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

INTRODUCTION TO DESIGNING WITH THE Am2900 FAMILY OF MICROPROGAMMING 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 MICROPROGAMMING AND THE Am2900 FAMILY

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.

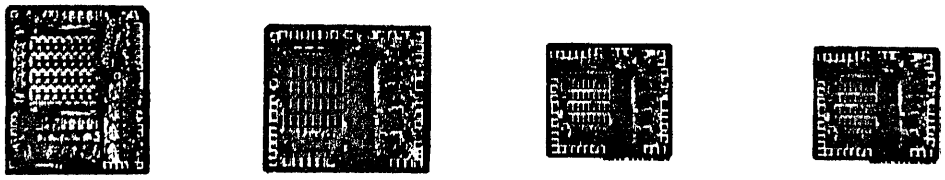
RALU-Arithmetic/Logic Unit (ALU) with registers
Interrupt controller/Direct Memory Access controller
Microprogram sequencer/Macro program controller
Memory controller/Input-Output controller
Microprocessors
16K-bit RAM

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

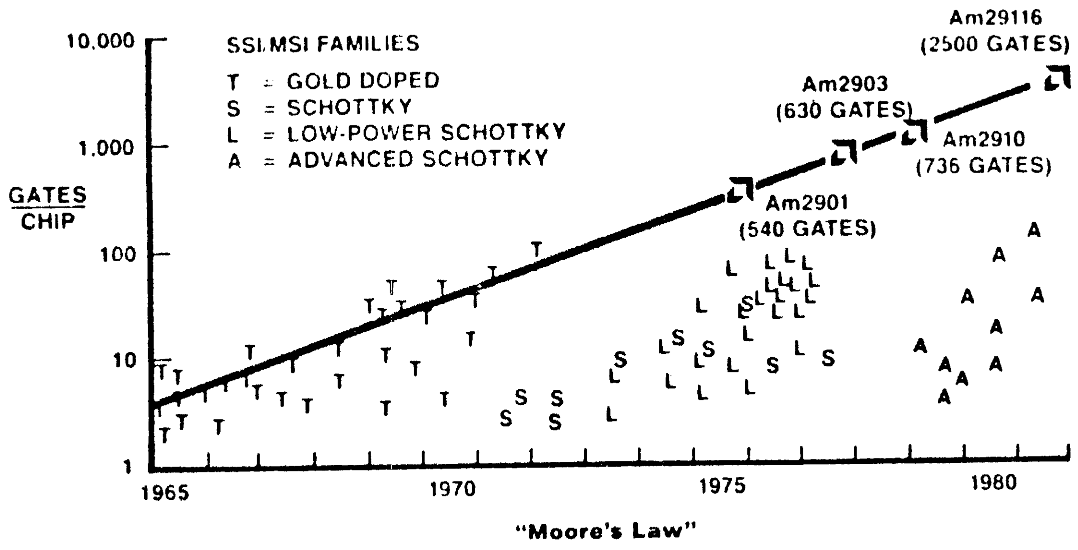
How do you build a large circuit (e.g. a microprocessor) with bipolar speed if it won't fit on a single chip?

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.

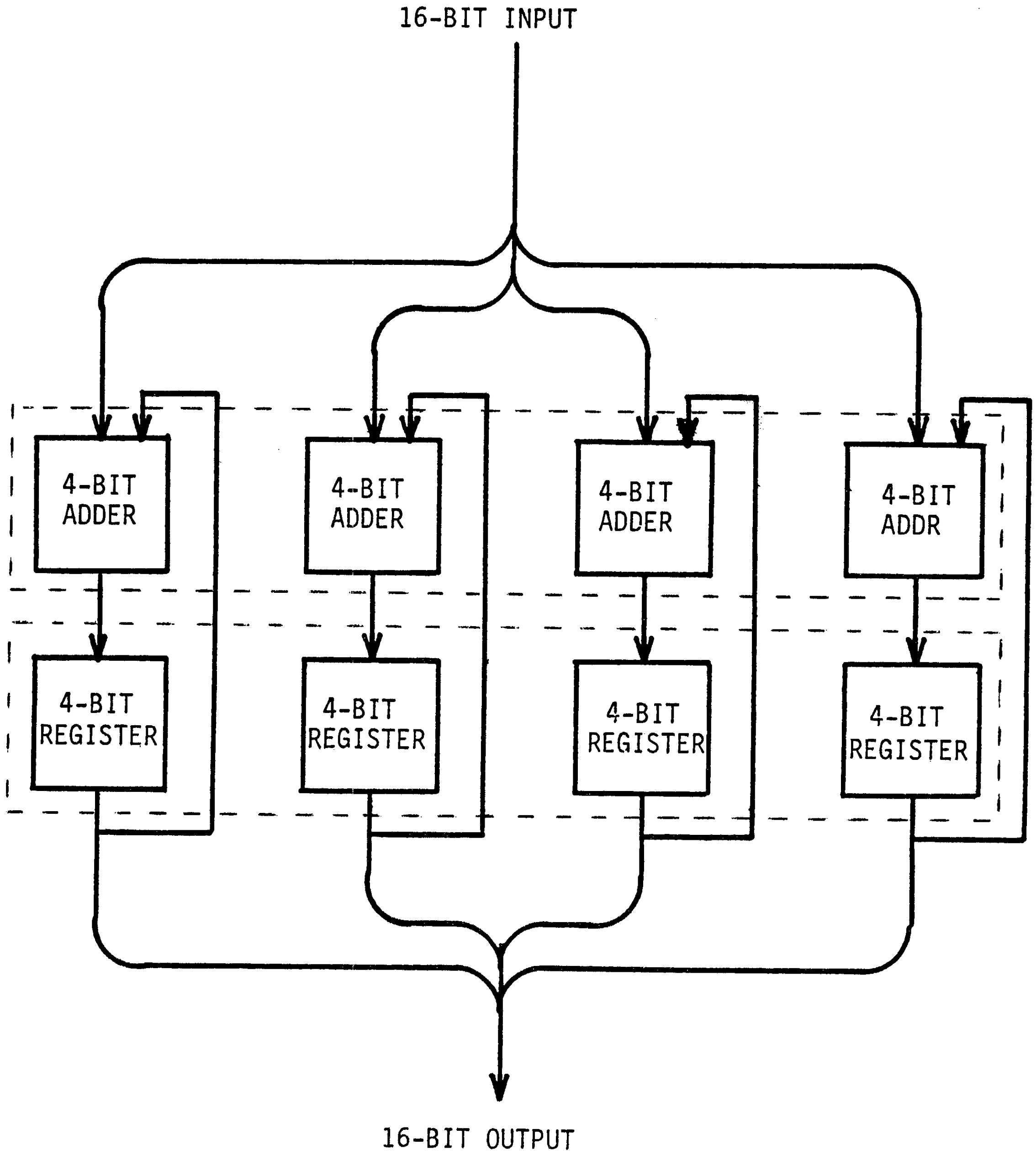
BIT-SLICE ARCHITECTURE

- 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

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

CHIP COUNT FOR SIMPLE SYSTEM

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

ARCHITECTURE

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

INSTRUCTION SET

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

CONCLUSIONS

FIXED INSTR.	BIT SLICE	SSI/MSI
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

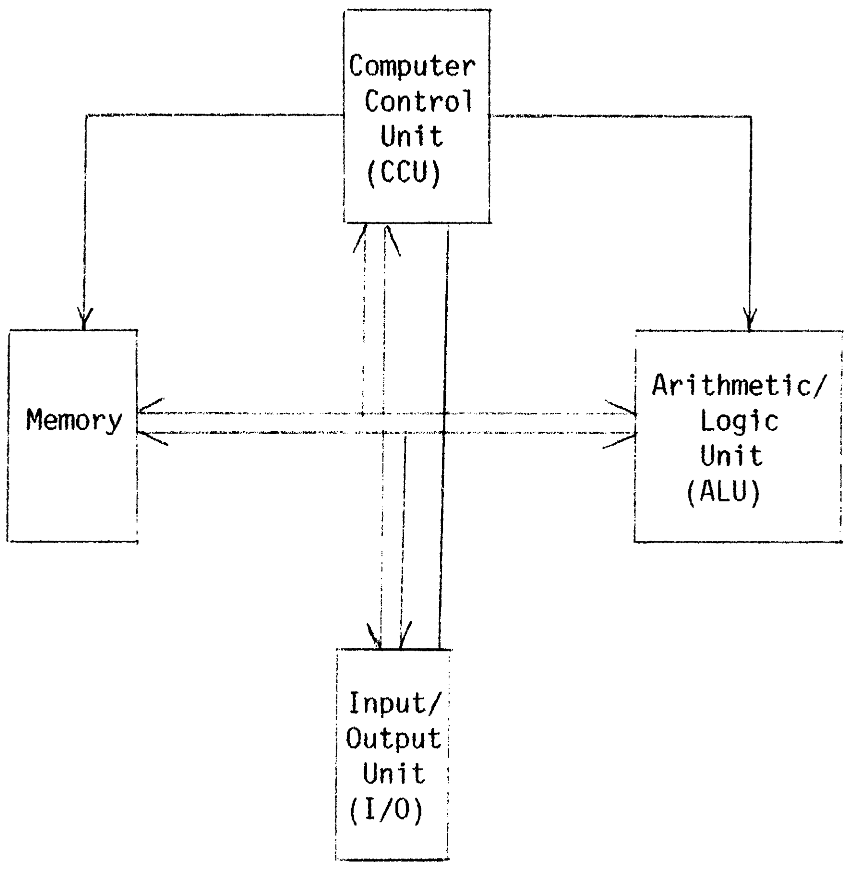
- Higher-order languages (compiler/interpreter translators)
- Lower-order languages (assembler translators)
- Machine language (macro level)
- Register-transfer languages-RTL (microprogramming)
- Boolean algebra (symbolic logic - state diagrams)
- Logic levels (timing diagrams - waveforms)

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.



