline{CC}	COUNTER = 0 LINE	STACK	ADDRESS SOURCE	REGISTER COUNTER	\overline{OE}
PASS FAIL	X	PUSH	D R	NC	PL

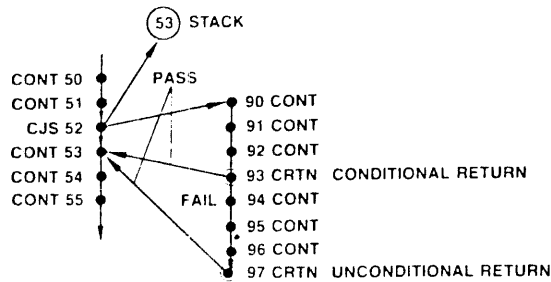
Figure 4-17. Conditional jump subroutine register/pipeline (JSRP, 5). LDCT or a register load must occur somewhere prior to JSRP.





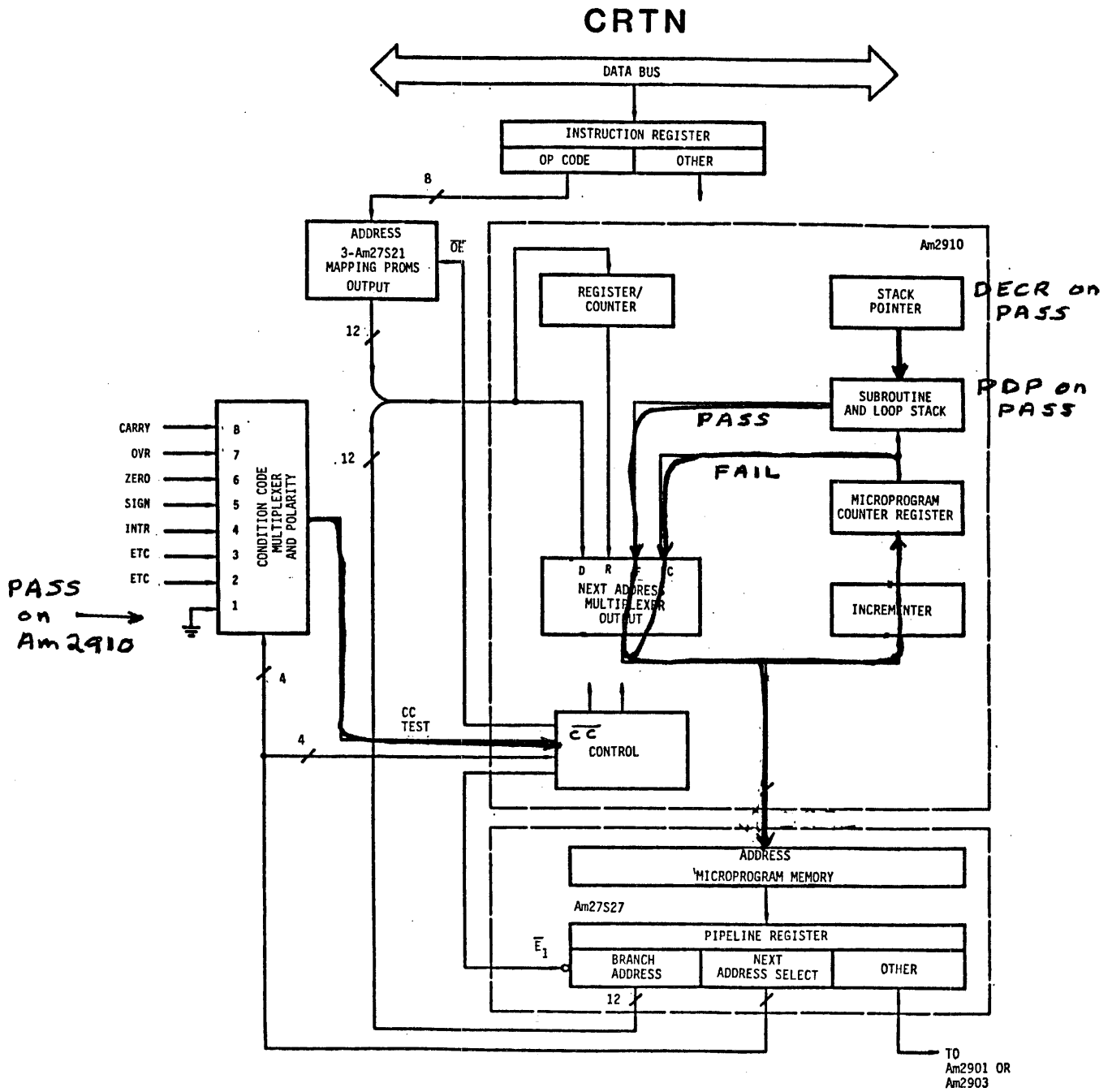
CRTN Conditional Return from Subroutine

ADDRESS (HEX)	LABEL	2910 INSTR	COND MUX	BRANCH ADDRESS
50	START	CONT	#	#
51		CONT	#	#
52		CJS	TESTF	SUB90
53		CONT	#	#
54		CONT	#	#
55		CONT	#	#
.				
.				
.				
90	SUB90:	CONT	#	#
91		CONT	#	#
92		CONT	#	#
93		CRTN	TESTG	# <--- Return to TOS on TRUE
94		CONT	#	#
95		CONT	#	#
96		CONT	#	#
97		CRTN	PASS	# <--- Unconditional Return



\overline{CC}	COUNTER = 0 LINE	STACK	ADDRESS SOURCE	REGISTER/ COUNTER	\overline{OE}
PASS FAIL	X	POP NC	STACK μ PC	NC	PL
DISABLE (CCEN = H OR CC = L)	X	POP	STACK	NC	PL

Figure 4-18. Conditional return (CRTN, A).



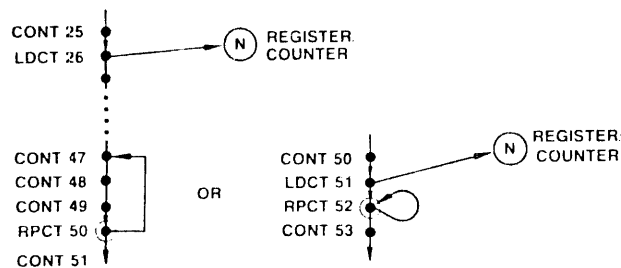
uncond RTN : choose MUX SEL = 1 or \overline{CCEN} = HIGH

RPCT Repeat Loop Until Counter = 0; Start at Branch Address

ADDRESS (HEX)	LABEL	2910 INSTR	COND MUX	BRANCH ADDRESS
25		CONT	#	#
26		LDCT	#	VAL-1
27		CONT	#	#
.				
.				
47	BEGIN:	CONT	#	#
48		CONT	#	#
49		CONT	#	#
50		RPCT	#	BEGIN

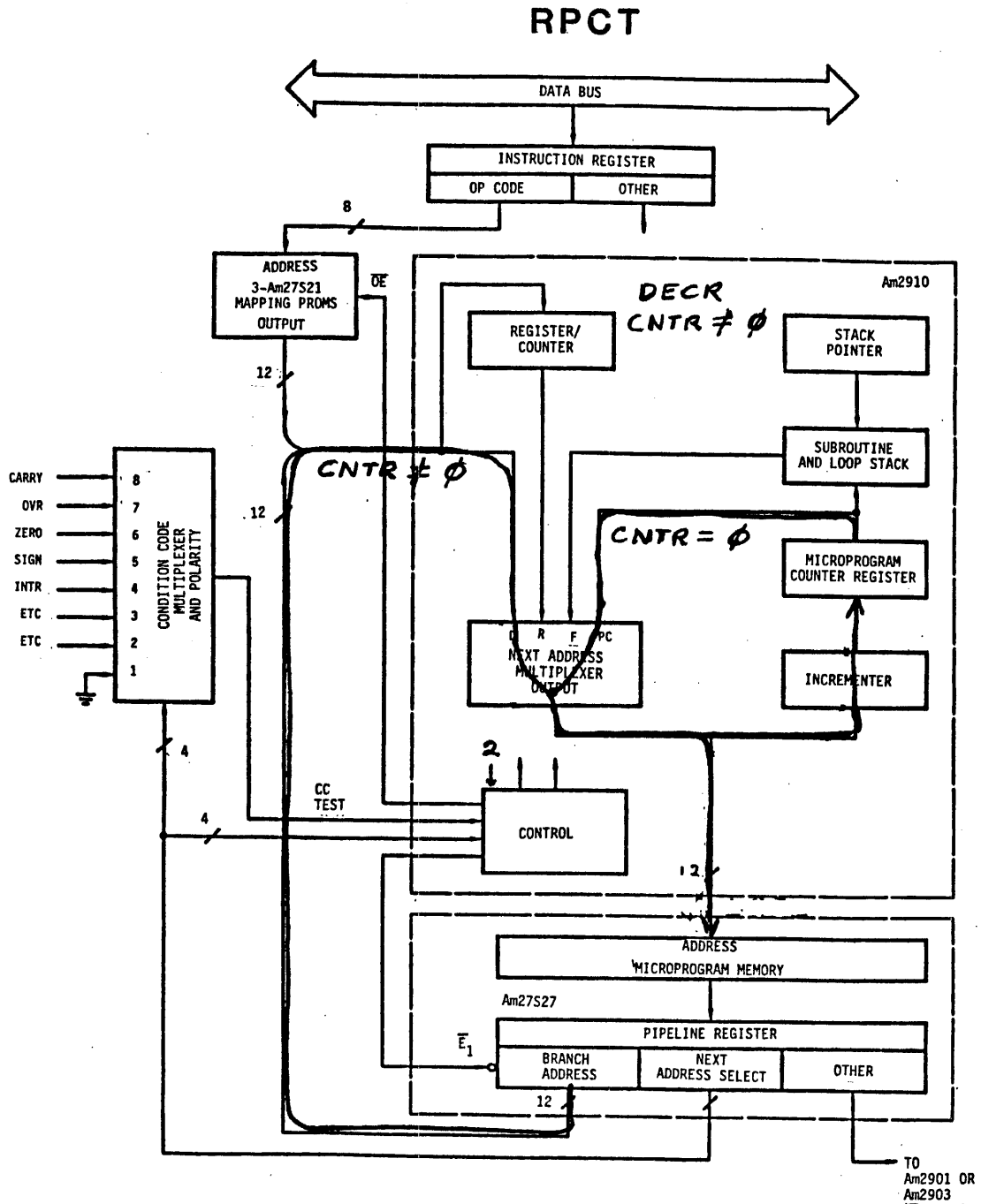
Or the One-Line Loop Version

50		CONT	#	#
51		LDCT	#	VAL-1
52	BEGIN:	RPCT	#	BEGIN
53		CONT	#	#



\overline{CC}	COUNTER = 0 LINE	STACK	ADDRESS SOURCE	REGISTER/ COUNTER	\overline{OE}
x	= 0 ≠ 0 (PART OF INSTR. PLA)	NC	μ PC D	NC DECREMENT	PL

Figure 4-19. Repeat pipeline if counter ≠ 0 (RPCT, 9). (Loop on one or more statements, beginning address of loop in pipeline [at RPCT statement].)



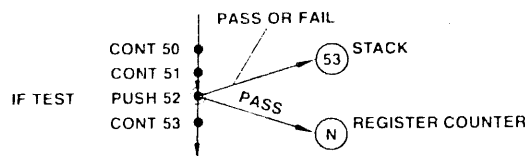
**PUSH Push Microprocessor to TOS and Continue;
Load Register/Counter Maybe**

ADDRESS (HEX)	LABEL	2910 INSTR	COND MUX	BRANCH ADDRESS	
50		CONT	#	#	
51		CONT	#	#	
52		PUSH	TESTH	VAL-1	<--- The result of TESTH only controls the Register Load
53		CONT	#	#	

PUSH may place an address or a value into the Register/Counter depending upon the value of TESTH.

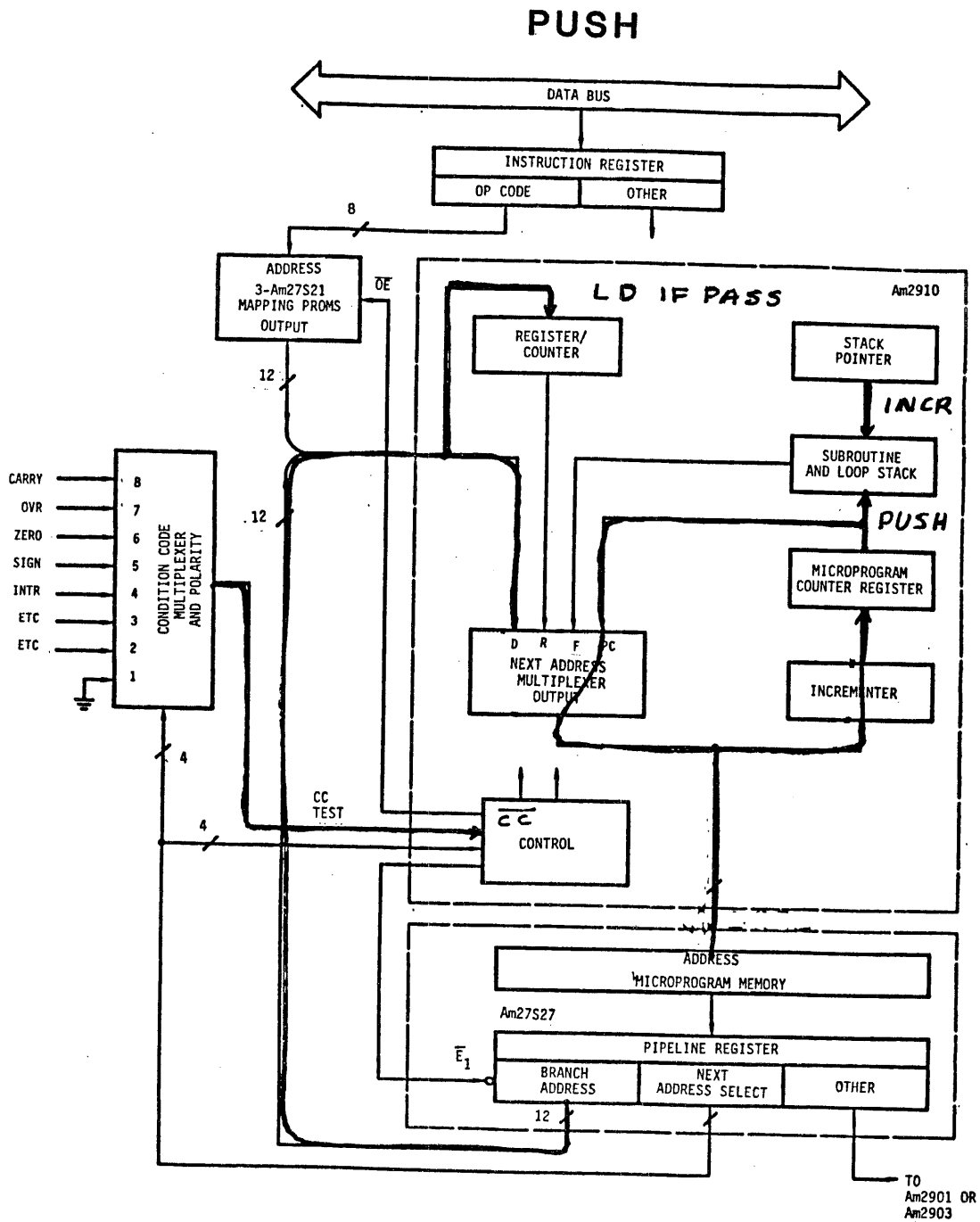
PUSH is an unconditional push of the microprogram counter onto the stack.