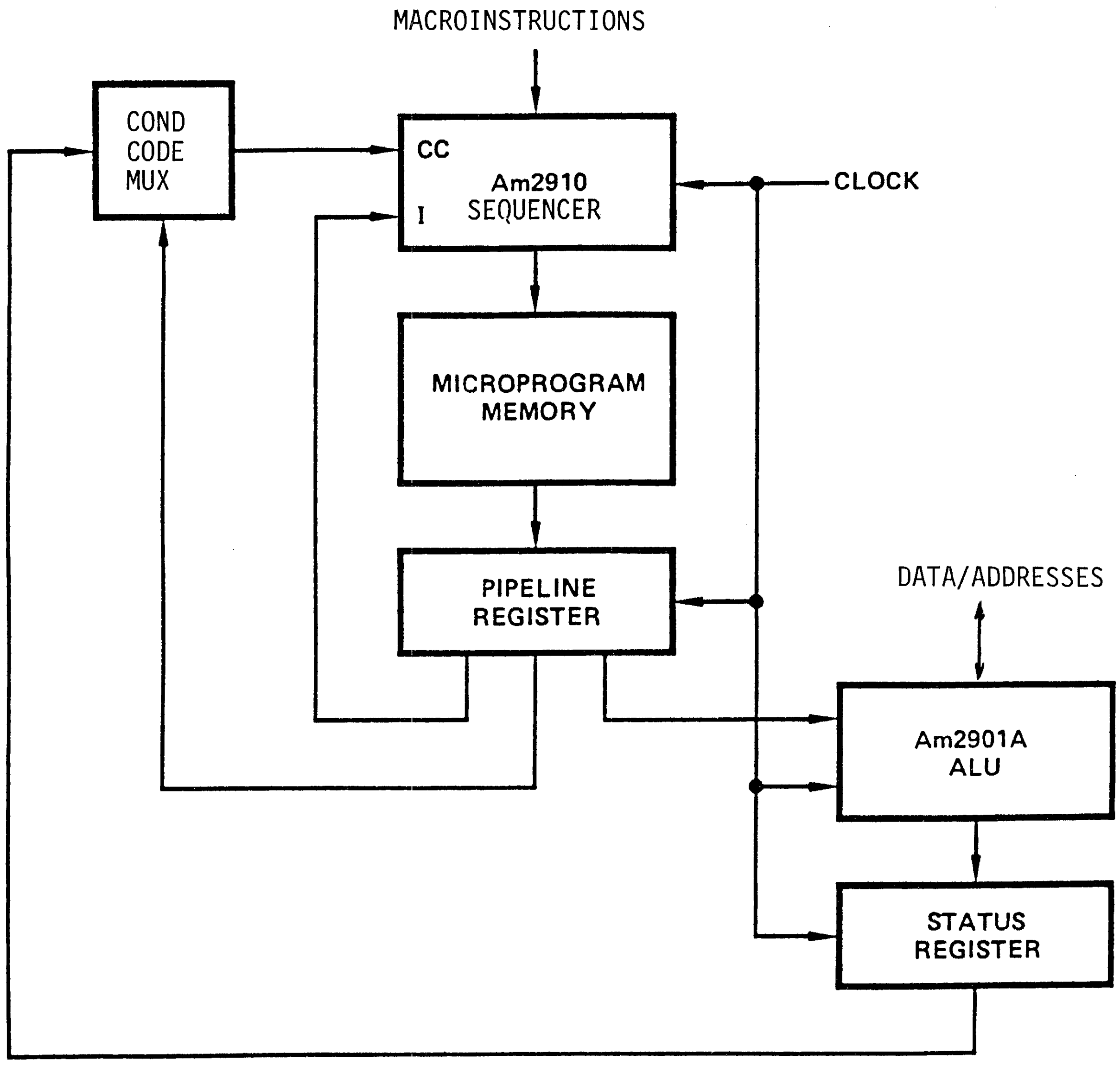
MICROPROGRAMMING DEFINITIONS

- 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

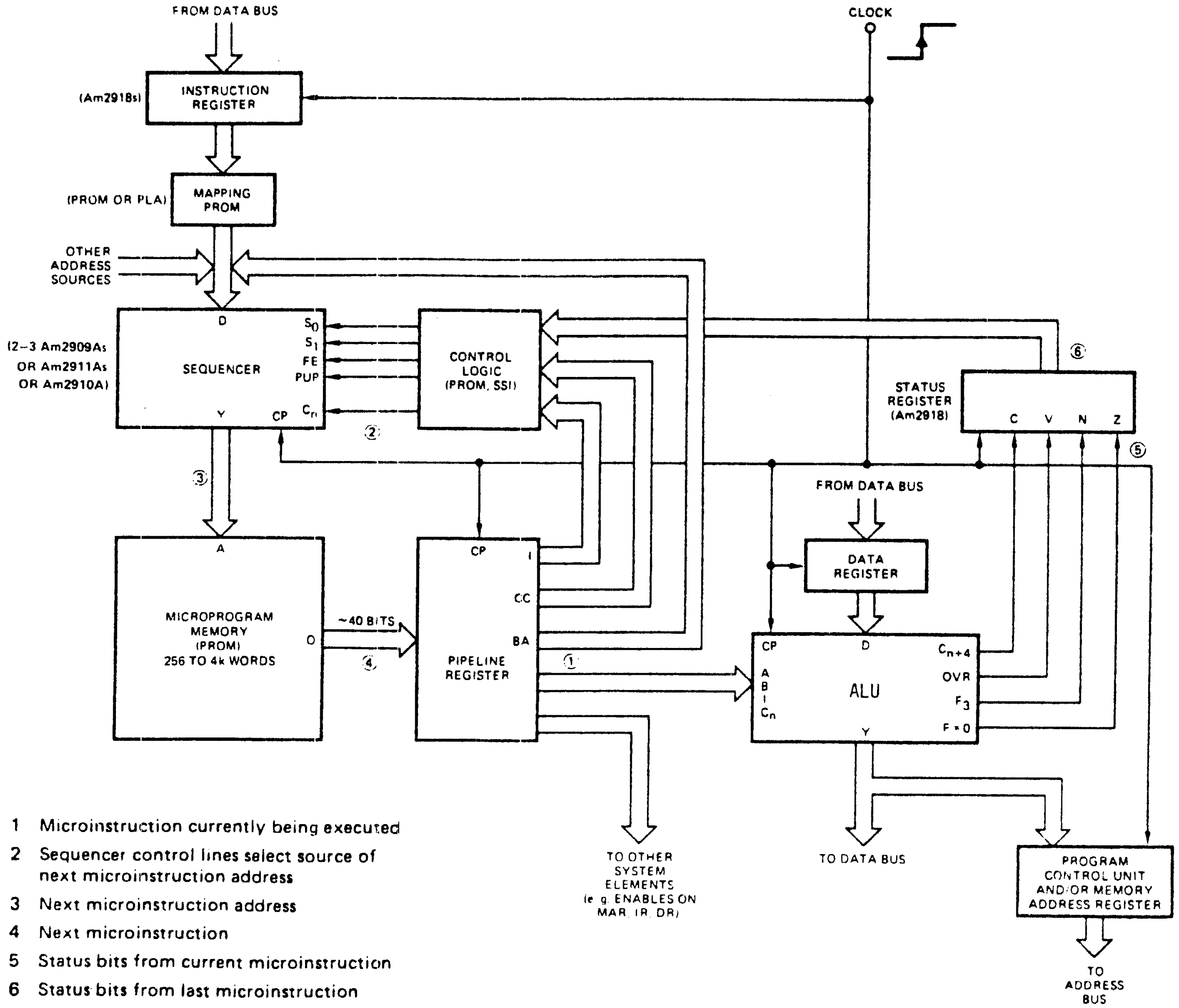
- Microprogramming then consists of defining in an encoded fashion using system variables (registers/variables operations), a step-by-step process of information movement and manipulation. The mnemonic microprogram is then decoded into zeros and ones and is put into a PROM. Each line statement or sequence 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 ARCHITECTURE

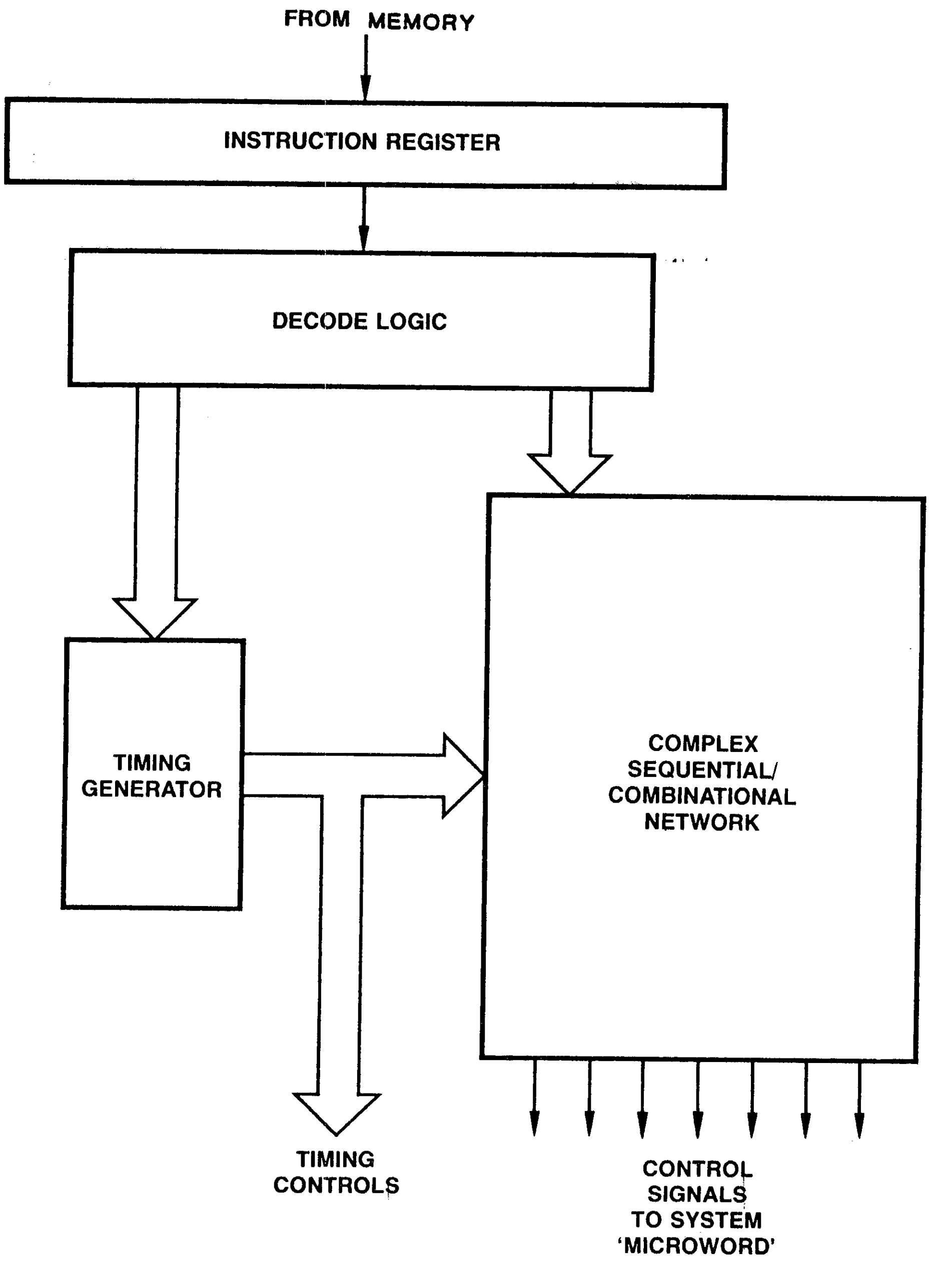


GENERAL MICROPROGRAMMED SYSTEM



1 Microinstruction currently being executed
2 Sequencer control lines select source of next microinstruction address
3 Next microinstruction address
4 Next microinstruction
5 Status bits from current microinstruction
6 Status bits from last microinstruction

TRADITIONAL HARDWIRED CCU:


FROM MEMORY
INSTRUCTION REGISTER
DECODE LOGIC
TIMING GENERATOR
COMPLEX SEQUENTIAL/ COMBINATIONAL NETWORK
TIMING CONTROLS
CONTROL SIGNALS TO SYSTEM 'MICROWORD'

HARDWIRED CCU

Advantages

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

Justification

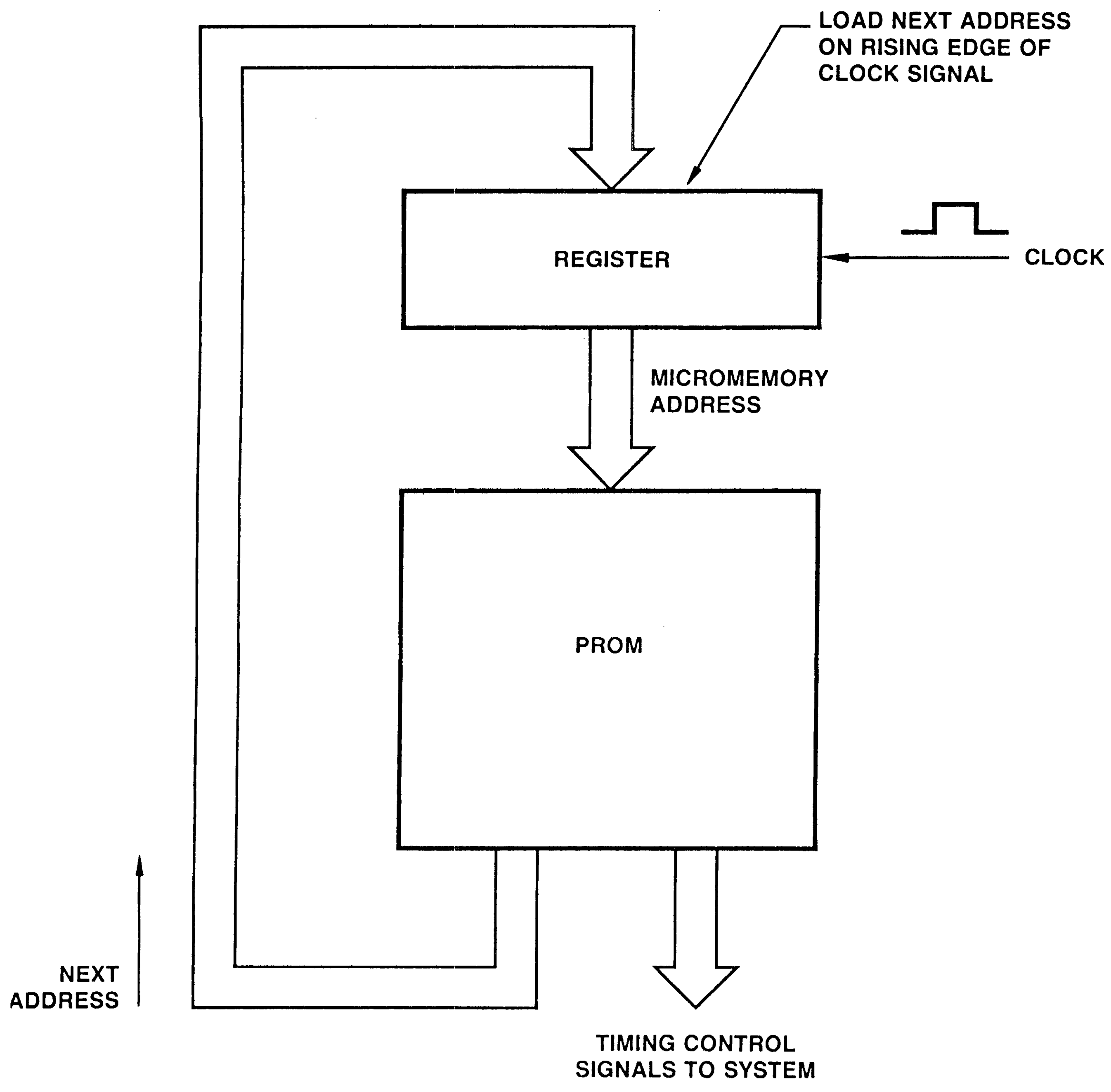
- Suitable if design is rigid or fixed
  for high volume production

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


LOAD NEXT ADDRESS ON RISING EDGE OF CLOCK SIGNAL
REGISTER
CLOCK
MICROMEMORY ADDRESS
PROM
NEXT ADDRESS
TIMING CONTROL SIGNALS TO SYSTEM

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

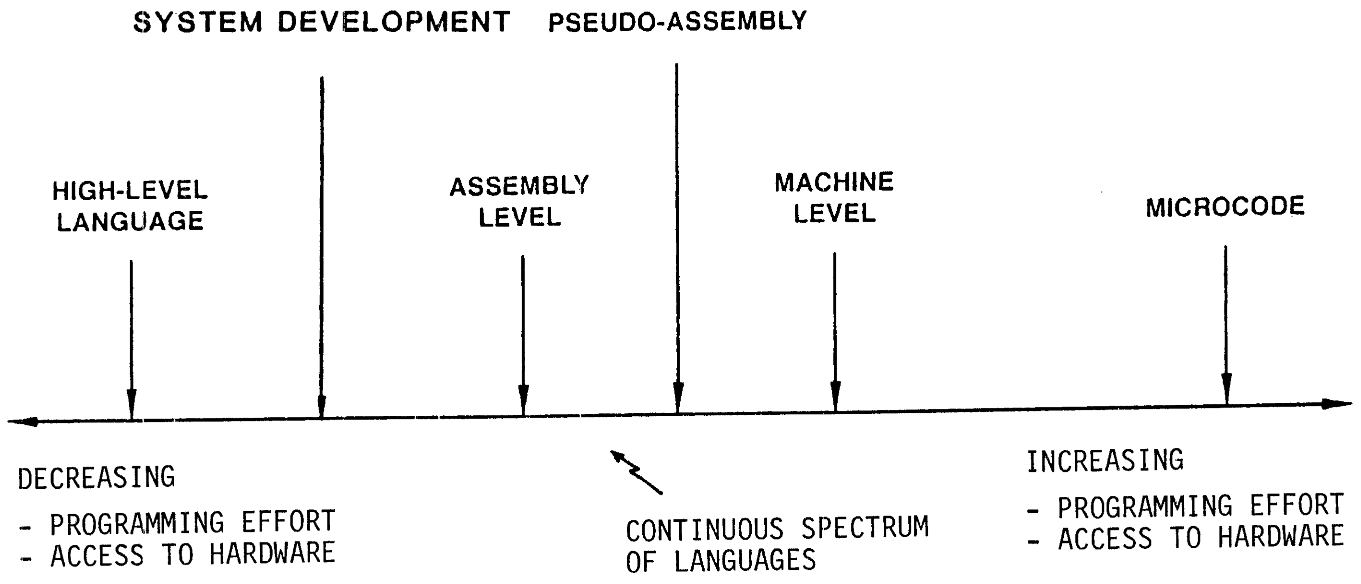
LANGUAGE INTERRELATIONSHIPS

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

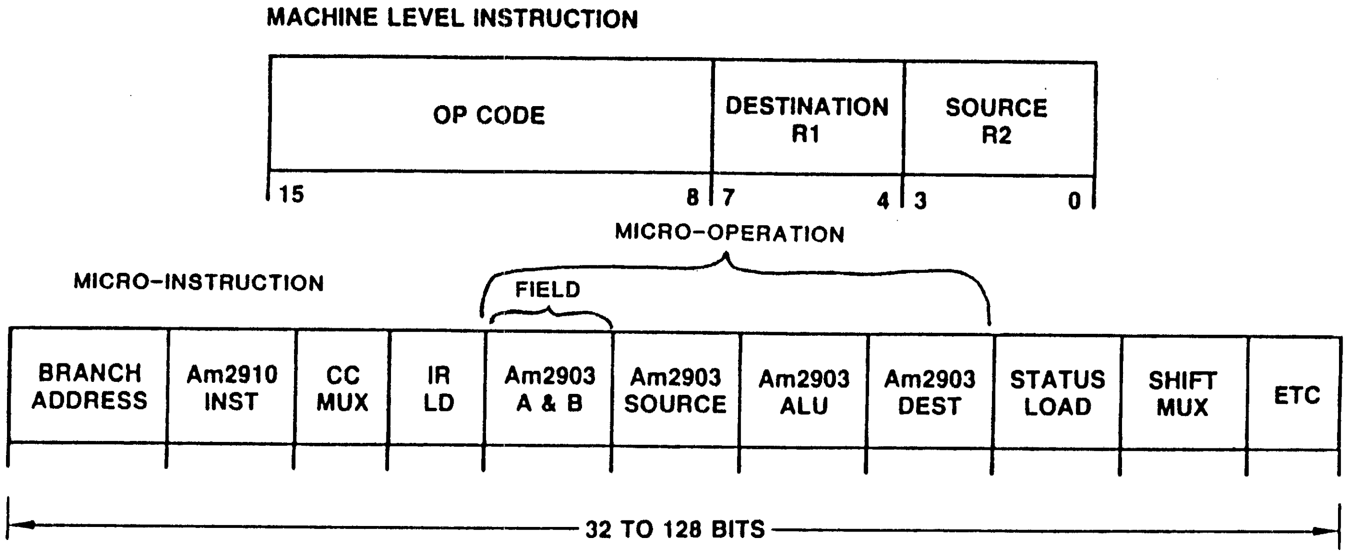
- The combination of all micro-operations or fields that specify the state of all control lines during a time interval (clock cyle).

Micro-operation

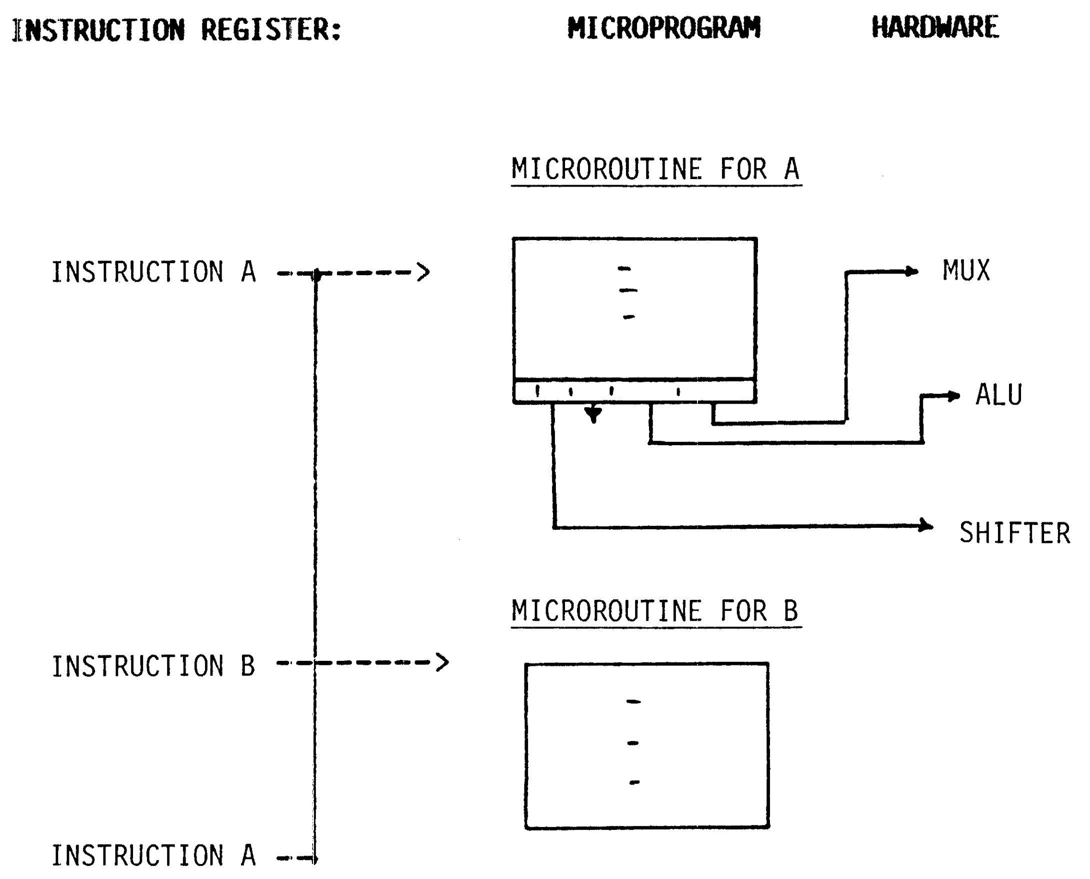
- The combination of one or more fields to control one functional unit, such as the ALU.

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
OP CODE
DESTINATION R1
SOURCE R2
15
8
7
4
3
0
MICRO-OPERATION
MICRO-INSTRUCTION
FIELD
BRANCH ADDRESS
Am2910 INST
CC MUX
IR LD
Am2903 A & B
Am2903 SOURCE
Am2903 ALU
Am2903 DEST
STATUS LOAD
SHIFT MUX
ETC
32 TO 128 BITS



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, etch)
- 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**

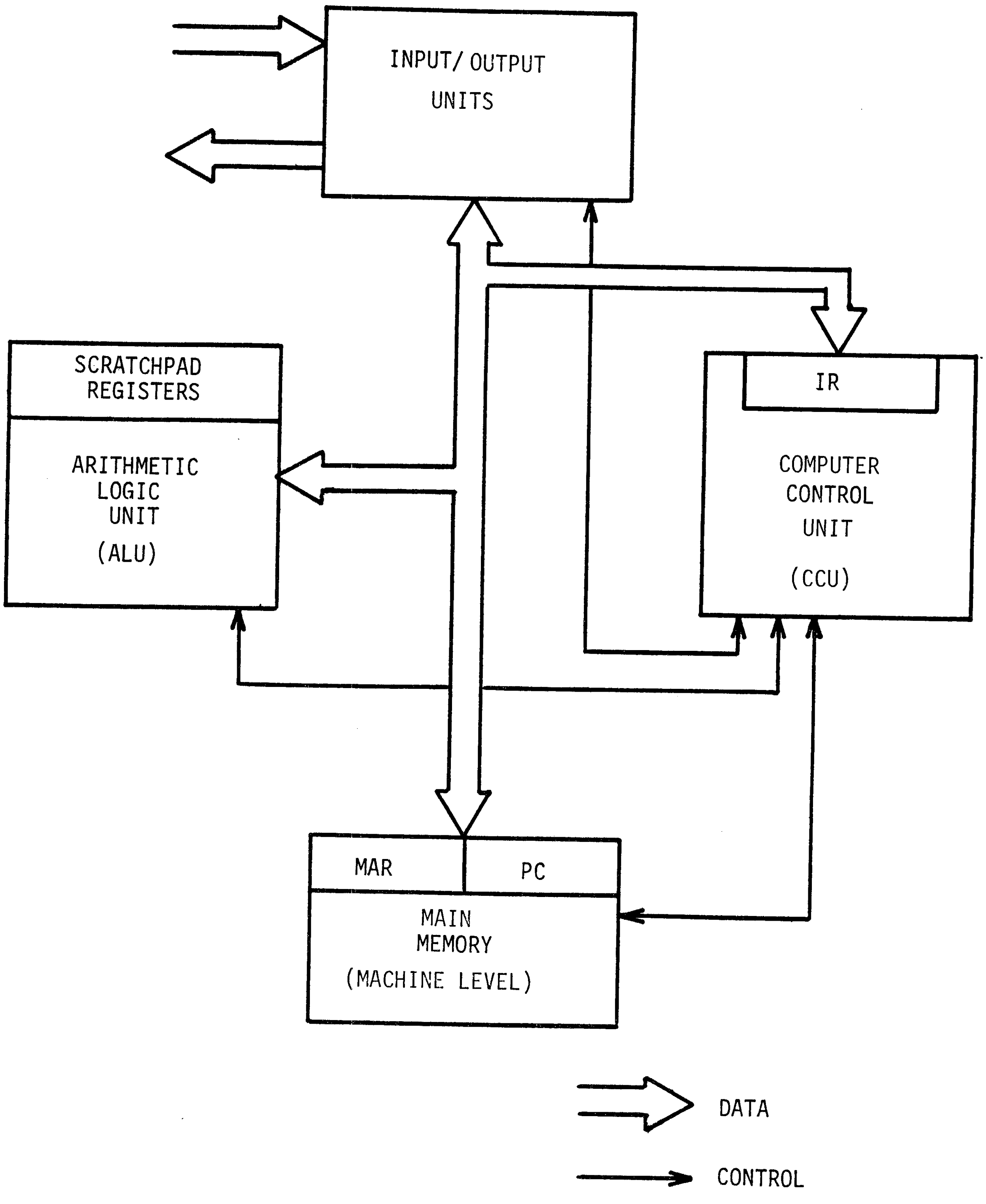
Registered PROM, 512 x 8	Am27S25
Registered PROM, 512 x 8	Am27S27
Registered PROM, 1024 x 8	Am27S35
Registered PROM, 1024 x 8	Am27S37
Registered PROM, 2048 x 8	Am27S45
Registered PROM, 2048 x 8	Am27S47

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).