Instruction execution then continues.



\overline{CC}	COUNTER = 0 LINE	STACK	ADDRESS SOURCE	REGISTER: COUNTER	\overline{OE}
PASS FAIL	X	PUSH	μ PC	LOAD NC	PL

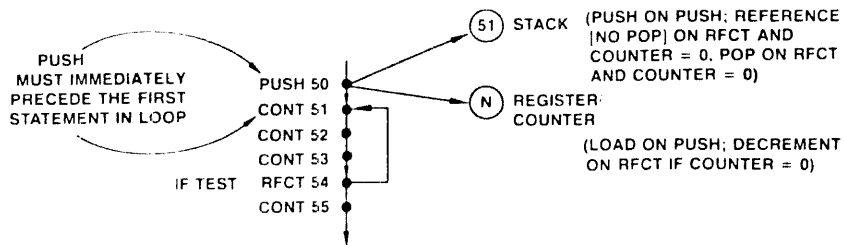
Figure 4-20. Push stack and conditional load counter (PUSH, 4). This instruction must immediately precede the first statement in a loop controlled by LOOP or RFCT.



RFCT Repeat Loop until Counter = 0; Start Address is TOS

ADDRESS (HEX)	LABEL	2910 INSTR	COND MUX	BRANCH ADDRESS	
50		PUSH	PASS	VAL-1	<--- Counter is 1 less than desired repeats
51	BEGIN:	CONT	#	#	
52		CONT	#	#	
53		CONT	#	#	
54		RFCT	#	#	<--- Return to TOS
55		CONT	#	#	

RFCT can also be used to form one-line loops.

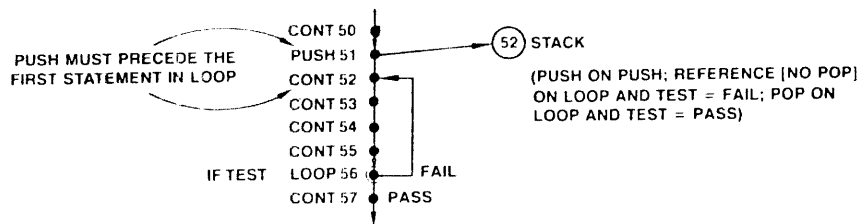


\overline{CC}	COUNTER = 0 LINE	STACK	ADDRESS SOURCE	REGISTER COUNTER	\overline{OE}
X	=0 ≠0	POP NC	μ PC STACK	NC DECREMENT	PL

Figure 4-21. Repeat loop from stack if counter ≠ 0 (RFCT, 8).

LOOP Repeat Loop until TEST = TRUE

ADDRESS (HEX)	LABEL	2910 INSTR	COND MUX	BRANCH ADDRESS	
50		CONT	#	#	
51		PUSH	FAIL	#	<--- Register/Counter not used
52	BEGIN:	CONT	#	#	
53		CONT	#	#	
54		CONT	#	#	
55		CONT	#	#	
56		LOOP	TESTI	#	<--- Go to TOS
57		CONR	#	#	

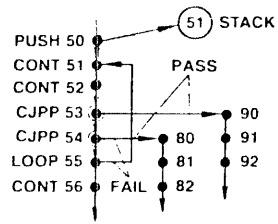


\overline{CC}	COUNTER = 0 LINE	STACK	ADDRESS SOURCE	REGISTER/ COUNTER	\overline{OE}
PASS FAIL	X	POP NC	μ PC STACK	NC	PL

Figure 4-22. Test end of loop (LOOP, D). Must be preceding first statement in loop.

CJPP Conditional Jump to Pipeline and POP TOS
 (Use to exit from a loop which uses the stack)

ADDRESS (HEX)	LABEL	2910 INSTR	COND MUX	BRANCH ADDRESS
50		PUSH	FAIL	#
51		CONT	#	#
52		CONT	#	#
53		CJPP	TESTJ	ADRJ
54		CJPP	TESTK	ADRK
55		LOOP	TESTL	#
56		CONT	#	#
.				
.				
.				
80	ADRK:	CONT	#	#
81		CONT	#	#
82		CONT	#	#
.				
.				
.				
90	ADRJ:	CONT	#	#
91		CONT	#	#
92		CONT	#	#

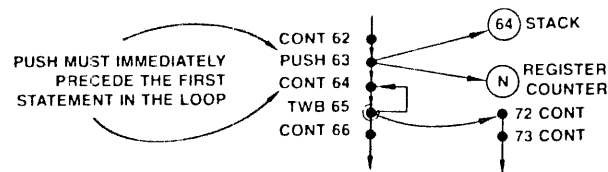


\overline{CC}	COUNTER = 0 LINE	STACK	ADDRESS SOURCE	REGISTER/ COUNTER	\overline{OE}
PASS FAIL	X	POP NC	D μ PC	NC	PL

Figure 4-23. Conditional jump pipeline and POP (CJPP, B).

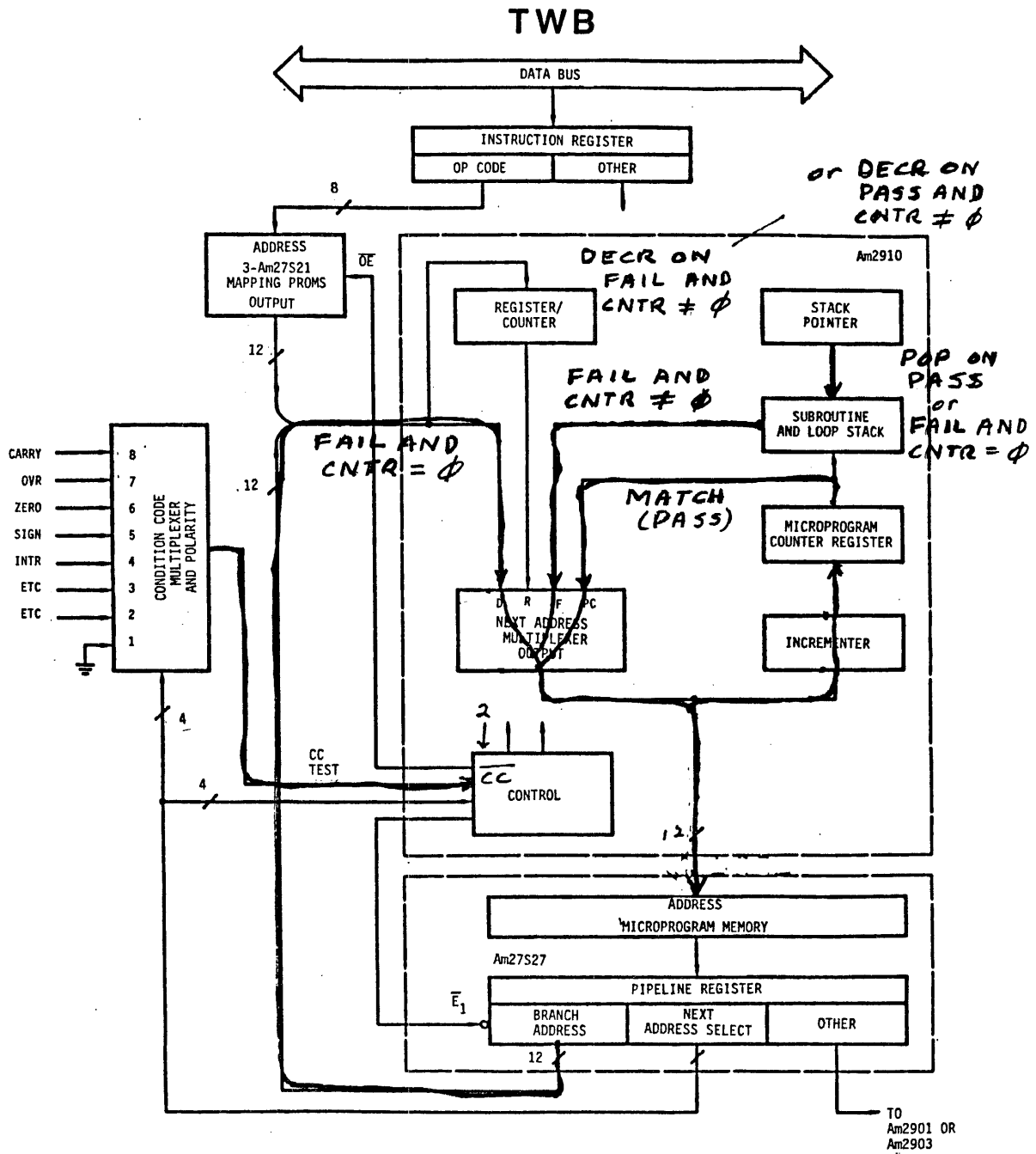
TWB Three-Way Branch (Dead-Man Time-Out)

ADDRESS (HEX)	LABEL	2910 INSTR	COND MUX	BRANCH ADDRESS
62		CONT	#	#
63		PUSH	PASS	VAL-1
64	BEGIN:	CONT	#	#
65		TWB	TESTM	ADRM
66		CONT	#	#
.				
.				
.				
72	ADRM:	CONT	#	#
73		CONT	#	#

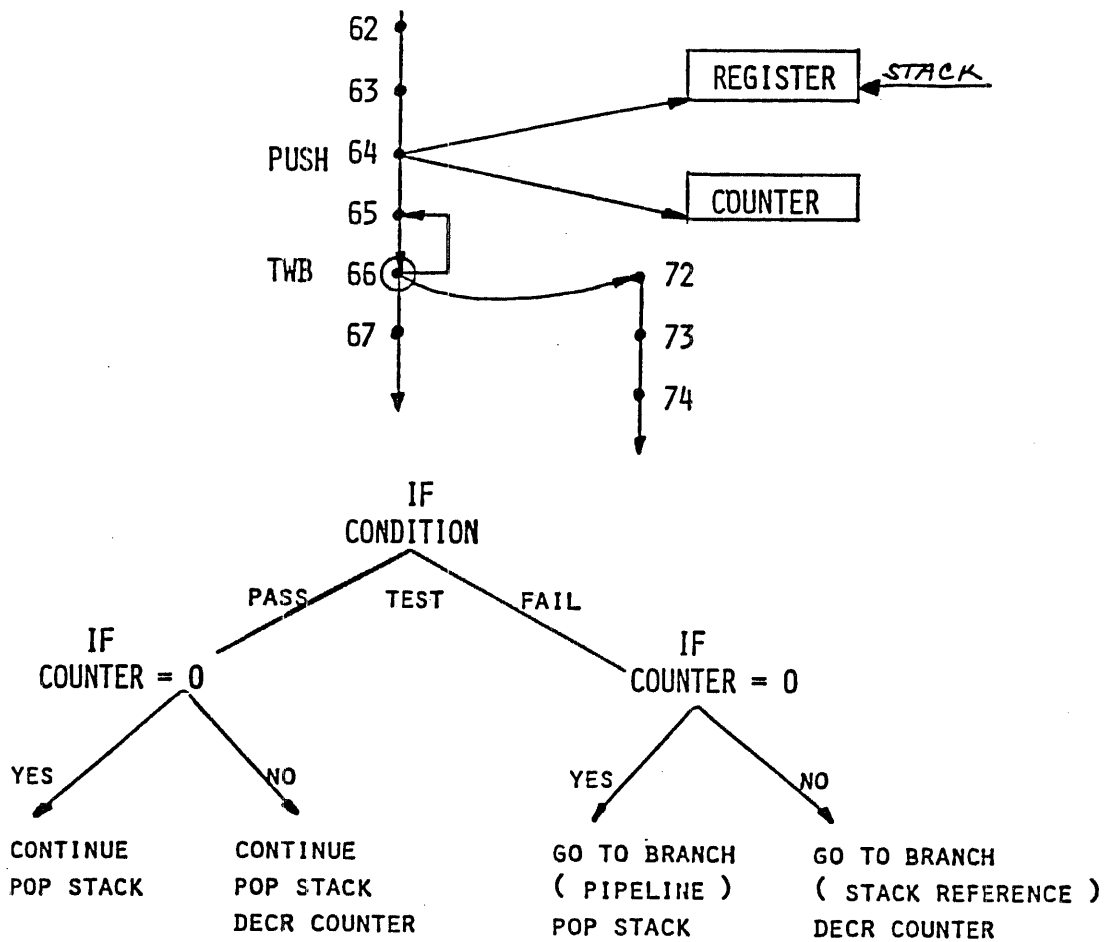


\overline{CC}	COUNTER = 0 LINE	STACK	ADDRESS SOURCE	REGISTER COUNTER	\overline{OE}
PASS	= 0 ≠ 0	POP	μ PC	NC DECREMENT	PL
FAIL	= 0 ≠ 0	POP NC	D STACK	NC DECREMENT	

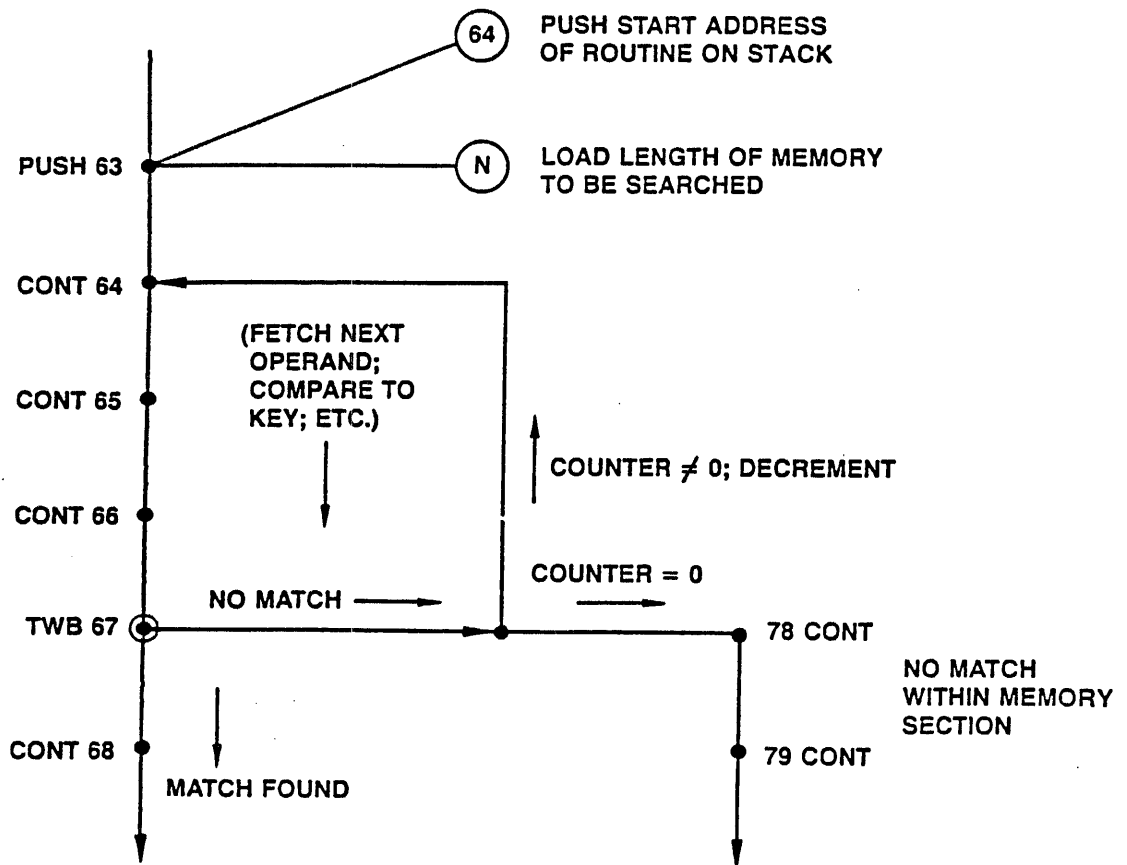
Figure 4-24. Three-way branch (TWB. F).



POWERFUL THREE-WAY BRANCHING



EXAMPLE OF THREE-WAY BRANCH



Special Pins on Am2910

\overline{RLD} Register Load

- For the basic instruction set, \overline{RLD} is held high
- For causing the register to load on the \uparrow clock transition, regardless of the instruction, \overline{RLD} is pulled low -- whatever is on the bus is loaded into the register

C_{IN} Carry In

- For normal operation, C_{IN} is held high
- To repeat an instruction, C_{IN} is driven low (not normally under pipeline control or you may have an infinite loop!)

Special Pins on Am2910

 \overline{CCEN} **Condition Code Enable**

- \overline{CCEN} = LOW; enables \overline{CC} (TEST) input to operate normally
- \overline{CCEN} = HIGH; all conditional instructions are unconditionally true (TEST = PASS)

 \overline{OE} Tri-state control of Y_I outputs **\overline{FULL}**

Five items are on stack; use in diagnostic test programs; debug

1-1840

ED2900A

1-1840

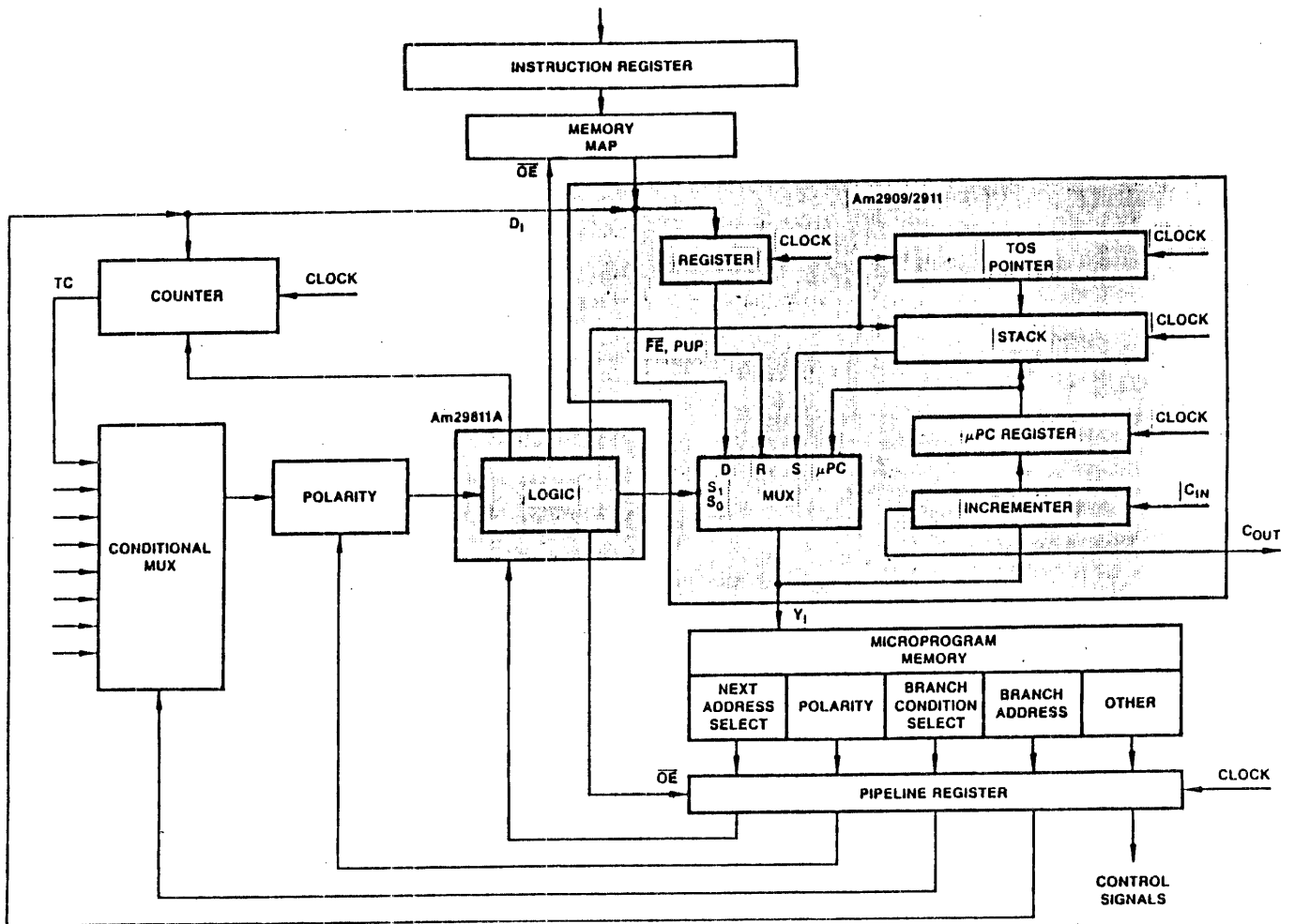
NEXT ADDRESS CONTROL

Am29811

MICROPROGRAMMED SEQUENCERS

Am2909 / Am2911

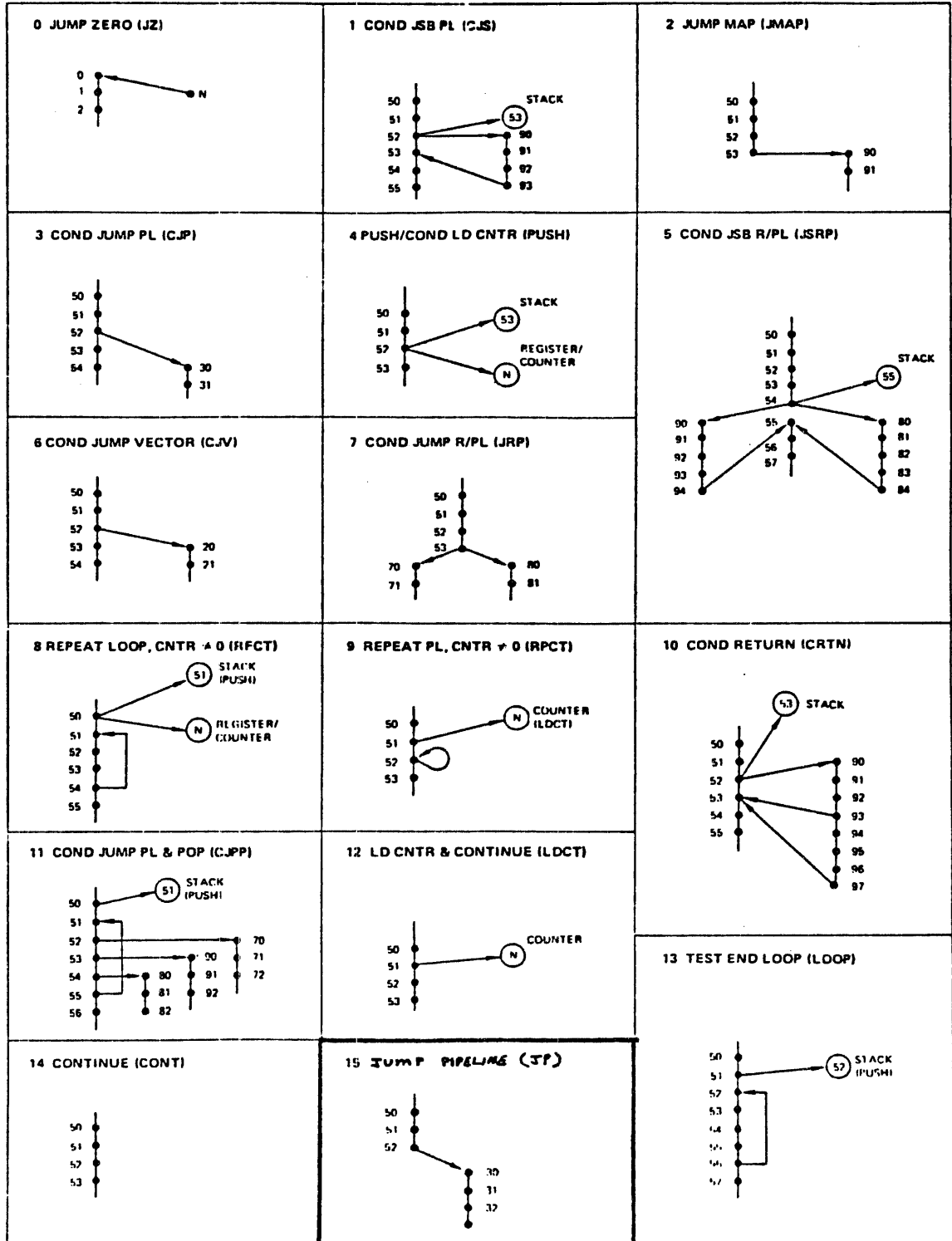
Am2909/2911



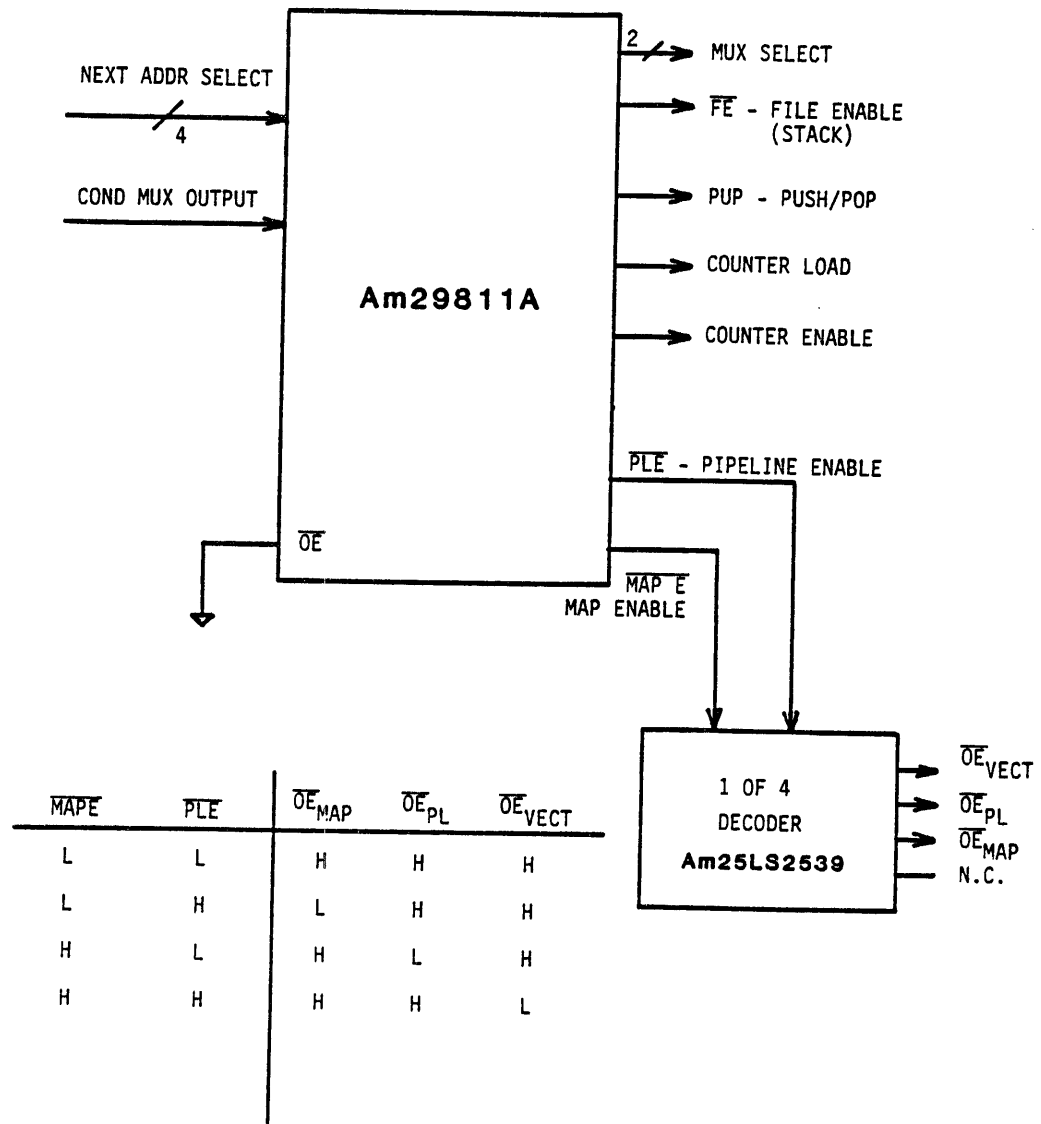
Using the Am29811A with the Am2909A/Am2911A

- Bit-slice architecture means more microword addresses due to more address lines, hence larger microprograms. (Sequencer width independent of ALU width.)
- ORed outputs on Am2909A allows use of Am29803A for 16-way branch.
- Separate register (Ri) and direct (Di) inputs on Am2909A for flexibility.
- Am2909A and Am2911A speeds are comparable to Am2910. (See Data Book)
- Could replace Am29811A with ROM for customer instruction set.

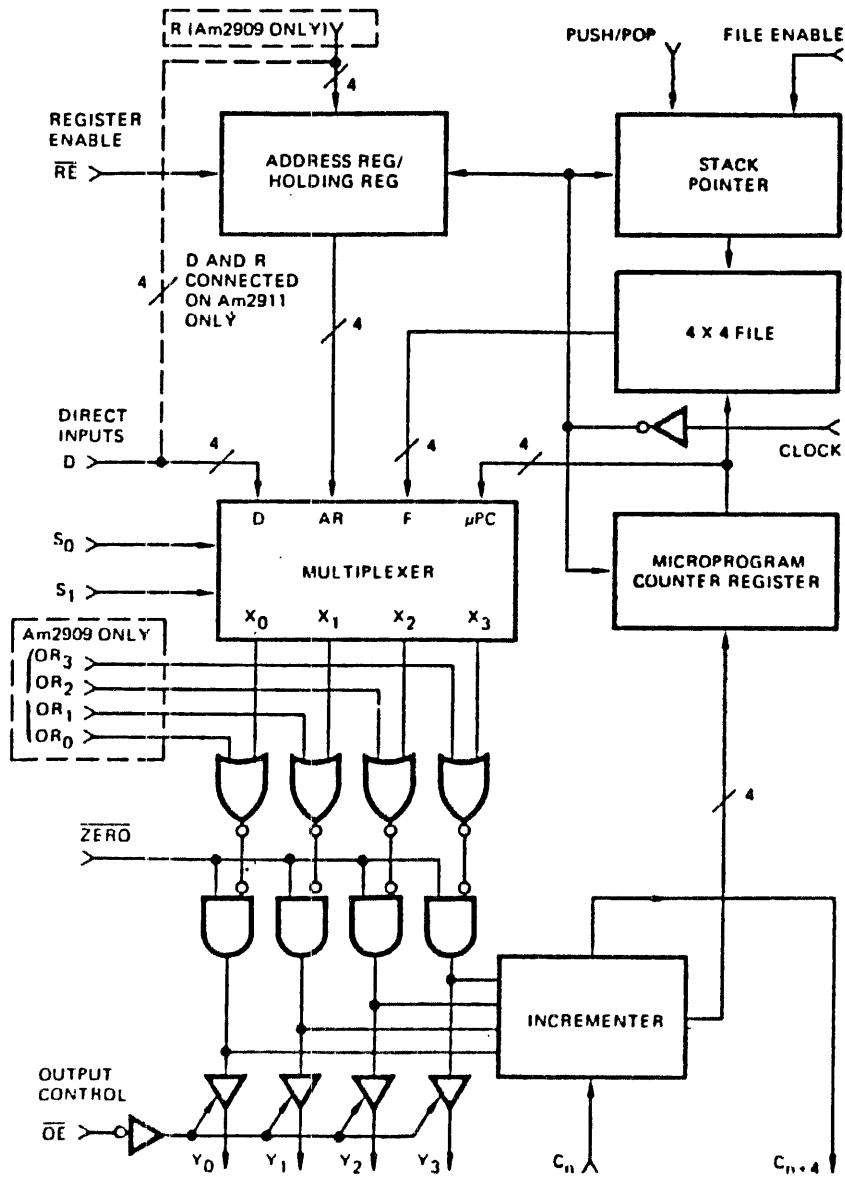
Am29811



SUMMARY OF NEXT ADDRESS CONTROL LOGIC BLOCK



Microprogram Sequencer Block Diagram



<u>2911</u>	<u>2909</u>	<u>2910</u>	<u>29112</u>
4 bit	4 bit	12 bit	8 bit
shared $R_i D_i$	separate $R_i D_i$	shared	separate
none	OR_i input for 29803	none	16-way branch
$\overline{RE} = \text{LOW}$ loads reg	$\overline{RE} = \text{LOW}$ loads reg	$\overline{RLD} = \text{LOW}$ loads reg	N/A
ZERO = LOW $Y_i = \emptyset$	ZERO = LOW $Y_i = \emptyset$	none	CZIO
\overline{OE}	\overline{OE}	\overline{OE}	HOLD
needs Am29811	needs Am29811	self contained	self contained
JP	JP	TWB	TWB+
\overline{OE}_{PL}	\overline{OE}_{PL}	\overline{OE}_{PL}	MINTA
\overline{OE}_{MAP}	\overline{OE}_{MAP}	\overline{OE}_{MAP}	
		\overline{OE}_{VECT}	
20pin DIP	28pin DIP	40pin DIP	48pin DIP