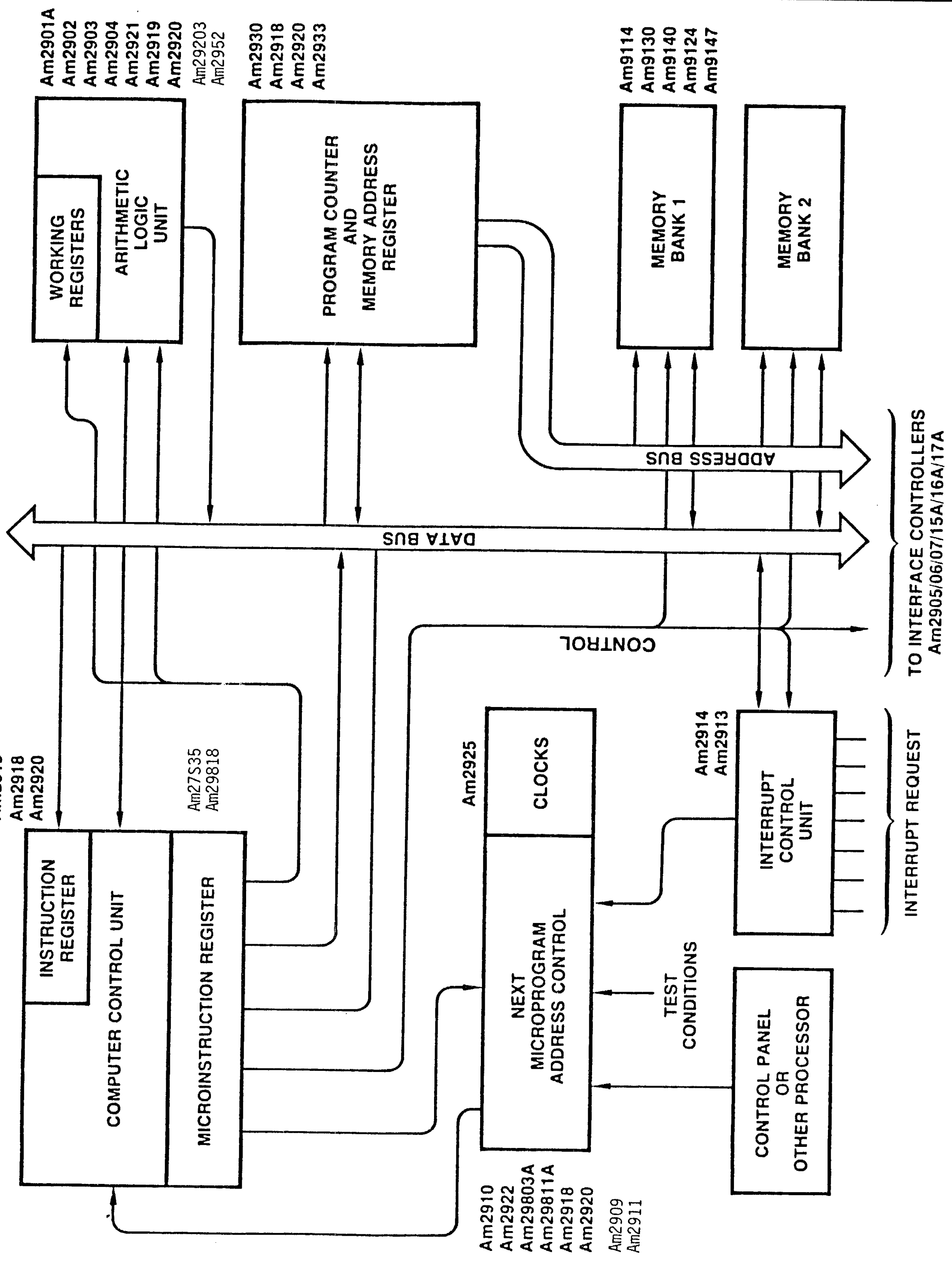

INPUT/ OUTPUT
UNITS
SCRATCHPAD
REGISTERS
ARITHMETIC
LOGIC
UNIT
(ALU)
IR
COMPUTER
CONTROL
UNIT
(CCU)
MAR
PC
MAIN
MEMORY
(MACHINE LEVEL)
DATA
CONTROL

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.


Am2901A
Am2902
Am2903
Am2904
Am2921
Am2919
Am2920
Am29203
Am2952
WORKING REGISTERS
ARITHMETIC LOGIC UNIT
Am2930
Am2918
Am2920
Am2933
PROGRAM COUNTER AND MEMORY ADDRESS REGISTER
Am9114
Am9130
Am9140
Am9124
Am9147
MEMORY BANK 1
MEMORY BANK 2
ADDRESS BUS
DATA BUS
CONTROL
TO INTERFACE CONTROLLERS Am2905/06/07/15A/16A/17A
Am2919
Am2918
Am2920
INSTRUCTION REGISTER
COMPUTER CONTROL UNIT
Am27S35
Am29818
MICROINSTRUCTION REGISTER
Am2925
CLOCKS
NEXT MICROPROGRAM ADDRESS CONTROL
Am2910
Am2922
Am29803A
Am29811A
Am2918
Am2920
Am2909
Am2911
TEST CONDITIONS
Am2914
Am2913
INTERRUPT CONTROL UNIT
INTERRUPT REQUEST
CONTROL PANEL OR OTHER PROCESSOR

SIMPLIFIED SYSTEM:

- In order to initially concentrate on the sequence controller (CCU) the remainder of the computer is simplified to

  - an ALU

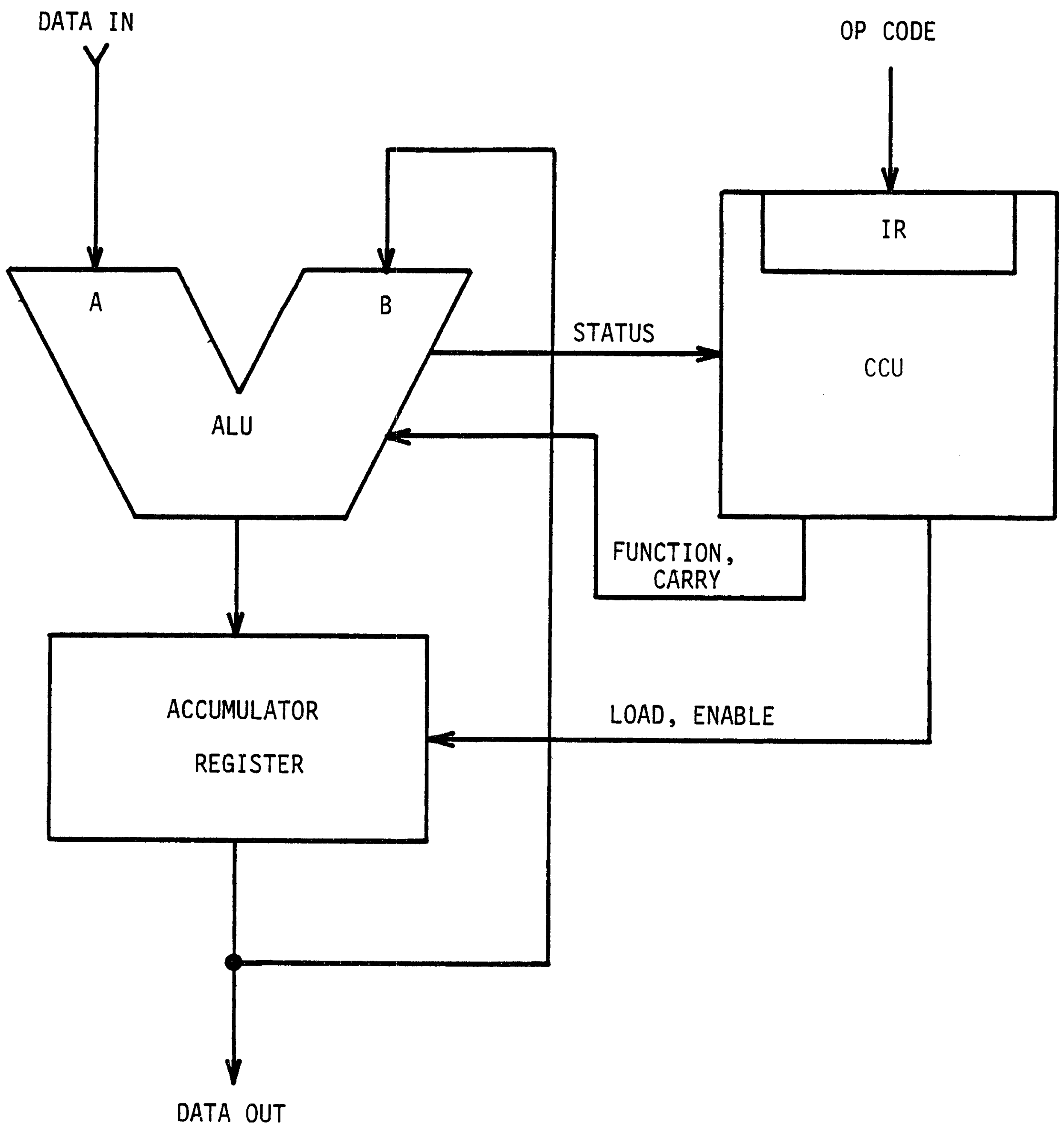
  - the accumulator register (ACC)

- 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


DATA IN
OP CODE
IR
A
B
STATUS
CCU
ALU
FUNCTION, CARRY
ACCUMULATOR REGISTER
LOAD, ENABLE
DATA OUT

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	$\overline{A} \wedge B$ "NOT A AND B"	
1 1 0	$A \forall B$ "A EXOR B"	
1 1 1	$\overline{A \forall B}$ "NOT (A EXOR B)"	

MACHINE INSTRUCTION SUPPORTED:

ADD

SUB

OR

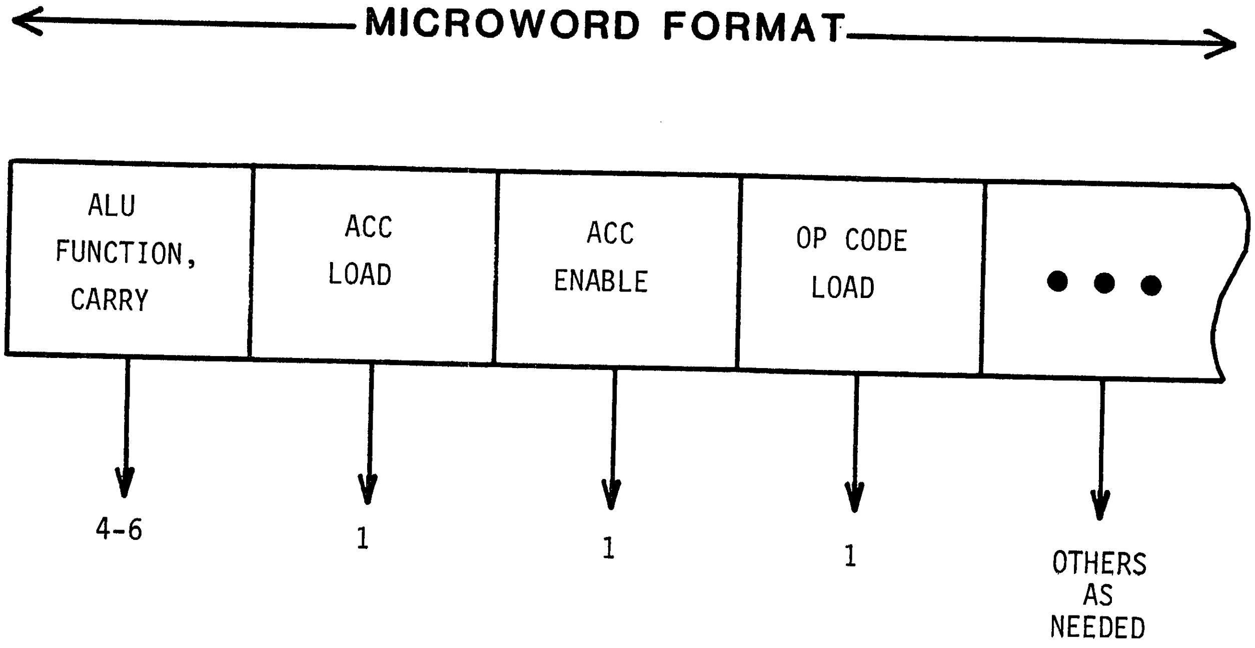
AND

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
ALU FUNCTION, CARRY
ACC LOAD
ACC ENABLE
OP CODE LOAD
• • •
4-6
1
1
1
OTHERS AS NEEDED