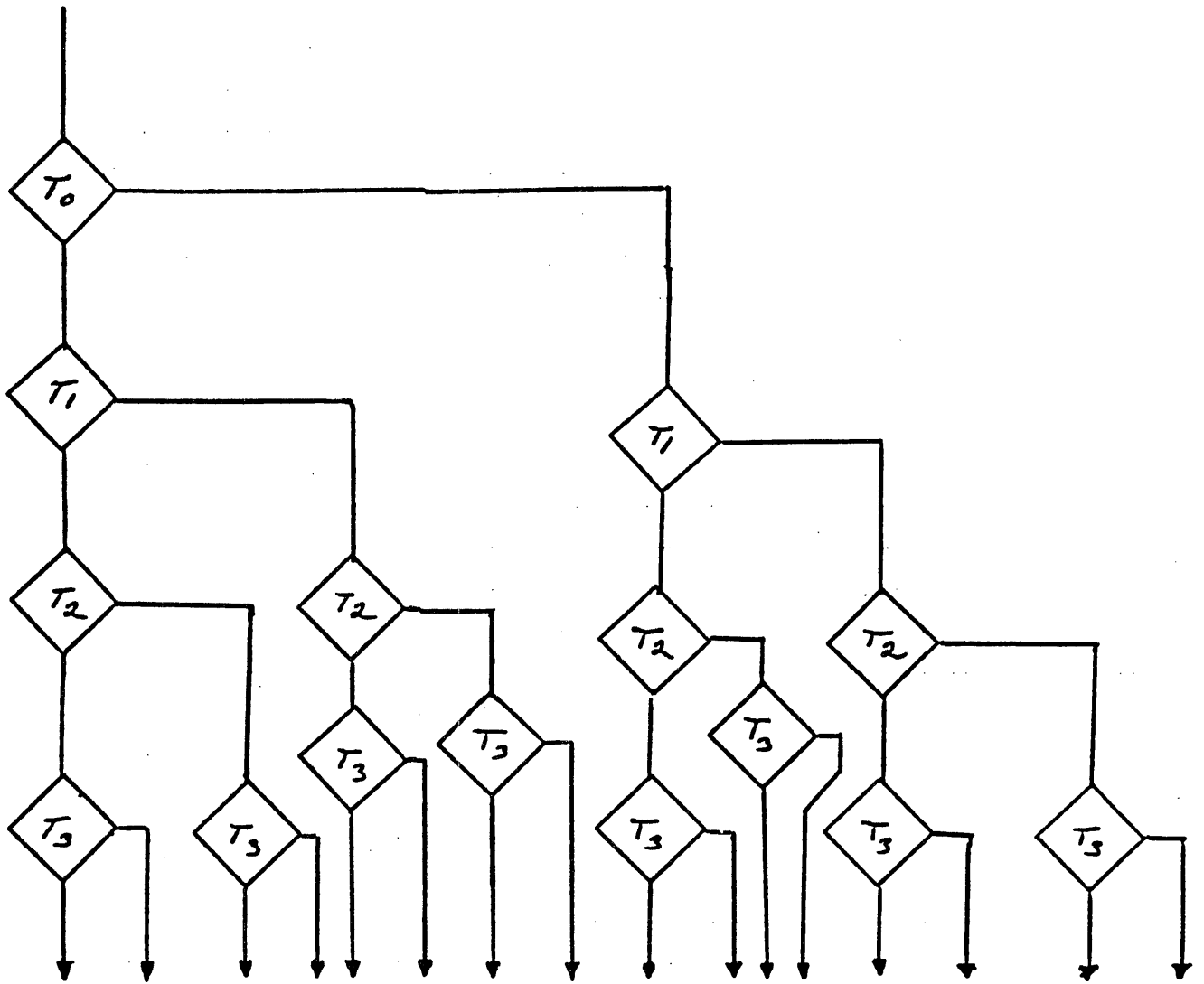
Am29803A

- There is another statement that can be used in structured code

THE CASE STATEMENT

- An N-way conditional branch
- Used for choosing 1 of n paths based on one or more test results.
- For the Am29803A, 1 of 16 branches can be selected.

16-Way Branch
(T_3, T_2, T_1, T_0)



Advantages of the Am29803A

- Allows any combination of up to four tests (16-way branch) to be decoded in two microcycles.
- Faster than a series of conditional jumps and tests written in microcode.
- Easier for microprogramming.

Am29803A FUNCTION TABLE

Function	I ₃	I ₂	I ₁	I ₀	T ₃	T ₂	T ₁	T ₀	OR ₃	OR ₂	OR ₁	OR ₀
No Test	L	L	L	L	X	X	X	X	L	L	L	L
Test T ₀	L	L	L	H	X	X	X	L	L	L	L	L
Test T ₁	L	L	H	L	X	X	L	X	L	L	L	L
Test T ₀ & T ₁	L	L	H	H	X	X	L	L	L	L	L	L
Test T ₂	L	H	L	L	X	L	X	X	L	L	L	L
Test T ₀ & T ₂	L	H	L	H	X	L	X	L	L	L	L	L
Test T ₁ & T ₂	L	H	H	L	X	L	L	X	L	L	L	L
Test T ₀ , T ₁ & T ₂	L	H	H	H	X	L	L	L	L	L	L	L
Test T ₃	H	L	L	L	L	X	X	X	L	L	L	L
Test T ₀ & T ₃	H	L	L	H	L	X	X	L	L	L	L	L
Test T ₁ & T ₃	H	L	H	L	L	X	L	X	L	L	L	L
Test T ₀ , T ₁ & T ₃	H	L	H	H	L	X	L	L	L	L	L	L
Test T ₂ & T ₃	H	H	L	L	L	L	X	X	L	L	L	L
Test T ₀ , T ₂ & T ₃	H	H	L	H	L	L	X	L	L	L	L	L
Test T ₁ , T ₂ & T ₃	H	H	H	L	L	L	L	X	L	L	L	L
Test T ₀ , T ₁ , T ₂ & T ₃	H	H	H	H	L	L	L	L	L	L	L	L

L = LOW, H = HIGH, X = Don't care

EXAMPLE

- Show the microcode (partial width only) to program these statements, assuming an Am2903-Am2909/11-Am29811 CCU.

```

IF A THEN ON (T2T0) GO TO (10, 200, 30, 40)*
      ELSE ON (T3T1) GO TO (20, 200, 10, 20)
IF B THEN ON (T3T2T1) GO TO (10, 20, 30, 40,..)
      ELSE ON (T2T1T0) GO TO (100, 200, 300,...)

```

Where:

A and B are condition multiplexer input lines.

T3, T2, T1, T0 are test inputs to the Am29803.

10, 20, 200, etc. are labels of statements.

The same label means the same statement.

The statements may be considered to be the beginning of a microroutine of unknown length.

```

* IF A IS TRUE, THEN
      IF (T2T0 = 00) GO TO 10
      IF (T2T0 = 01) GO TO 200
      IF (T2T0 = 10) GO TO 30
      IF (T2T0 = 11) GO TO 40

```

Am29803 SOLUTION

1.	LABEL/ ADDR	29811 INSTR	MUX SEL	BR ADDR	29803 INSTR	
	i	CJP	A	i + 2	NO TEST	
	i + 1	JP	#	350*	T ₃ T ₁	
	i + 2	JP	#	360*	T ₂ T ₀	* address must have 0 as final HEX digit if LSS 2909 attached to 29803
	j	CJP	B	j + 2	NO TEST	
	j + 1	JP	#	370	T ₂ T ₁ T ₀	
	j + 2	JP	#	380	T ₃ T ₂ T ₁	
	350	JP	#	20	NO TEST	
	351	JP	#	200	NO TEST	
	352	JP	#	10	NO TEST	
	353	JP	#	20	NO TEST	
	360	JP	#	10	NO TEST	
	361	JP	#	200	NO TEST	
	362	JP	#	30	NO TEST	
	363	JP	#	40	NO TEST	
	370	JP	#	100	NO TEST	
	371	JP	#	200	NO TEST	
	.					
	.					
	.					

1-1990

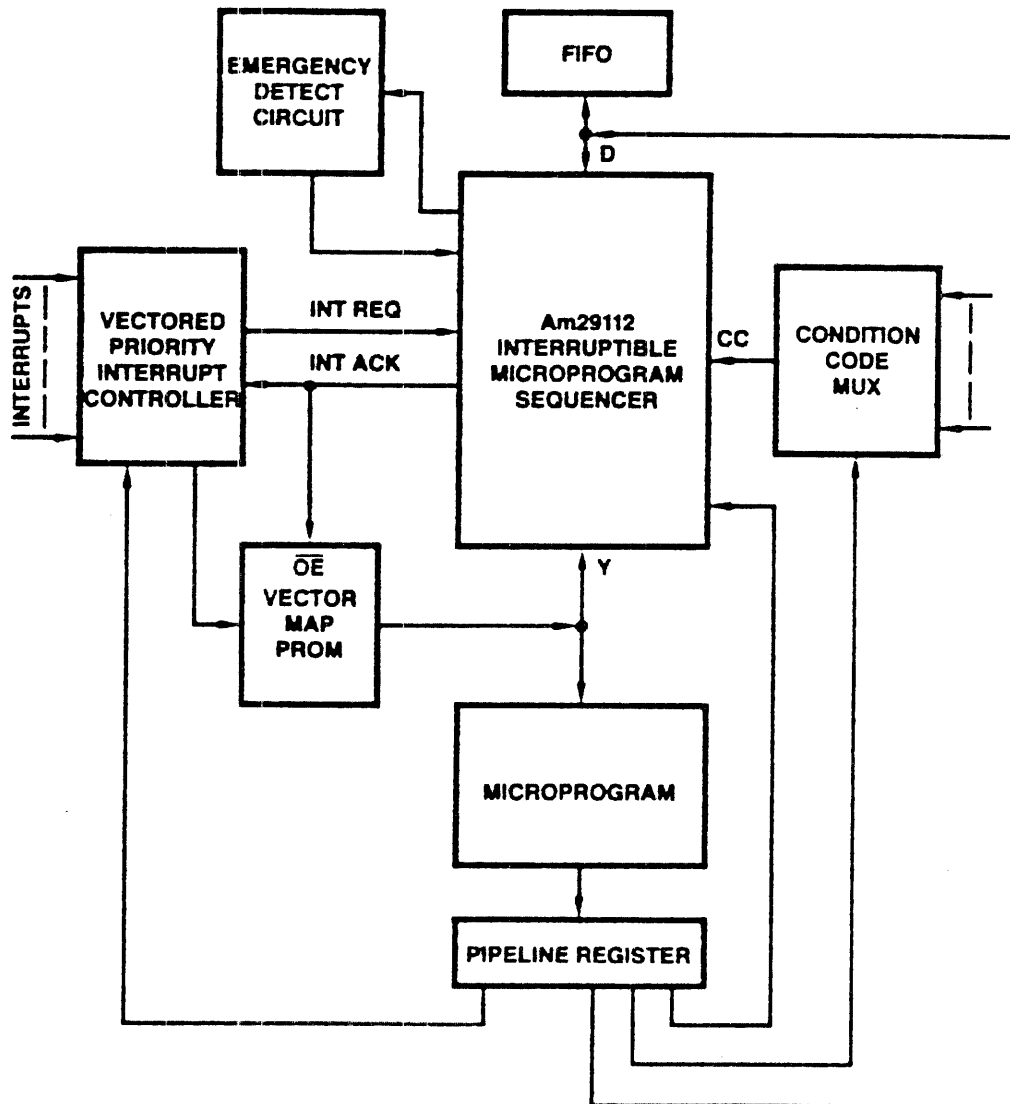
ED2900A

1-1990

Am29112

MICROPROGRAM SEQUENCER

Am29112 in a Single Pipelined System



Am29112 CHARACTERISTICS

Functional Description

- The Am29112 is a high performance interruptible microprogram controller intended for use in very high speed microprogrammed machines and optimized for the new state-of-the-art ALU's and other processing components.
- It has an instruction set featuring relative and multiway branching, a rich variety of looping constructs, and provision for loading and unloading the on-chip stack.
- Interrupts are accepted at the microcycle level and serviced in a manner completely transparent to the interrupted microcode.