SIMPLE CCU:

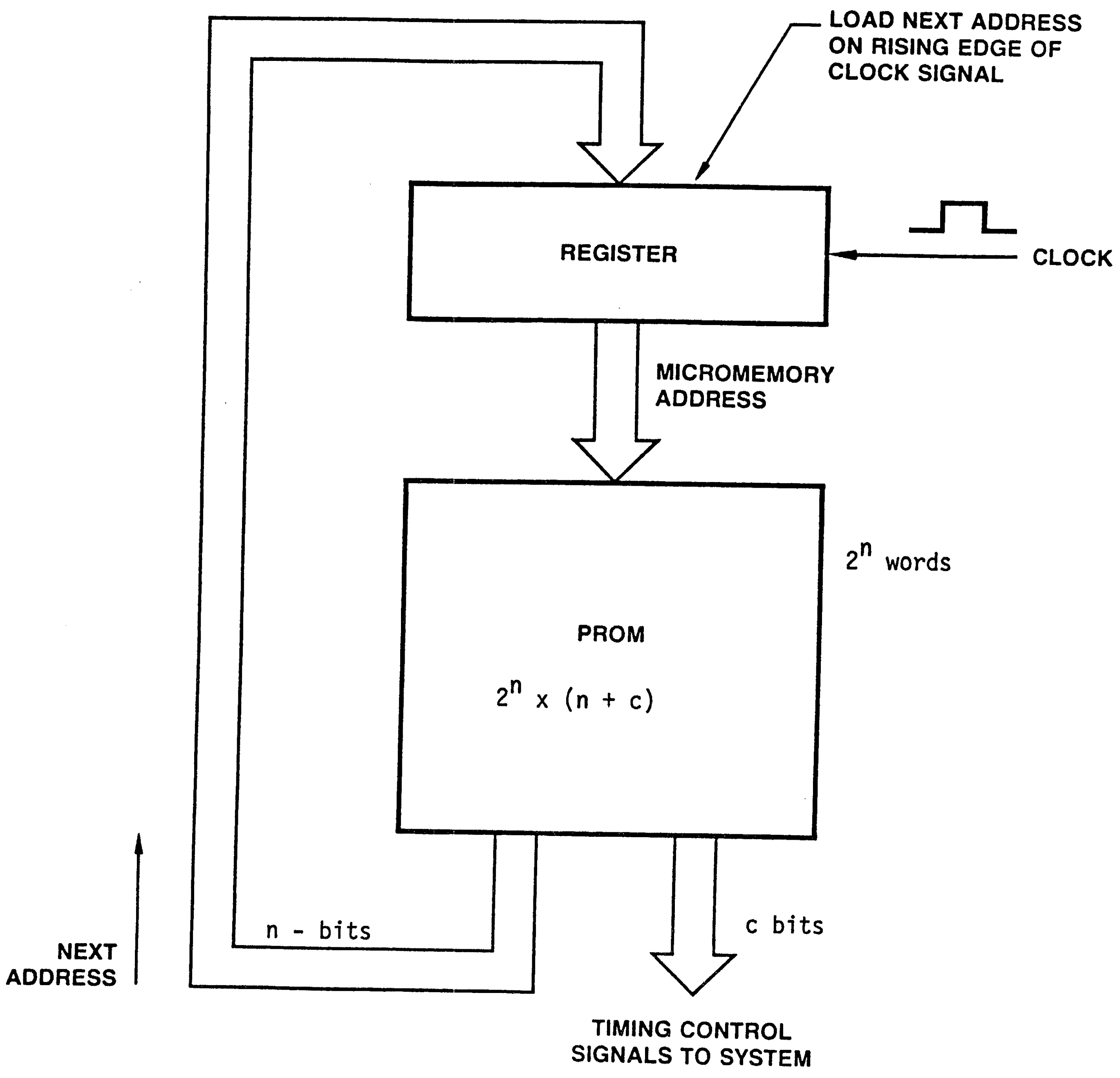
- Each microinstruction contains the address of the next microinstruction to be executed in addition to the fields for the necessary functional unit control signals. The result is a single-sequence controller (i.e. no conditional decisions). Any microinstruction can unconditionally "jump" to any other 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

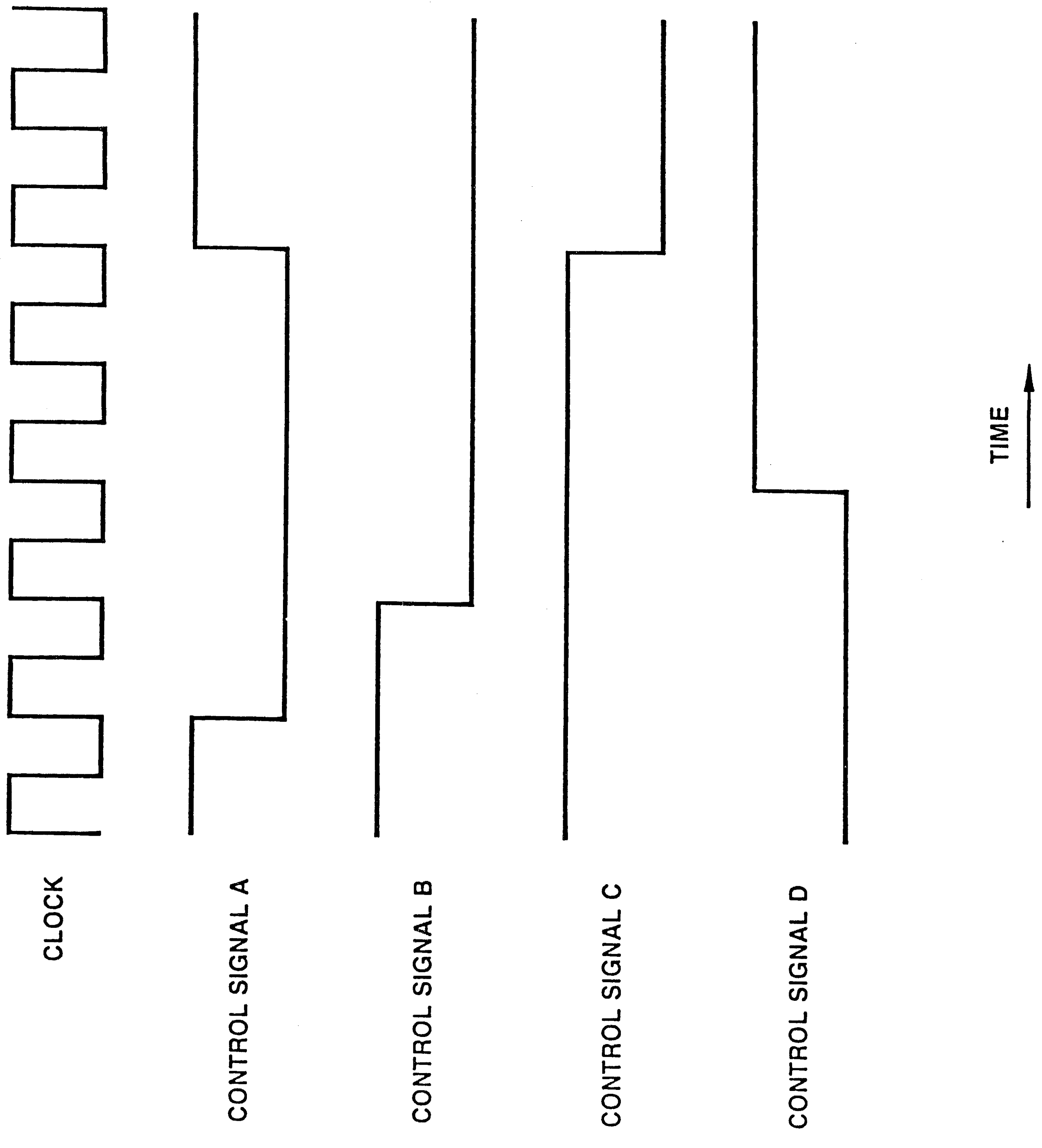
  - the bit pattern formed by the time slice is defined as the microword

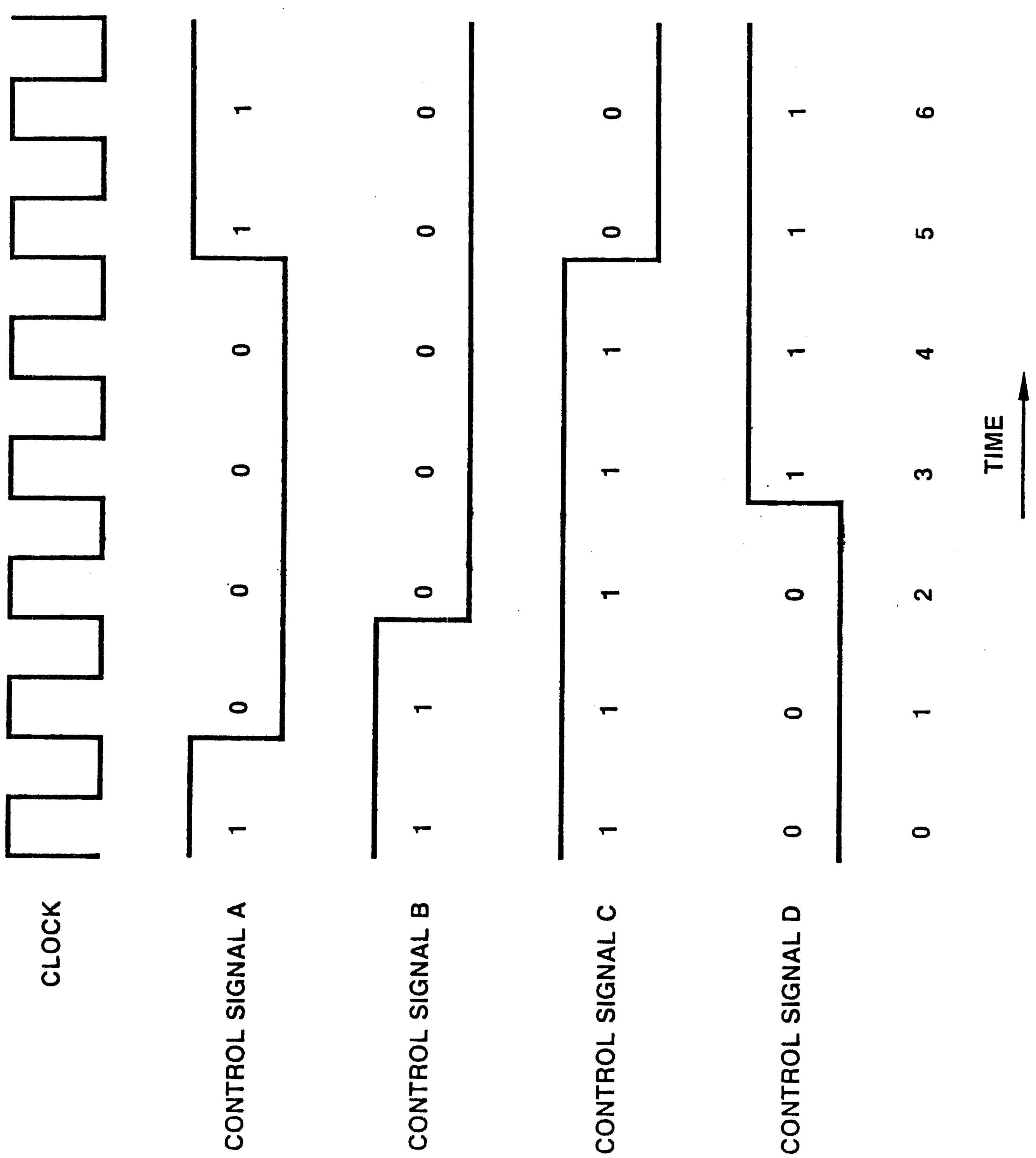
- The following three pages present :

  - a timing diagram for a four-signal system

  - the timing diagram digitized on the clock edge

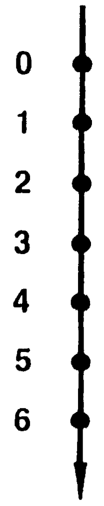
  - the resulting program flow and the clocked microprogram that would generate the desired timing diagram


CLOCK
CONTROL SIGNAL A
CONTROL SIGNAL B
CONTROL SIGNAL C
CONTROL SIGNAL D
TIME


CLOCK
CONTROL SIGNAL A
1 0 0 0 0 1 1
CONTROL SIGNAL B
1 1 0 0 0 0 0
CONTROL SIGNAL C
1 1 1 1 1 0 0
CONTROL SIGNAL D
0 0 0 1 1 1 1
0 1 2 3 4 5 6
TIME

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

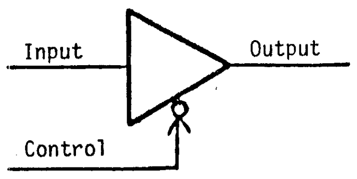
PROGRAM FLOW CONTROL ADDITION (conditional branches)

- 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.

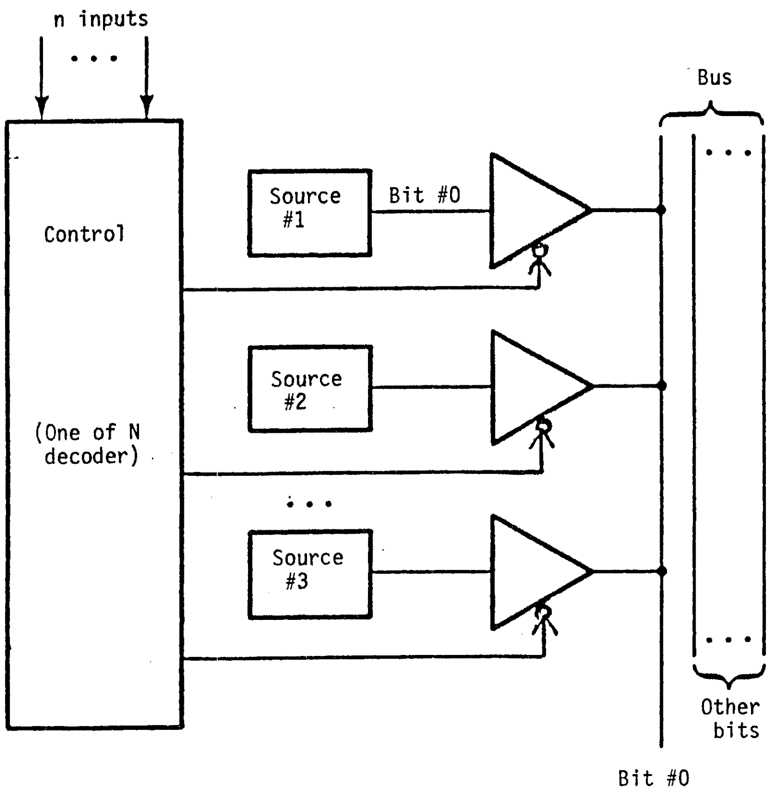
- The controller just described can execute one serial sequence of operations. In order to select from multiple sequences and to allow conditional branching, further addressing hardware is necessary. The current CCU configuration will be enhanced with additional hardware to provide this capability.

- 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 Ø is illustrated, all other bits would be attached similarly.

PROGRAM FLOW CONTROL (Cont'd)

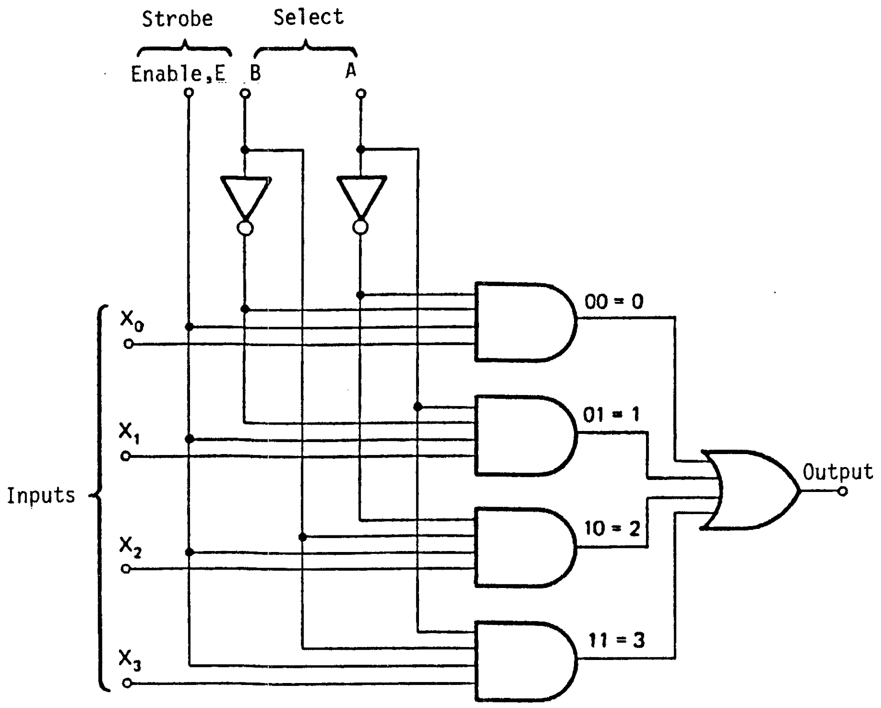
- A "load counter" signal allows the counter to be loaded from one of these tri-state sources or to simply be incremented.

- 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.

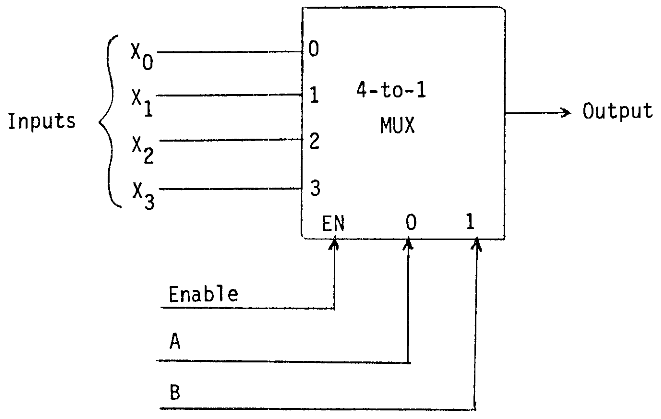
- The **multiplexer (MUX)** and polarity circuits provide the test signals for conditional jumps, and will be developed in more detail later. Likewise, the instruction register (IR) and its associated mapping PROM, which allow the introduction of new micro-addresses, will be developed later.

- 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



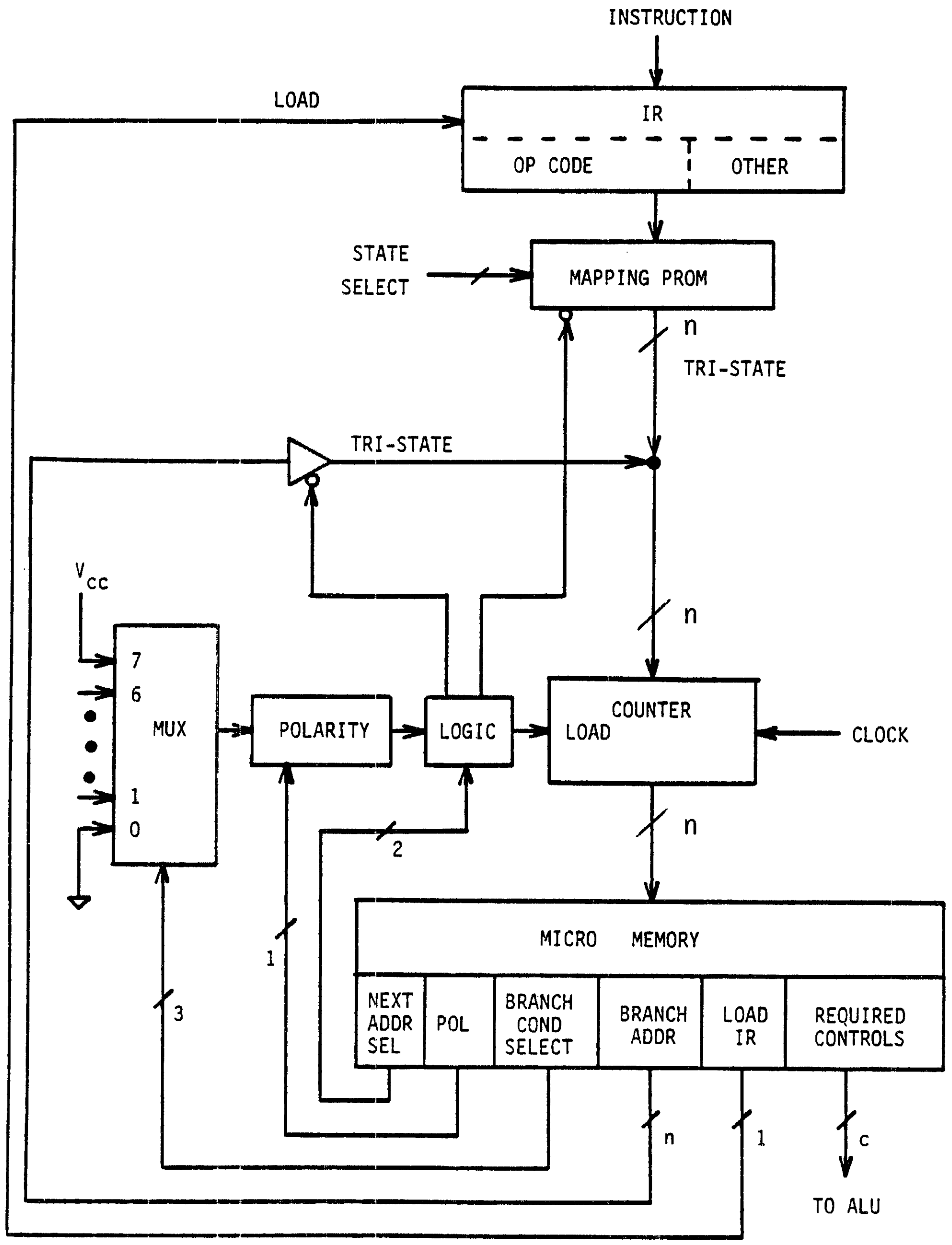
Logic Circuit



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)

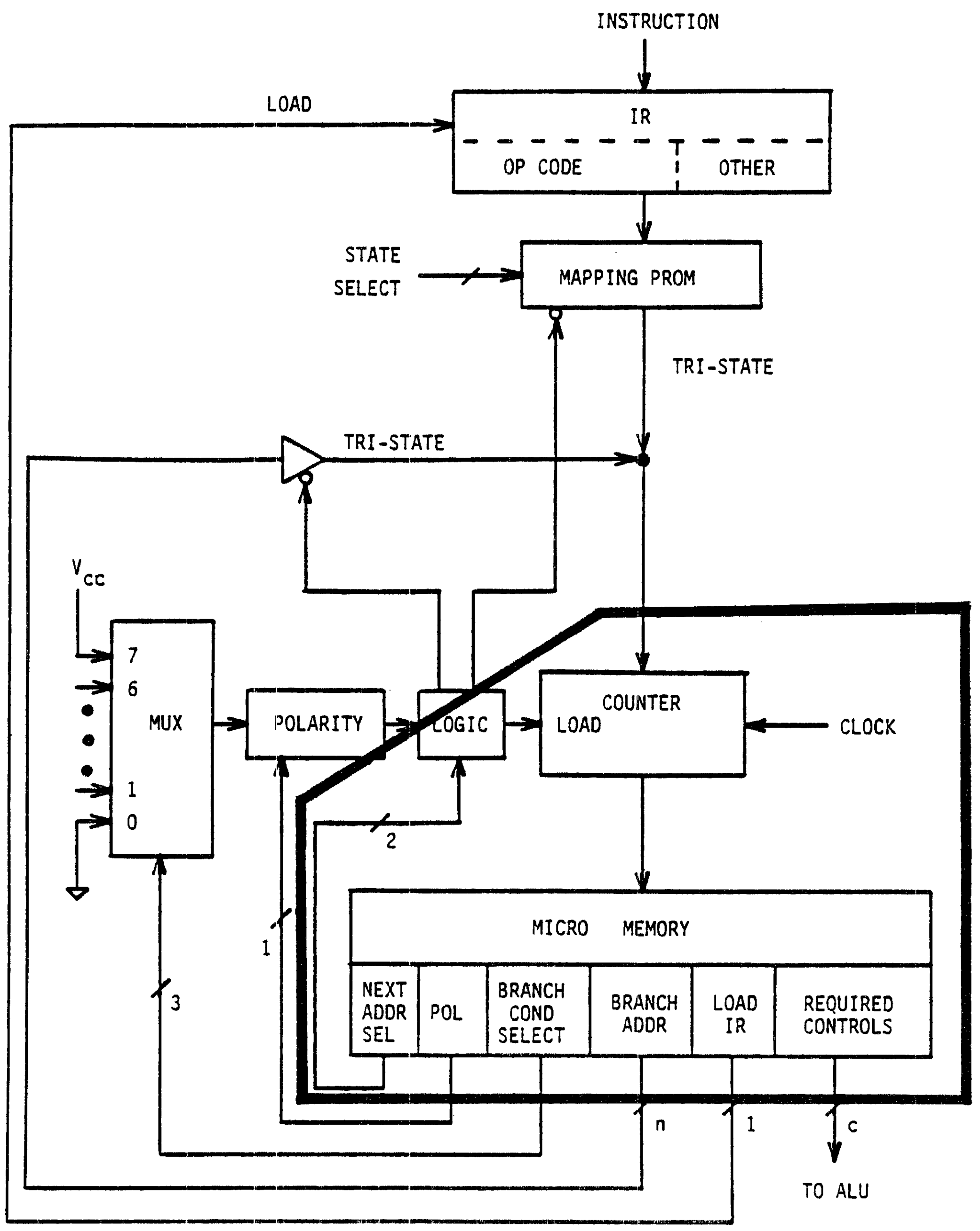
- In programming, quite often one instruction follows another. This is true of microprogramming as well. In the CCU, this is facilitated by using a counter register instead of the general register as previously shown. This counter contains the address of the current microinstruction, and can be incremented 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**
o					
	CONT	X	XXX	XXXX	* * *
o					
	CONT	X	XXX	XXXX	* * *
o					
	CONT	X	XXX	XXXX	* * *


INSTRUCTION
LOAD
IR
OP CODE
OTHER
STATE SELECT
MAPPING PROM
TRI-STATE
TRI-STATE
V_{cc}
7
6
1
0
MUX
POLARITY
LOGIC
COUNTER
LOAD
CLOCK
2
1
3
MICRO MEMORY
NEXT ADDR SEL
POL
BRANCH COND SELECT
BRANCH ADDR
LOAD IR
REQUIRED CONTROLS
n
1
c
TO ALU

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.