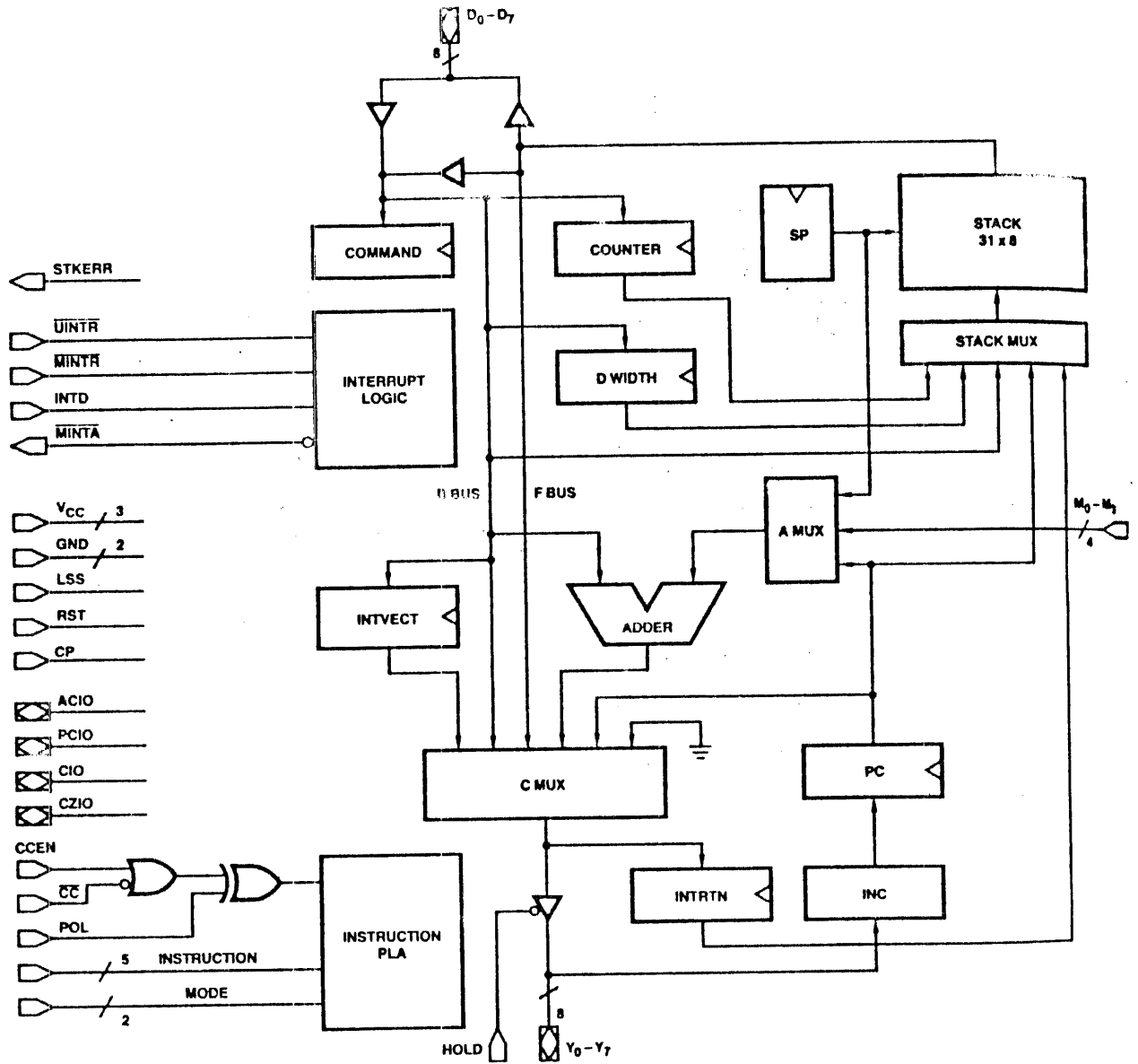
DISTINCTIVE CHARACTERISTICS:

- The Am29112 is designed to operate in 10 MHz microprogrammed systems.
- A single Am29112 is 8 bits wide and addresses up to 256 words of microprogram memory. Two Am29112's may be cascaded to directly address up to 64K of microprogram memory.
- A 33 register deep on-chip stack is used for subroutine linkage, interrupt handling and loop control.
- Two kinds of interrupts: maskable and unmaskable.
- Features an 8-bit counter for loop control. When two Am29112s are cascaded, the counters can act as a single 16-bit counter or two independent 8-bit counters.
- Features direct, multiway, multiway relative and program counter relative addressing.
- Support for writable control store.
- Hold feature - a hold pin facilitates multiple sequencer implementations.

Am29112 OVERVIEW:

- The Am29112 is designed for use in single-level pipelined systems. A typical configuration is shown on the next page.
- Branch addresses, constants for the various registers and stack pointer values are supplied to the Am29112 through the D port which is bidirectional to allow the stack to be unloaded onto an external LIFO.
- The next address generated by the sequencer is output on the Y port and directly drives the micromemory program.
- A single register at the output of the microprogram memory contains the microinstruction being executed, while the next is being fetched.
- External conditions are applied to the CC input of the Am29112 via the condition code MUX and also to the multiway inputs.

Am29112 Configuration



Am29112 OVERVIEW (cont'd):

- A vectored, priority-interrupt controller generates a prioritized interrupt request (MINTR) to the Am29112, which acknowledges the request via the MINTA pin. Upon receiving the acknowledge, the priority-interrupt control puts out the encoded vector from the mapping PROM. The MINTA output of the Am29112 turns on the PROM output and simultaneously turns off the Y port, enabling the interrupt vector onto the microprogram address bus. In the Am29112, internal states are automatically saved on the stack while the interrupt vector is transmitted through the Y port and incremented to form the next microprogram address.

- The emergency detect circuit generates an unmaskable interrupt request upon power failure or stack error. On receiving an unmaskable interrupt, the sequencer branches to the unmaskable interrupt routine; the address of this routine is stored on the Am29112 in the INTVECT register.

- The internal organization of the Am29112 is shown in the figure. The most important control loop inside the sequencer consists of the CMUX, incrementer, and PC register.

Am29112 OVERVIEW (cont'd)

- The CMUX selects the next microprogram address based on the instruction and condition code inputs. The next microprogram address is selected from: the PC register for a continue, the D port for a branch, the adder for relative and multiway branches, the interrupt register for unmaskable interrupts, the stack for subroutine returns or loop repeats, or all zeros for the JUMP ZERO instruction.

Am29112 INSTRUCTION SET

MODE BITS

- The Am29112 is controlled by five instruction inputs, two mode inputs, and the condition code. In typical applications it is expected that the instruction inputs are driven directly from the pipeline, whereas the mode inputs are either permanently wired high or low to select the desired operating mode, or driven indirectly via external logic. (In some applications it might be justifiable to drive the mode bits directly from the pipeline.) The two mode bits select among three operating modes: normal (0,0), extended (01) and forced continue (10 and 11). In the normal mode the entire instruction set of the Am29112 applies.

MODE CONTROLS

I_{65}	Mode	Description
00	Normal	For cascaded Am29112s, two independent 8-bit counters
01	Extended	For cascaded Am29112s, one 16-bit counter
10	Forced	The Am29112 executes a continue instruction regardless of instruction, condition code, and multiway inputs.
11	Continue	

Extended Mode:

- The instruction set includes the instructions that differentiate between upper and lower counters (when there are two cascaded Am29112s). In the normal mode, the two counters on cascaded Am29112s function independently.

- In the extended mode, however, the counters on cascaded Am29112s behave like one 16-bit counter and instructions that differentiate between counters degenerate into identical instructions.

- The instructions of the Am29112 are classified into four groups:
 - branching and subroutine linkage
 - looping
 - stack and register
 - interrupt

- The sequencer has a repertoire of 40 different instructions. In order to encode these instructions with only five instruction lines, the condition code is used in some cases to differentiate between two distinct instructions sharing the same opcode.

Am29112 INSTRUCTION SET

Opcode (I ₄₀)	Condition	Mnemonic	Description
0	X	JZ.U	UNCONDITIONAL JUMP ZERO
1	PASS	PUSHD.P	PUSH D (PASS)
1	FAIL	LDCMD.F	LOAD COMMAND REGISTER FROM D (FAIL)
2	COND	POP.C	POP; CONDITIONAL STACKOUT TO D
3	COND	CJD.C	CONDITIONAL JUMP D
4	COND	CJSD.C	CONDITIONAL JUMP SUBROUTINE D
5	COND	CJMW.C	CONDITIONAL JUMP MULTIWAY D
6	COND	CJSMW.C	CONDITIONAL JUMP SUBROUTINE MULTIWAY D
7	COND	CRTN.C	CONDITIONAL RETURN
8	COND	PUSHPL.C	PUSH PC: COND LOAD LOWER COUNTER
9	COND	LDLC.C	LOAD LOWER COUNTER; COND PUSH COUNTER
10	PASS	POPLC.P	POP TO LOWER COUNTER (PASS)
11	PASS	RSTSP.P	RESET STACK POINTER (PASS)
11	FAIL	LDINTV.F	LOAD UNMASKABLE INTERRUPT VECTOR (FAIL)
12*	PASS	RFCTU.P	REPEAT LOOP, UPPER COUNTER = 0 (PASS)
12*	FAIL	RFCTL.F	REPEAT LOOP, LOWER COUNTER = 0 (FAIL)
13**	PASS	RPCTU.P	REPEAT PIPELINE, UPPER COUNTER = 0 (PASS)
13**	FAIL	RPCTL.F	REPEAT PIPELINE, LOWER COUNTER = 0 (FAIL)
14	COND	LOOP.C	TEST END LOOP
15	PASS	ENINT.P	ENABLE INTERRUPTS (PASS)
15	FAIL	DISINT.F	DISABLE INTERRUPTS (FAIL)
16***	COND	TWBL.C	THREE-WAY BRANCH, LOWER COUNTER
17***	COND	TWBU.C	THREE-WAY BRANCH, UPPER COUNTER
18	PASS	TSTSP.P	TEST SP WITH D (PASS)
18	FAIL	TSTMT.F	JUMP D IF STACK NOT EMPTY
19	COND	CJDF.C	COND JUMP D/STACK AND POP
20	COND	CJSDF.C	COND JUMP SUBROUTINE D/STACK AND POP
21	COND	CJMWR.C	COND JUMP MULTIWAY RELATIVE D
22	COND	CJSMWR.C	COND JUMP SUBROUTINE MULTIWAY RELATIVE D
23	COND	CJPP.C	COND JUMP PIPELINE AND POP
24	COND	PUSHPU.C	PUSH PC: COND LOAD UPPER COUNTER
25	COND	LDUC.C	LOAD UPPER COUNTER; COND PUSH COUNTER
26	PASS	POPUC.P	POP TO UPPER COUNTER (PASS)
26	FAIL	POPDW.F	POP TO DISPLACEMENT WIDTH (FAIL)
27	COND	LDDW.C	LOAD DISPLACEMENT WIDTH; COND PUSH DW
28	COND	CJR.C	COND JUMP D PC REL
29	COND	CJRN.C	COND JUMP D PC REL NEGATIVE
30	COND	CJSR.C	COND JUMP SUBROUTINE D PC REL
31	COND	CJSRN.C	COND JUMP SUBROUTINE D PC REL NEGATIVE

*These instructions are identical in the extended mode.

**These too.

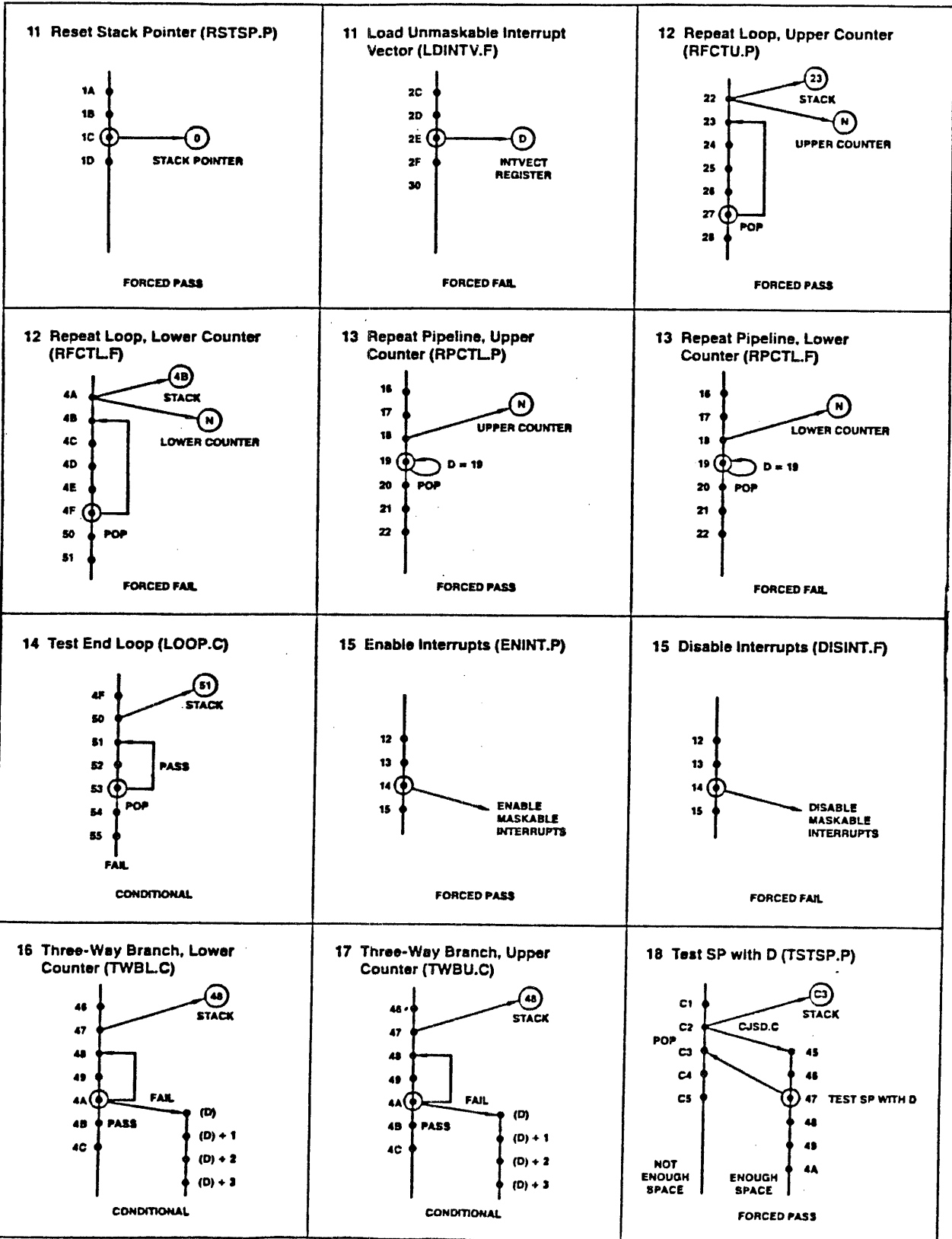
***These too.

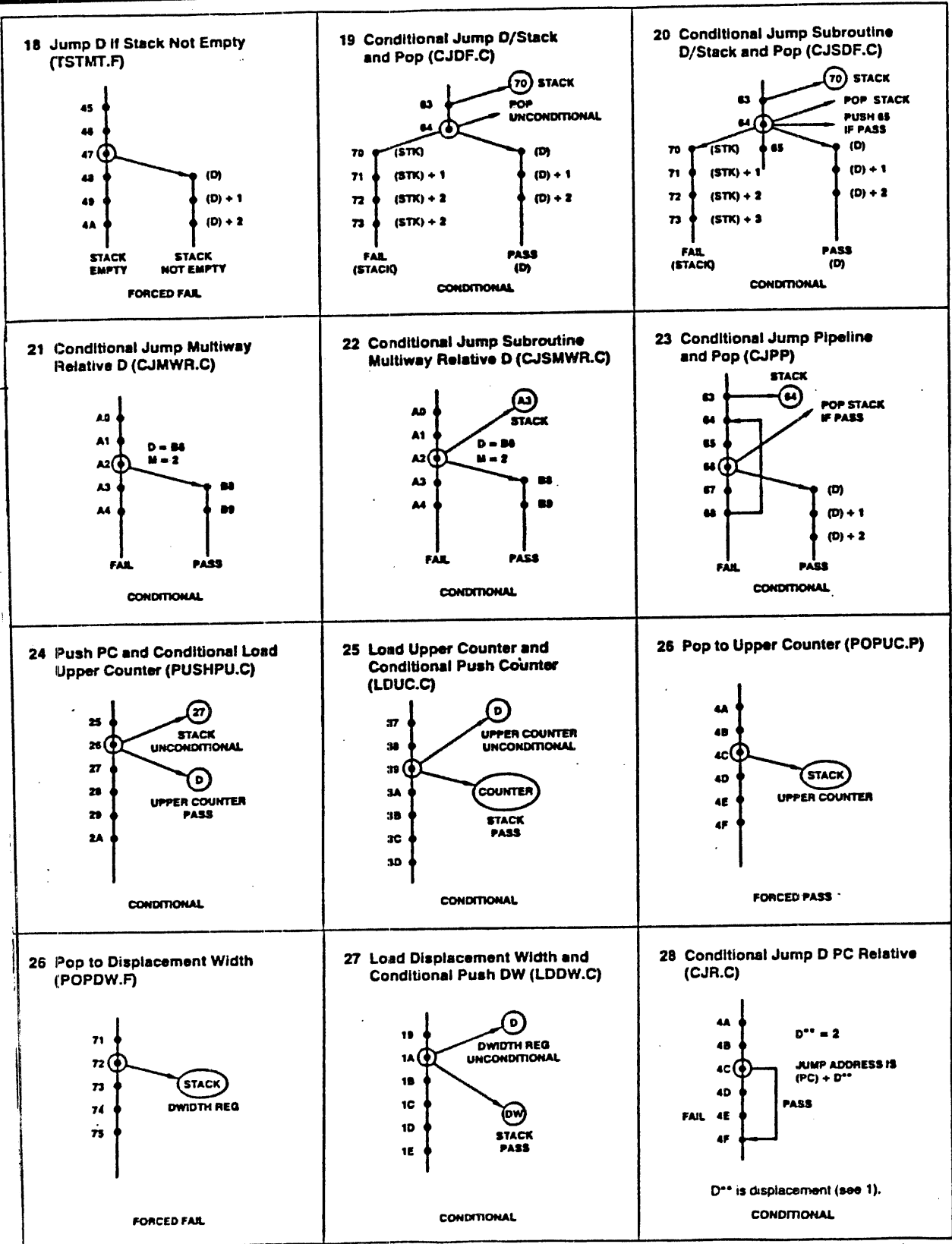
Extensions: U – unconditional; C – conditional; P – PASS condition; F – FAIL condition.

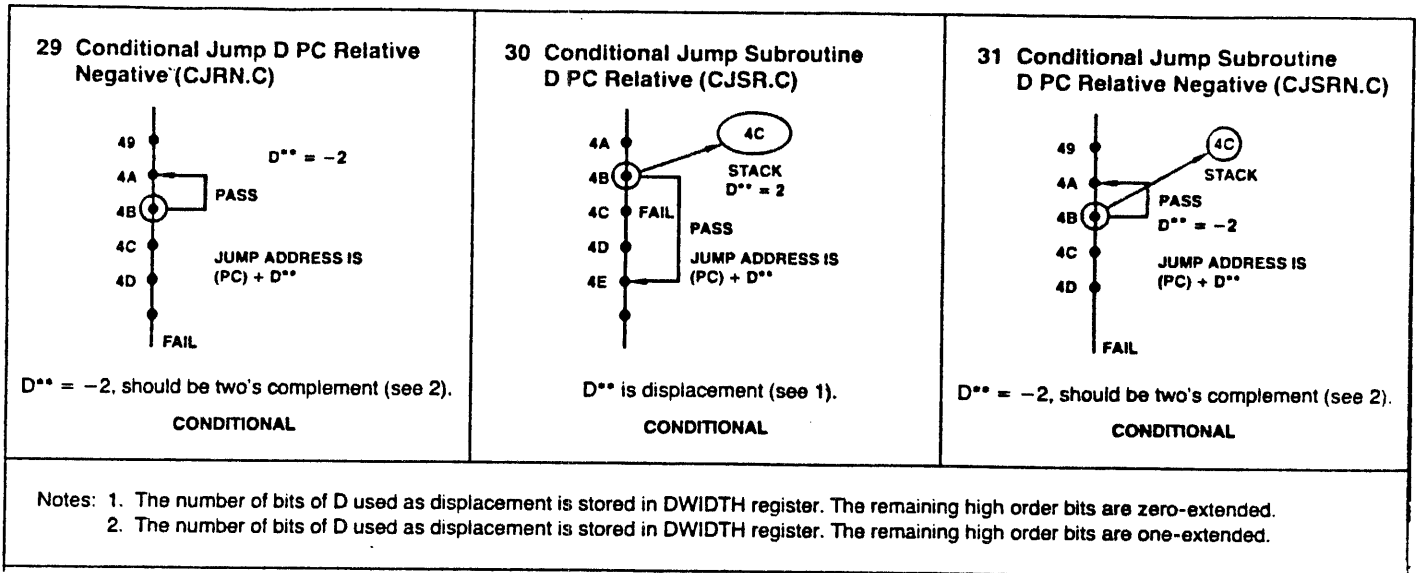
Note: PASS/FAIL condition can be produced as follows. P stands for polarity and I for input.

CC	CCEN	POL	Condition
X	1	0	PASS
X	1	1	FAIL
I	0	P	COND

<p>0 Jump Zero (JZ.U)</p> <p>UNCONDITIONAL</p>	<p>1 Push D (PUSHD.P)</p> <p>FORCED PASS</p>	<p>1 Load Command Latch from D (LDCMD.F)</p> <p>FORCED FAIL</p>
<p>2 Pop and Unconditional Stackout to D (POP.C)</p> <p>CONDITIONAL</p>	<p>3 Jump D (CJD.C)</p> <p>CONDITIONAL</p>	<p>4 Jump Subroutine D (CJSD.C)</p> <p>CONDITIONAL</p>
<p>5 Jump Multiway D (CJMW.C)</p> <p>CONDITIONAL</p>	<p>6 Jump Subroutine Multiway D (CJSMW.C)</p> <p>CONDITIONAL</p>	<p>7 Return (CRTN.C)</p> <p>CONDITIONAL</p>
<p>8 Push PC and Conditional Load Lower Counter (PUSHPLC)</p> <p>CONDITIONAL</p>	<p>9 Load Lower Counter and Conditional Push Counter (LDLC.C)</p> <p>CONDITIONAL</p>	<p>10 Pop to Lower Counter (POPLC.P)</p> <p>FORCED PASS</p>







1-2140

ED2900A

1-2140

HOMEWORK - Am2910

- Turn to your Am2900A Exercise and Laboratory Manual.
Find the exercises for the Am2910 and perform exercises 1 through 18 inclusive.
- For homework, do the famous Coffee Machine problem in ED2900 Exercise and Laboratory Manual.

DESIGN EXAMPLE:

- Solve the advanced traffic light problem using Boolean logic and the state diagram design approach. See ED2900 Exercise and Laboratory Manual.

EVALUATION BOARD EXERCISE

- Read Am29203 Evaluation Board description in ED2900A Exercise and Laboratory manual.
- Perform (Day 2) Am2910 sequencer laboratory experiments.

HOMEWORK DESIGN PROBLEM:

THE FAMOUS COFFEE MACHINE

(See ED2900A Exercise and Laboratory Manual)

MICROCYCLE TIMING - Am2910

CCU MICROCYCLE TIMING

- The objective is to determine the minimum clock period possible for a given design yielding maximum execution speed.
- Each system design is different, requiring detailed analysis.
- Always use maximum (guaranteed, worst-case) delay times and set-up times from the data sheet for the specific system component.
- The basic technique is straightforward:
 - find all possible paths from one register to another
 - calculate the path delay time using worst-case device times
 - the longest path determines the minimum clock period
 - if necessary, look for design changes to reduce the time delay on the longest path
 - alternately, use a variable-length clock to accommodate the longer delays when needed
- The timing analysis approach is learned by considering examples for the CCU using the Am2910. In addition, a similar analysis would be performed for the ALU and other system circuits and devices.

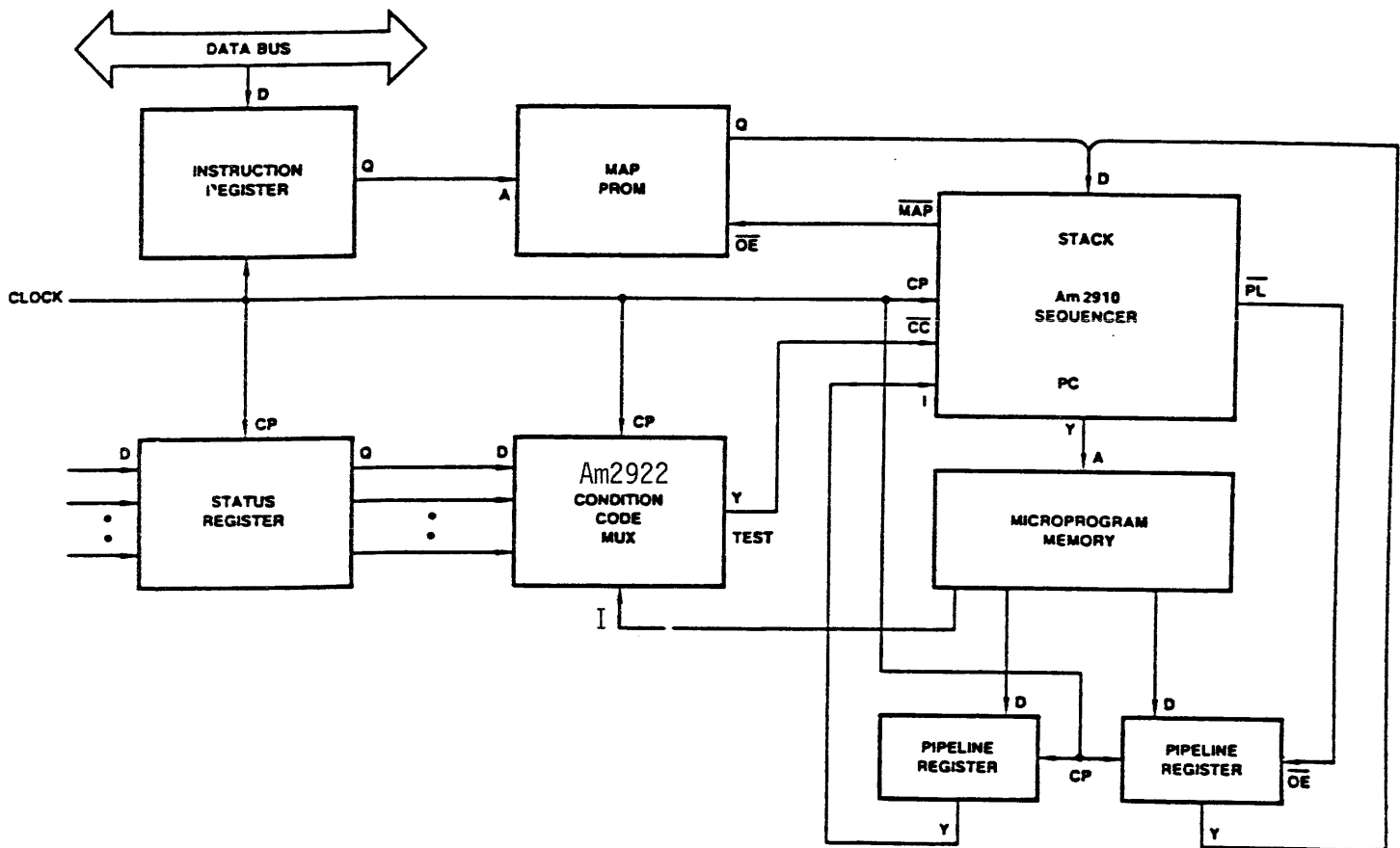
MICROCYCLE TIMING (CONT'D):

- Use the AMD Data Book for all Am2900 parts.
- Data for the non-Am2900 parts is assumed. (For a real design, use the data sheets!)
- For the IR, status register, and pipeline register assume Schottky technology. Also shown are delays for the mapping PROM and the microprogram PROM.

DEVICE	MIN	TYP	MAX
Schottky Register			
clock-to-output		9	15
OE-to-output		13	20
data-set-up-time	5	2	
Mapping PROM			
address-to-output		25	45
OE-to-output		15	20
Microprogram PROM			
address-to-output		30	50
OE-to-output		18	25

MICROCYCLE TIMING (CONT'D):

- The architecture to be used in these examples is the typical computer CCU.
- Although the ALU is not shown, a similar timing analysis must be conducted for its paths for a complete design.
- Note that the Am2922 multiplexer includes a latch on its input (I) lines that makes up part of the pipeline register. This allows a smaller overall part count.
- Observe from the Data Book for the Am2910 that different instructions have different delay times. This means that each involved path has to be calculated for all possible instructions.
- The potentially huge numbers of timing paths will, in practice, be reduced by experience.
- In addition to timing path diagrams, PERT charts are employed to find the longest path.



Am2910-1 SWITCHING CHARACTERISTICS

The tables below define the Am2910-1 switching characteristics. Tables A are setup and hold times relative to the clock LOW-to-HIGH transition. Tables B are combinational delays. Tables C are clock requirements. All measurements are made at 1.5V with input levels at 0 or 3V. All values are in ns. All outputs have maximum DC loading.

I. GUARANTEED CHARACTERISTICS OVER COMMERCIAL OPERATING RANGE

Am2910-1DC ($T_A = 0$ to $+70^\circ\text{C}$, $V_{CC} = 4.75$ to 5.25V , $C_L = 50\text{pF}$)

A. Set-up and Hold Times

Input	t_s	t_h
D \rightarrow R	24	6
D _i \rightarrow PC	58	4
I ₀₋₁₃	75	0
\overline{CC}	63	0
\overline{CCEN}	63	0
CI	46	5
\overline{RLD}	36	6

B. Combinational Delays

Input	Y	PL. VECT. MAP	Full
D _{0-D₁₁}	20	-	-
I ₀₋₁₃	50	51	-
\overline{CC}	30	-	-
\overline{CCEN}	30	-	-
CP (Note 2)	75	-	60
I = 8, 9, 15	85	-	60
CP All other I	55	-	60
\overline{OE} (Note 3)	35, 30	-	-

C. Clock Requirements (Note 1)

Minimum Clock LOW Time	50	ns
Minimum Clock HIGH Time	35	ns
Minimum Clock Period, I = 8, 9, 15 (Note 2)	113	ns
	123	ns
Minimum Clock Period, I = 14	93	ns

Boldface times indicate speed selected critical paths.

II. GUARANTEED CHARACTERISTICS OVER MILITARY OPERATING RANGE

Am2910-1DM ($T_C = -55$ to $+125^\circ\text{C}$, $V_{CC} = 4.5$ to 5.5V , $C_L = 50\text{pF}$)

A. Set-up and Hold Times

Input	t_s	t_h
D _i \rightarrow R	28	6
D _i \rightarrow PC	62	4
I ₀₋₁₃	81	0
\overline{CC}	65	0
\overline{CCEN}	63	0
CI	58	5
\overline{RLD}	42	6

B. Combinational Delays

Input	Y	PL. VECT. MAP	Full
D _{0-D₁₁}	25	-	-
I ₀₋₁₃	54	58	-
\overline{CC}	35	-	-
\overline{CCEN}	37	-	-
CP (Note 2)	77	-	67
I = 8, 9, 15	98	-	67
CP All other I	61	-	67
\overline{OE} (Note 3)	40, 30	-	-

C. Clock Requirements (Note 1)

Minimum Clock LOW Time	58	ns
Minimum Clock HIGH Time	42	ns
Minimum Clock Period, I = 8, 9, 15 (Note 2)	114	ns
	125	ns
Minimum Clock Period, I = 14	100	ns

NOTES:

1. Clock periods for instructions not specified are determined by external conditions.
2. These instructions are conditional on the counter. Use the shorter specified delay times if the previous instruction could produce no

change in the counter or could only decrement the counter. Use the longer delays from CP to outputs if the instruction prior to the clock was 4 or 12 or \overline{RLD} was LOW.

3. Enable/Disable. Disable times measured to 0.5V change on output voltage level with $C_L = 5.0\text{pF}$.

CONTINUE INSTRUCTION TIMING ANALYSIS

- Locate "all" register-to-register timing paths.
 - start at pipeline, CP -> output
 - Am2910 I->Y, CP->Y and I->PC-setup in parallel
 - after Am2910 output is stable, add micromemory address -> output delay
 - finally, setup for pipeline and Am2922

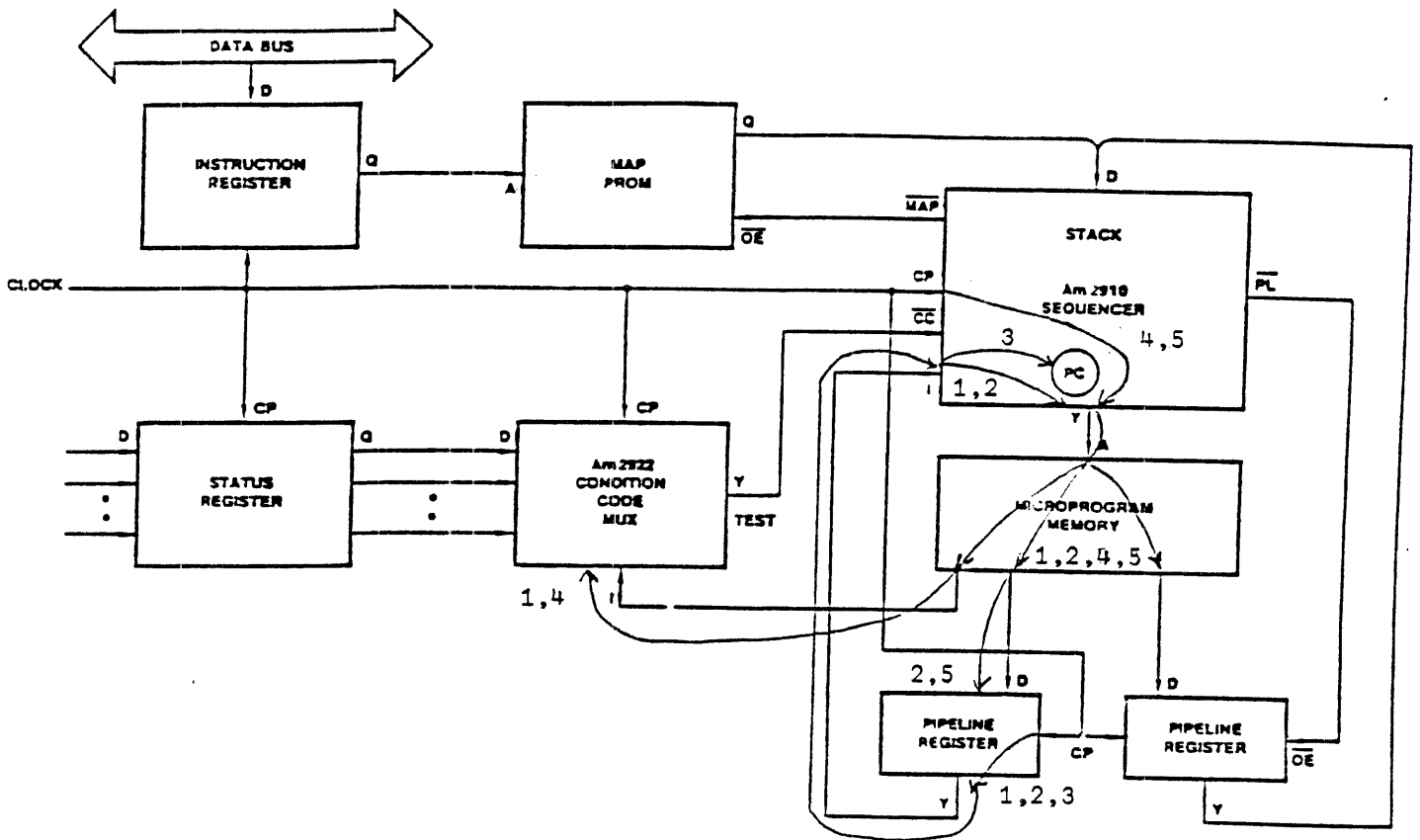
- On **PERT chart**, assign worst-case times to each block.

- Add up times along each path.

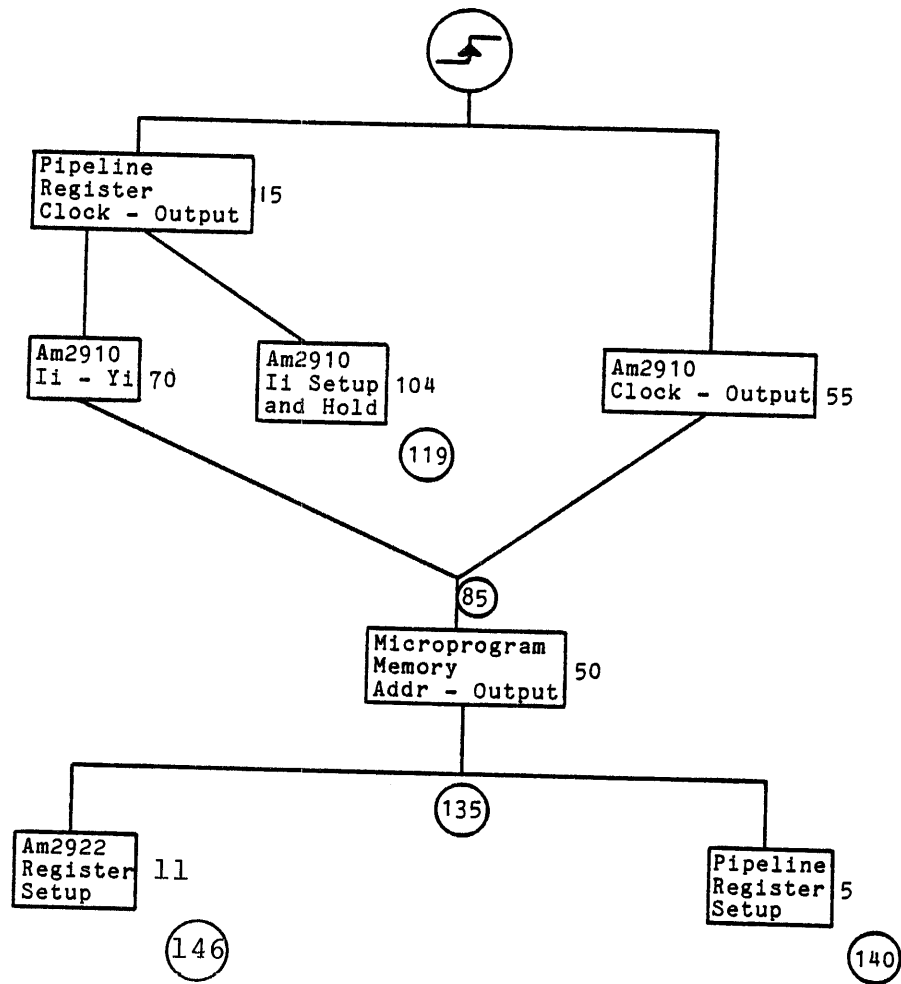
- For PERT chart, converging paths must all be satisfied, hence use maximum time at that point (e.g. address input to micromemory).

- Maximum path defines minimum clock cycle possible.

CONTINUE



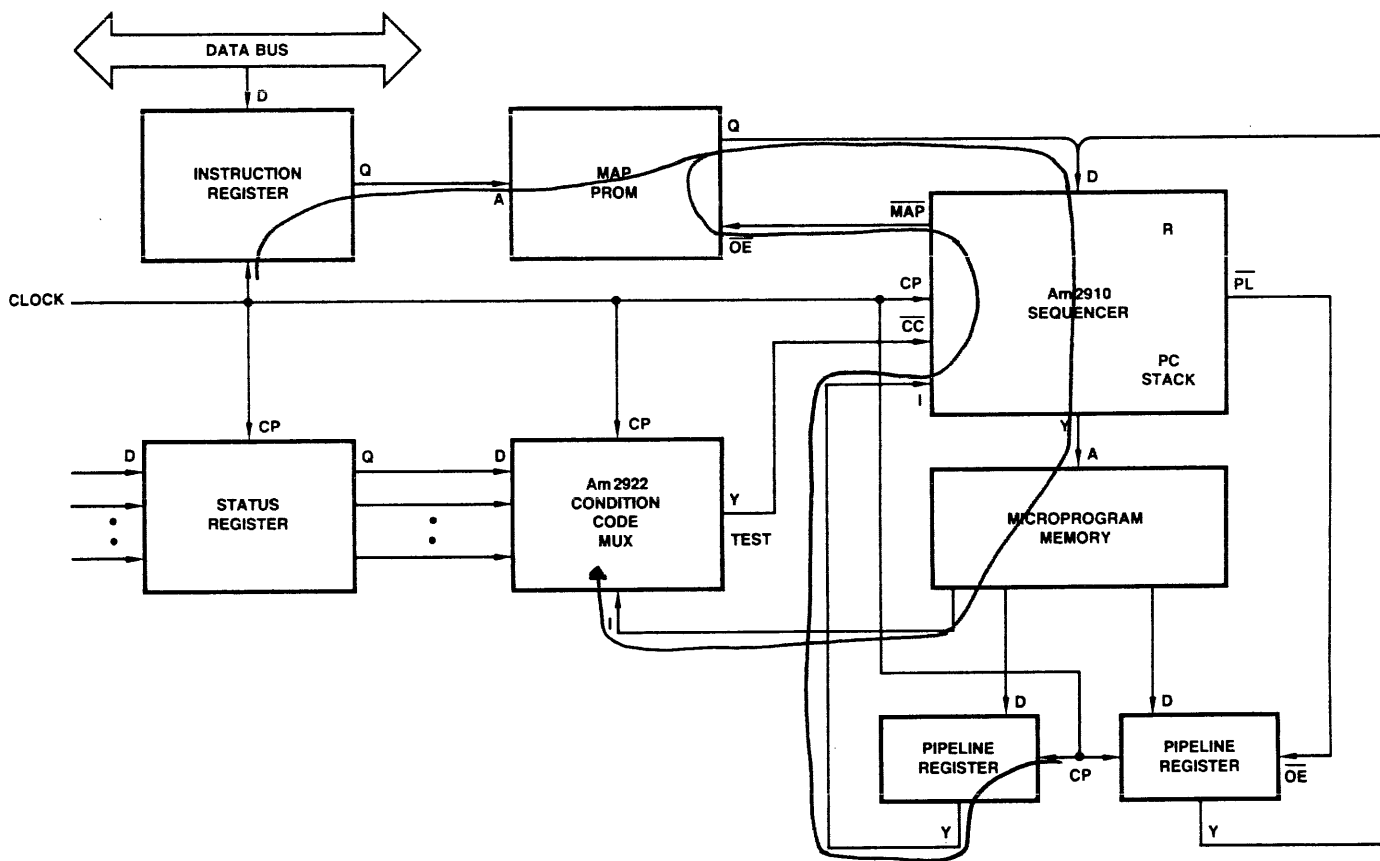
DEVICE	DEVICE PATH	PATH 1	PATH 2	PATH3	PATH 4	PATH 5
PIPELINE	CP → Y	15	15	15	---	---
2910	I → Y	70	70	---	---	---
2910	I → PC SETUP	---	---	104	---	---
2910	CP → Y	---	---	---	55	55
MEMORY	ADDR OUT	50	50	---	50	50
2922	SET-UP	11	---	---	11	---
PIPELINE	SET-UP	---	5	---	---	5
TOTAL ns		146	140	119	116	110

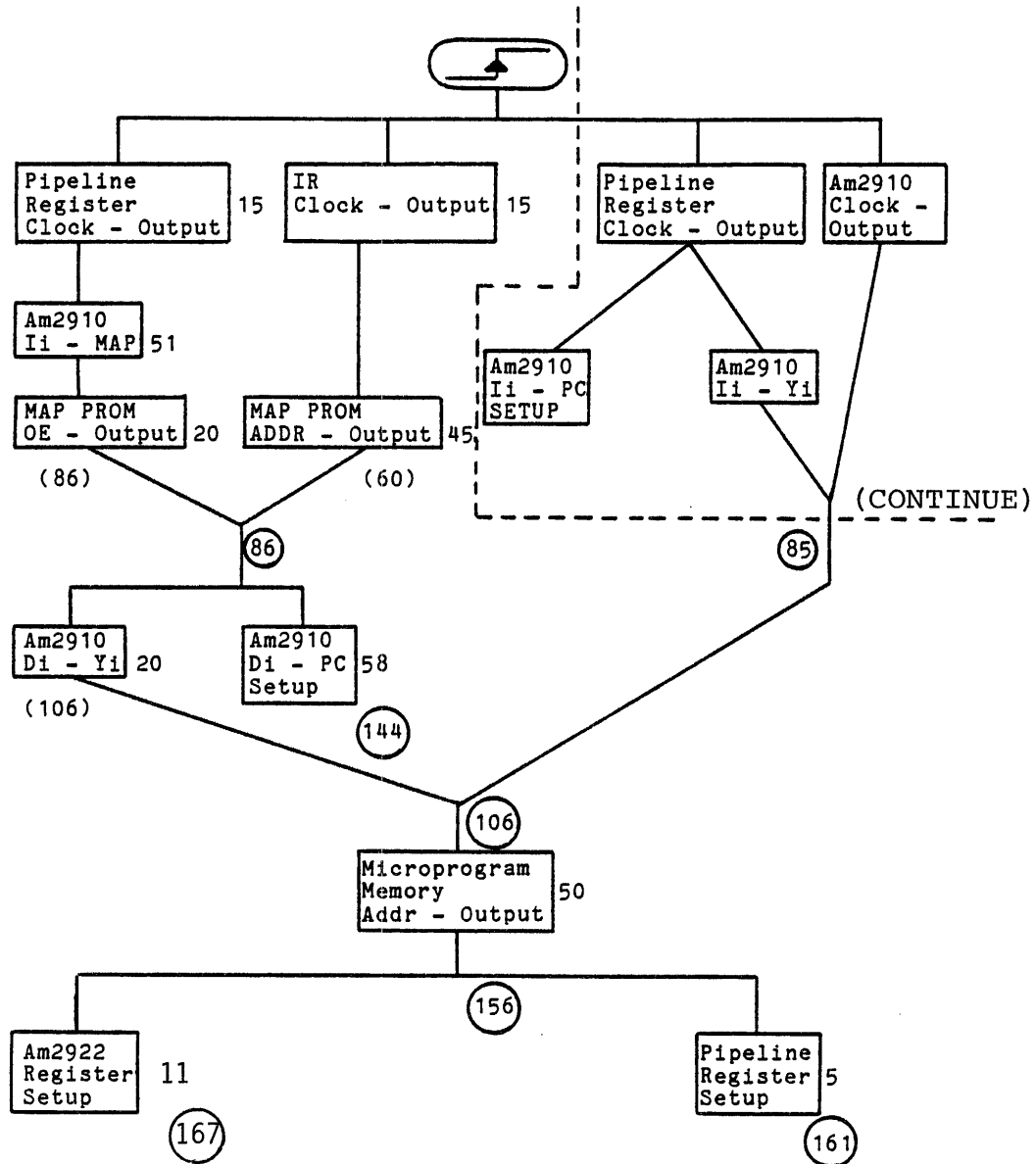


CONTINUE INSTRUCTION

PERT CHART

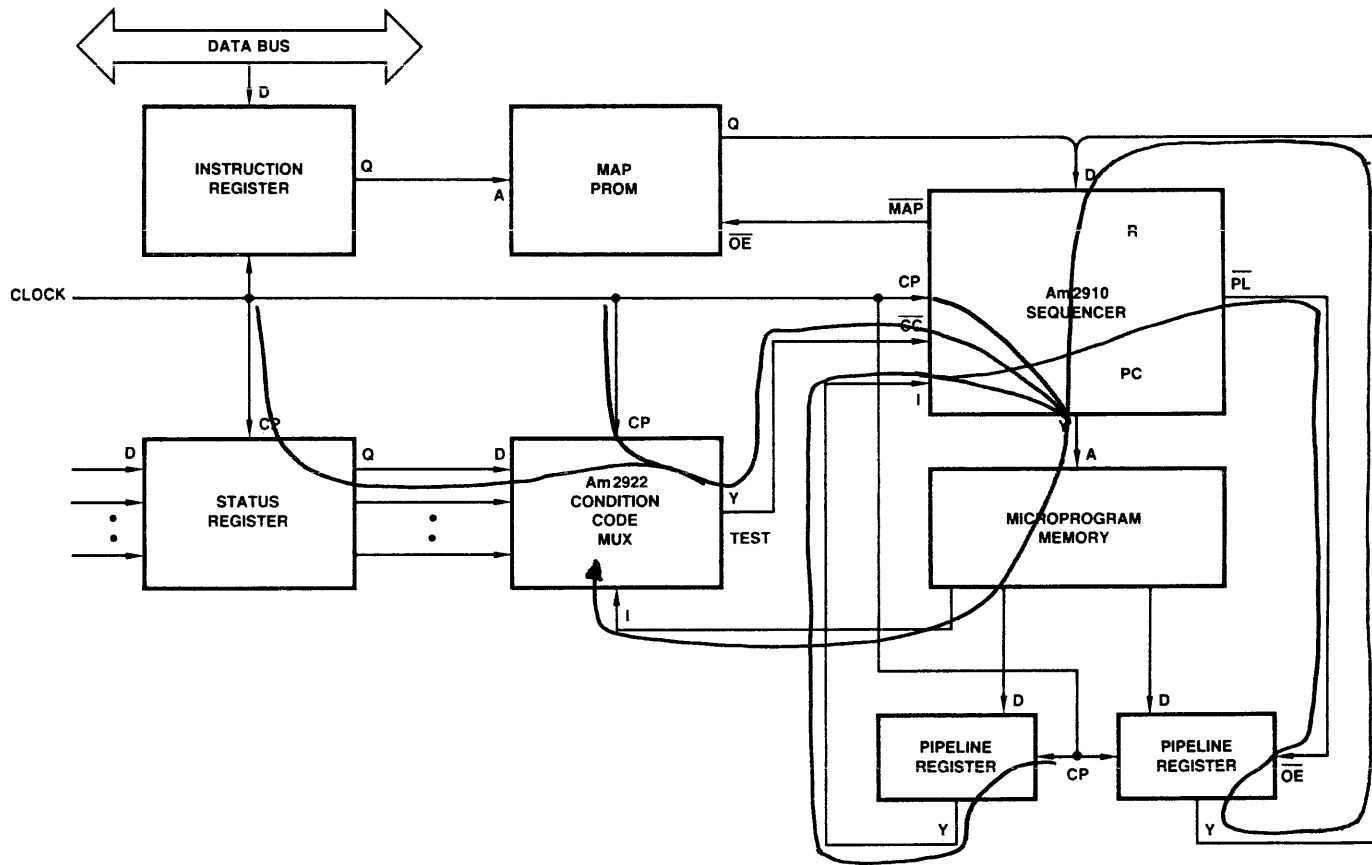
JUMP MAP





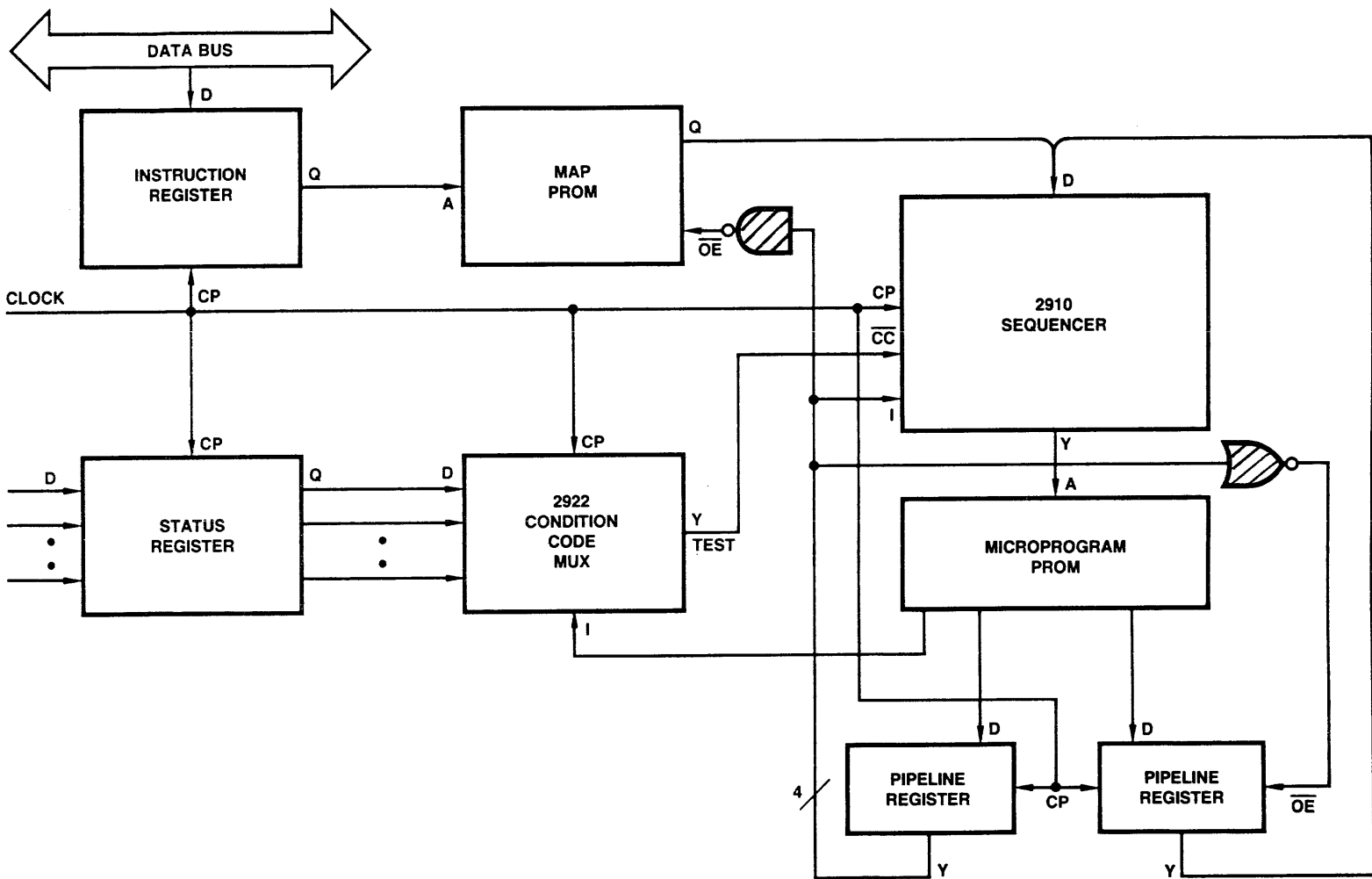
JUMP MAP
PERT CHART

CONDITIONAL JUMP - TAKEN



SPEEDING UP THE MICROCYCLE

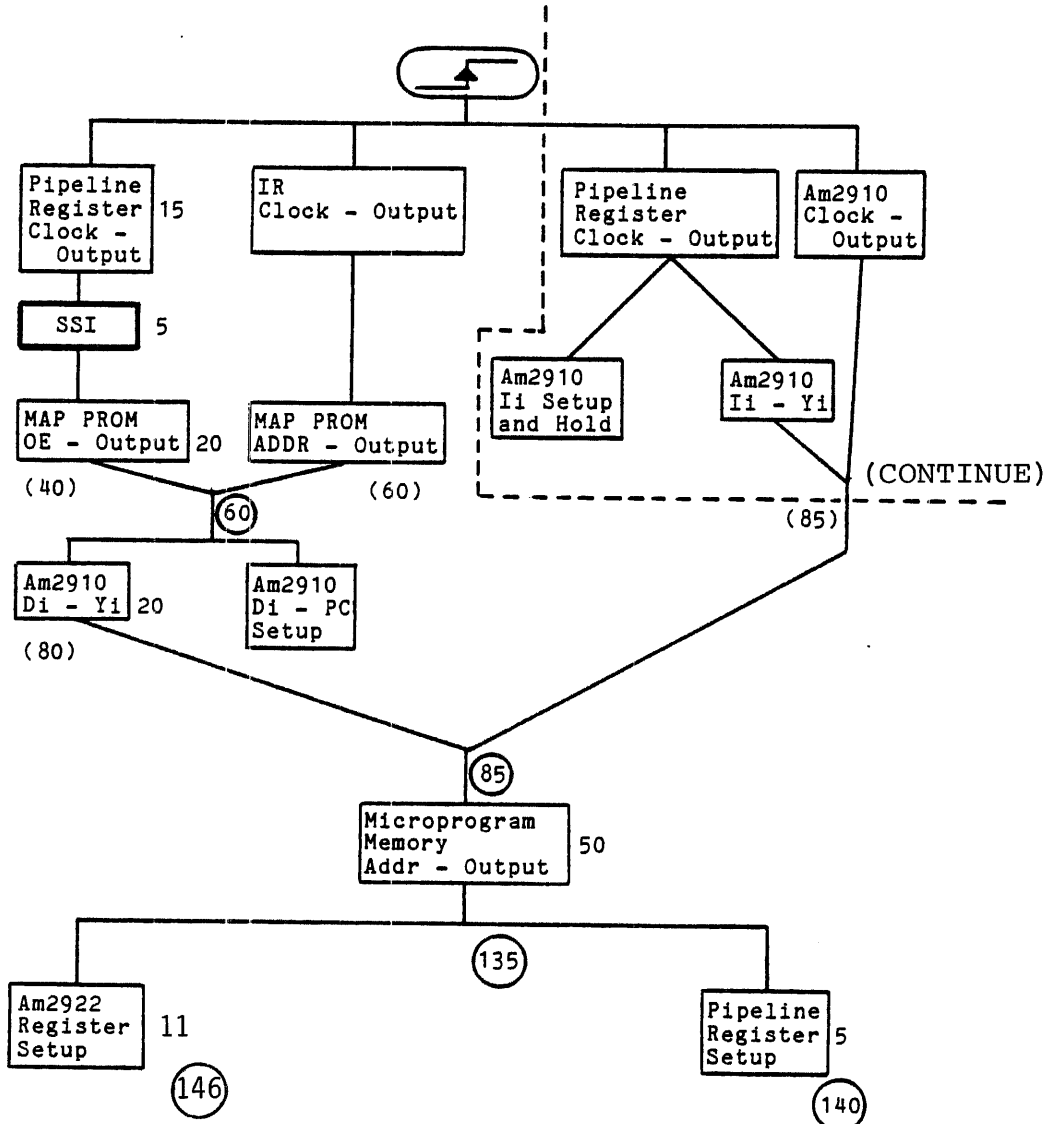
- Consider a change to the architecture to speed up the microcycle.
- Use combinatorial SSI circuits to decode the pipeline enable and map enable directly from the Am2910 instruction inputs.
- Although the SSI delay is small, it too could be eliminated by driving the map and pipeline enables directly from the microword.



1-2310

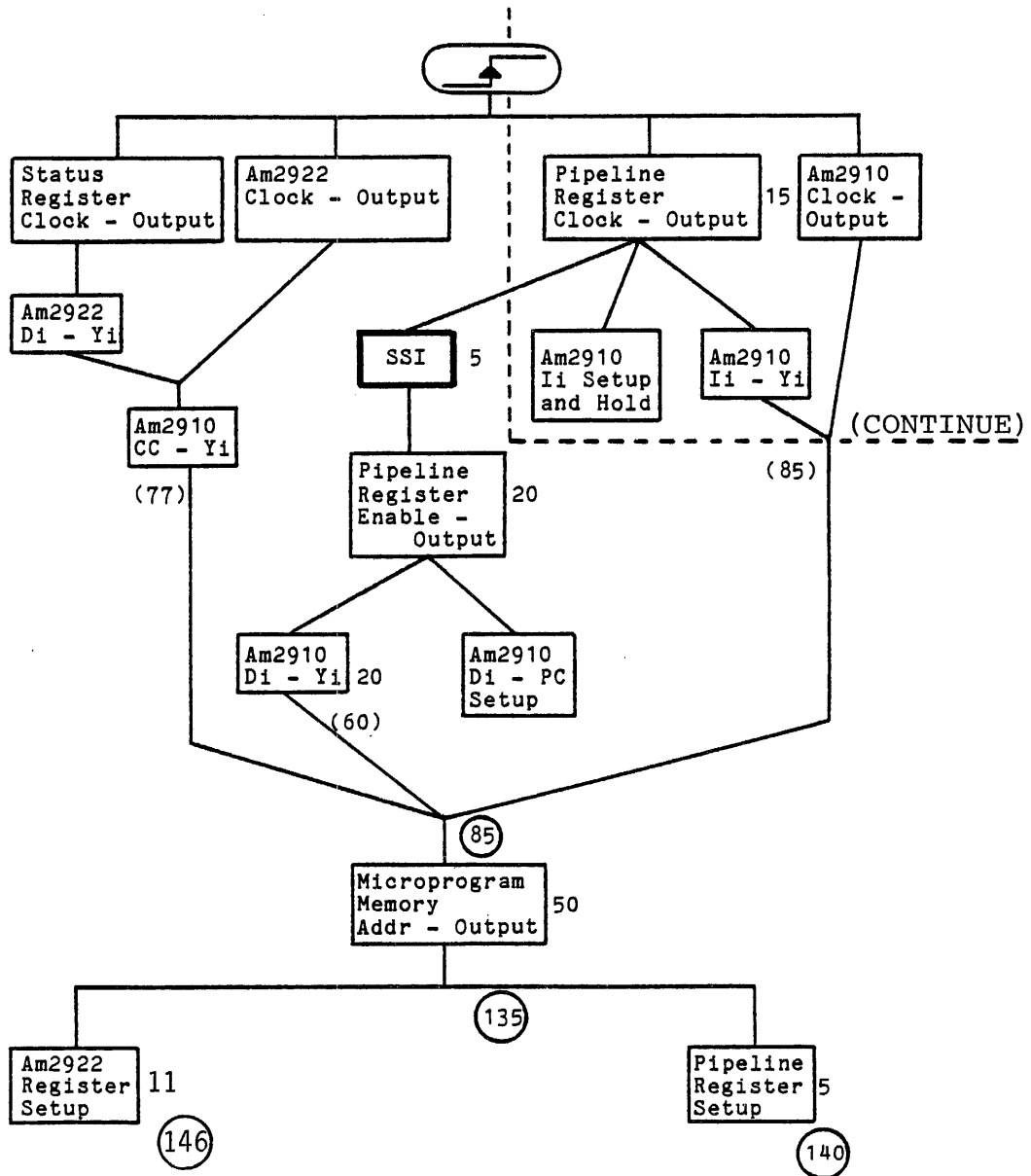
ED2900A

1-2310



JUMP MAP - IMPROVED ARCHITECTURE

PERT CHART



JUMP TAKEN - IMPROVED ARCHITECTURE

PERT CHART

COFFEE MACHINE SOLUTION

(See ED2900A Exercise and Laboratory Manual)