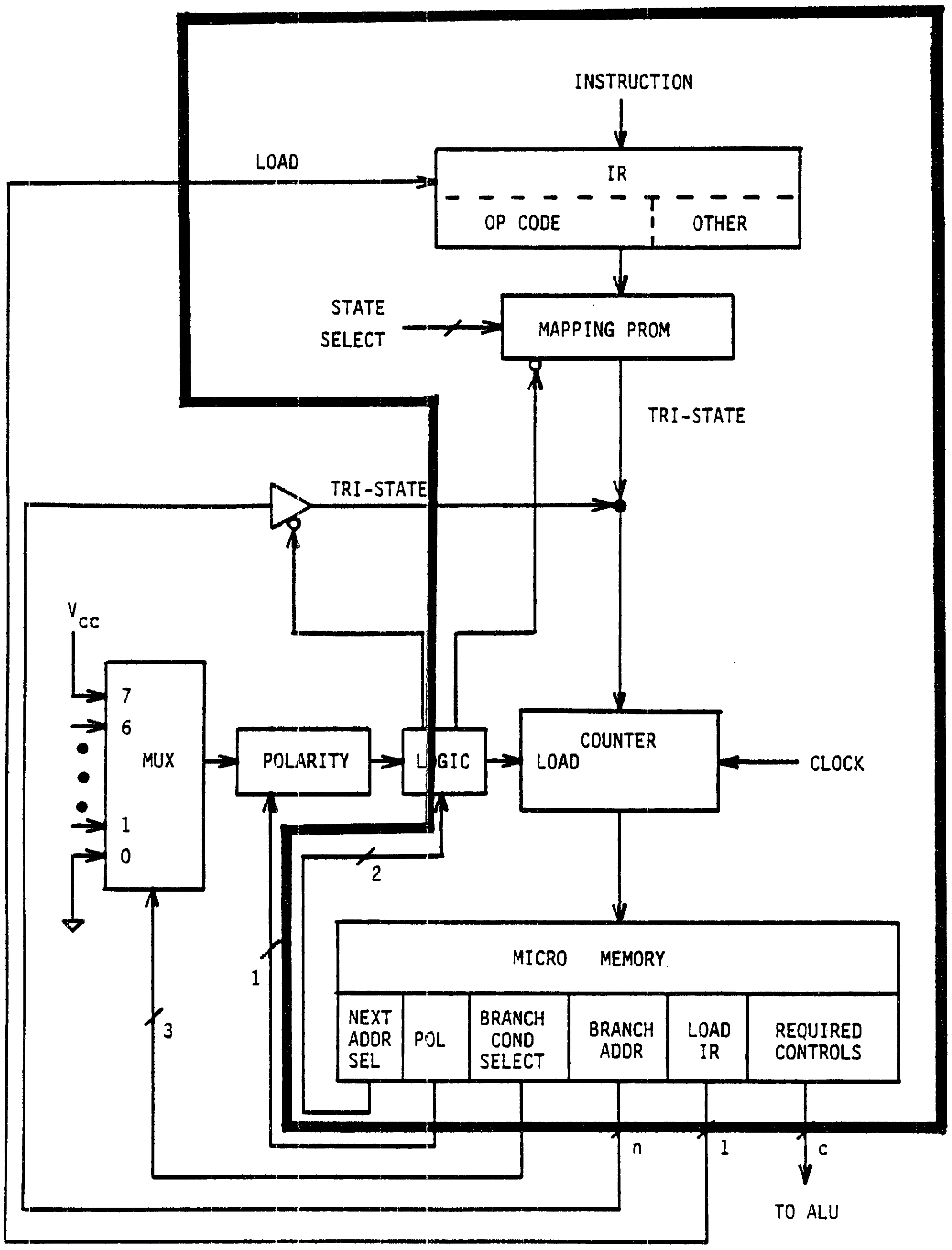
- First consider the interpretation of a new macro level instruction. Once the counter is loaded with a new microroutine starting address, each microinstruction in this microroutine sequence could have a "CONT" in the next address select field, except possibly for the last one.

  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".


INSTRUCTION
LOAD
IR
OP CODE
OTHER
STATE SELECT
MAPPING PROM
TRI-STATE
TRI-STATE
Vcc
7
6
1
0
MUX
POLARITY
LOGIC
COUNTER
LOAD
CLOCK
2
1
3
MICRO MEMORY
NEXT ADDR SEL
POL
BRANCH COND SELECT
BRANCH ADDR
LOAD IR
REQUIRED CONTROLS
n
1
c
TO ALU

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

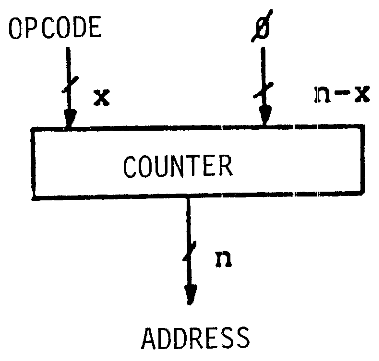
x bit opcode and n bit counter

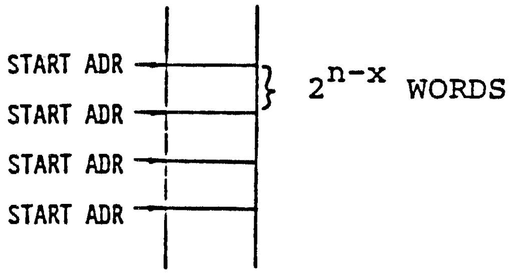
where

$x < n$

SOLUTION

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





EXAMPLE

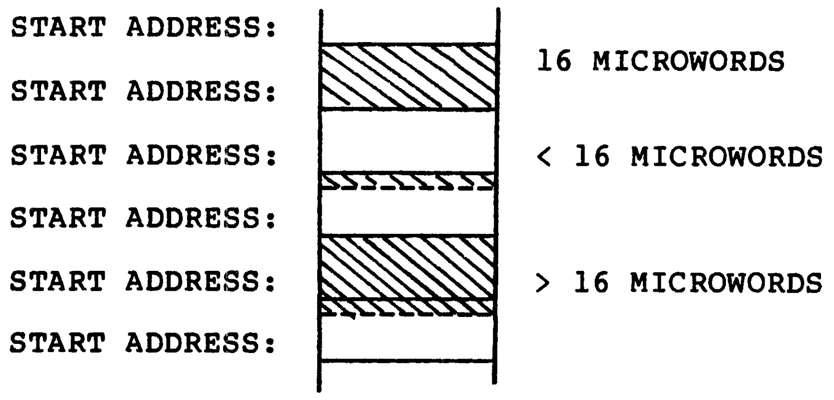
$x = 8$

$n = 12$

$n-x = 4$

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

Examine the micromemory:

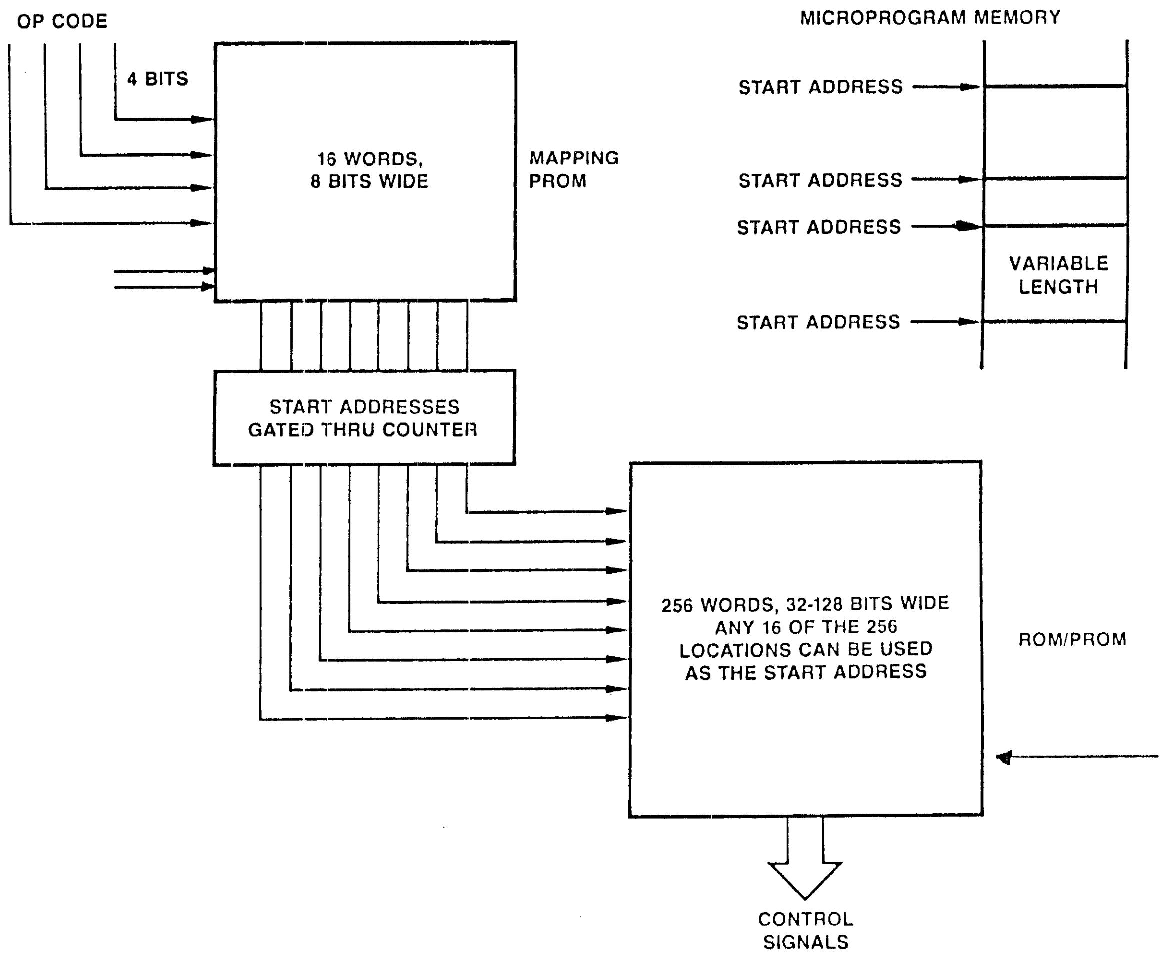


PROBLEM

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

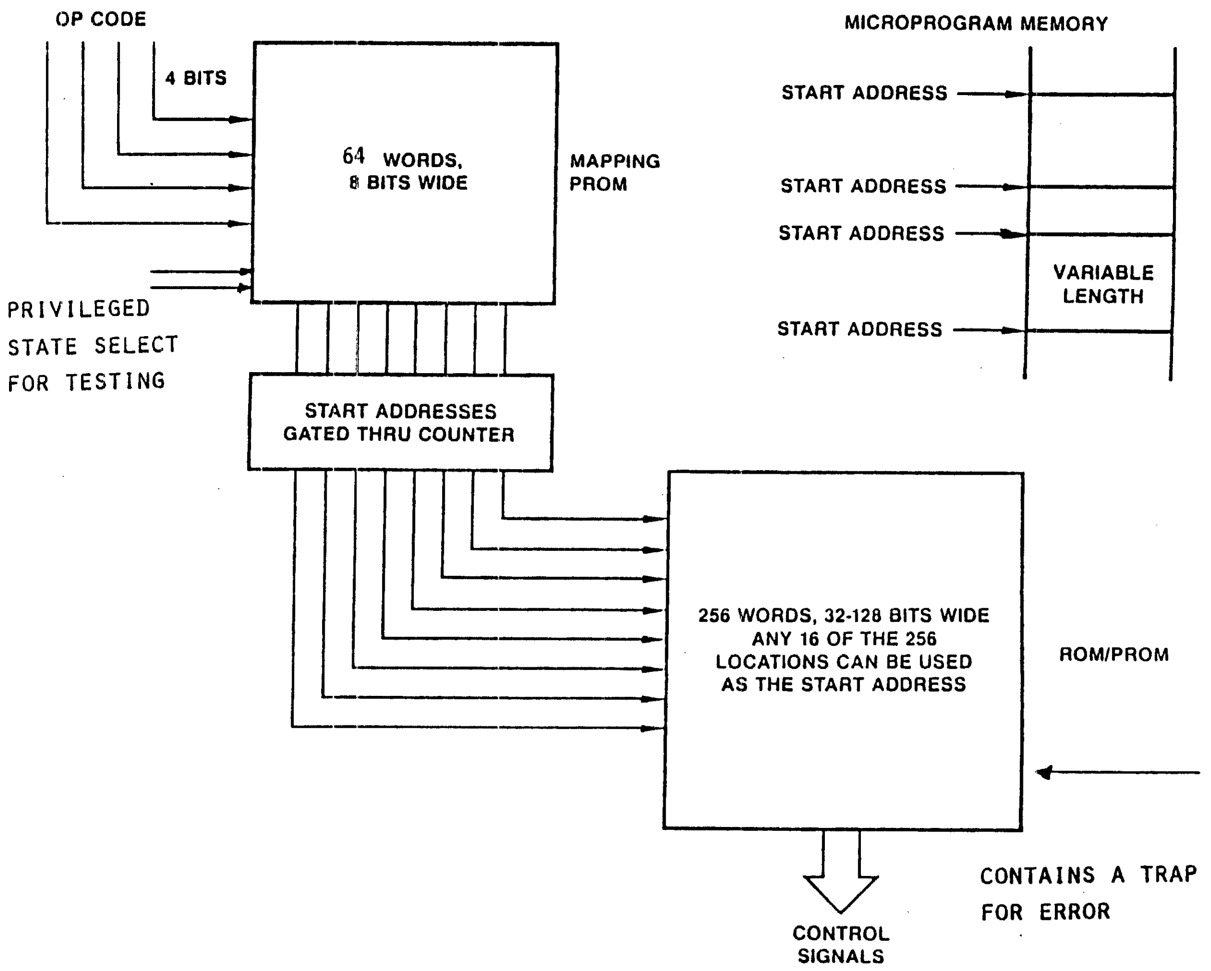
SOLUTION

- Add a micromemory address decoded (mapper)


OP CODE
4 BITS
16 WORDS, 8 BITS WIDE
MAPPING PROM
MICROPROGRAM MEMORY
START ADDRESS
START ADDRESS
START ADDRESS
VARIABLE LENGTH
START ADDRESS
START ADDRESSES GATED THRU COUNTER
256 WORDS, 32-128 BITS WIDE ANY 16 OF THE 256 LOCATIONS CAN BE USED AS THE START ADDRESS
ROM/PROM
CONTROL SIGNALS

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.


OP CODE
4 BITS
64 WORDS, 8 BITS WIDE
MAPPING PROM
PRIVILEGED STATE SELECT FOR TESTING
START ADDRESSES GATED THRU COUNTER
MICROPROGRAM MEMORY
START ADDRESS
START ADDRESS
START ADDRESS
VARIABLE LENGTH
START ADDRESS
256 WORDS, 32-128 BITS WIDE ANY 16 OF THE 256 LOCATIONS CAN BE USED AS THE START ADDRESS
ROM/PROM
CONTAINS A TRAP FOR ERROR
CONTROL SIGNALS

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.

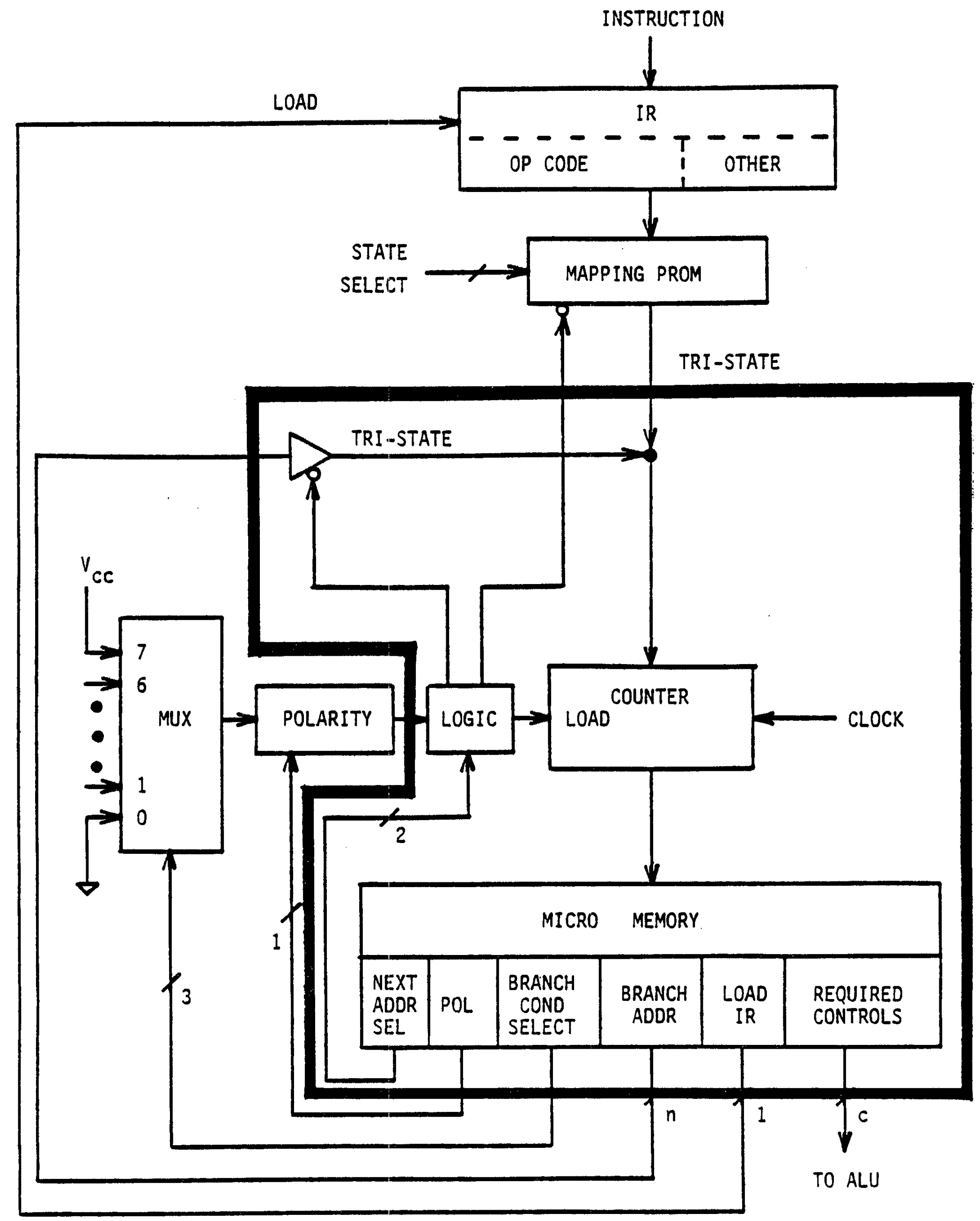
- The next address select field would carry a bit pattern to
  - enable the counter load control
  - enable the tri-state gates from the microword

    branch address field

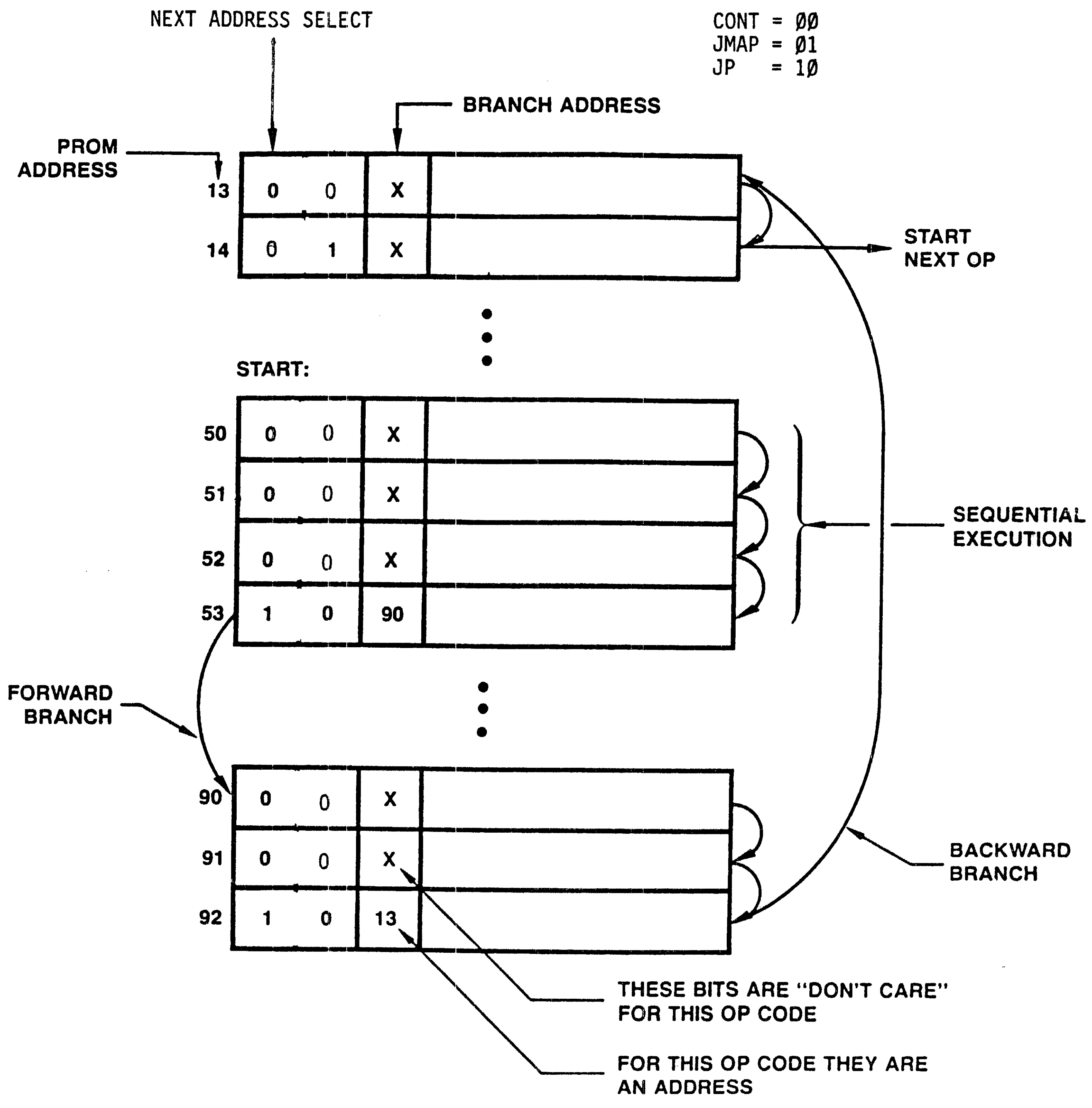
- The mnemonic **"JP"** is used for this next address operation

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	* * *


INSTRUCTION
LOAD
IR
OP CODE
OTHER
STATE SELECT
MAPPING PROM
TRI-STATE
TRI-STATE
Vcc
7
6
1
0
MUX
POLARITY
LOGIC
COUNTER
LOAD
CLOCK
2
1
3
MICRO MEMORY
NEXT ADDR SEL
POL
BRANCH COND SELECT
BRANCH ADDR
LOAD IR
REQUIRED CONTROLS
n
1
c
TO ALU

EXAMPLE - JP



NEXT ADDRESS SELECT
CONT = ØØ
JMAP = Ø1
JP = 1Ø
BRANCH ADDRESS
PROM ADDRESS
13 0 0 X
14 0 1 X
START NEXT OP
START:
50 0 0 X
51 0 0 X
52 0 0 X
53 1 0 90
SEQUENTIAL EXECUTION
FORWARD BRANCH
90 0 0 X
91 0 0 X
92 1 0 13
BACKWARD BRANCH
THESE BITS ARE "DON'T CARE" FOR THIS OP CODE
FOR THIS OP CODE THEY ARE AN ADDRESS

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

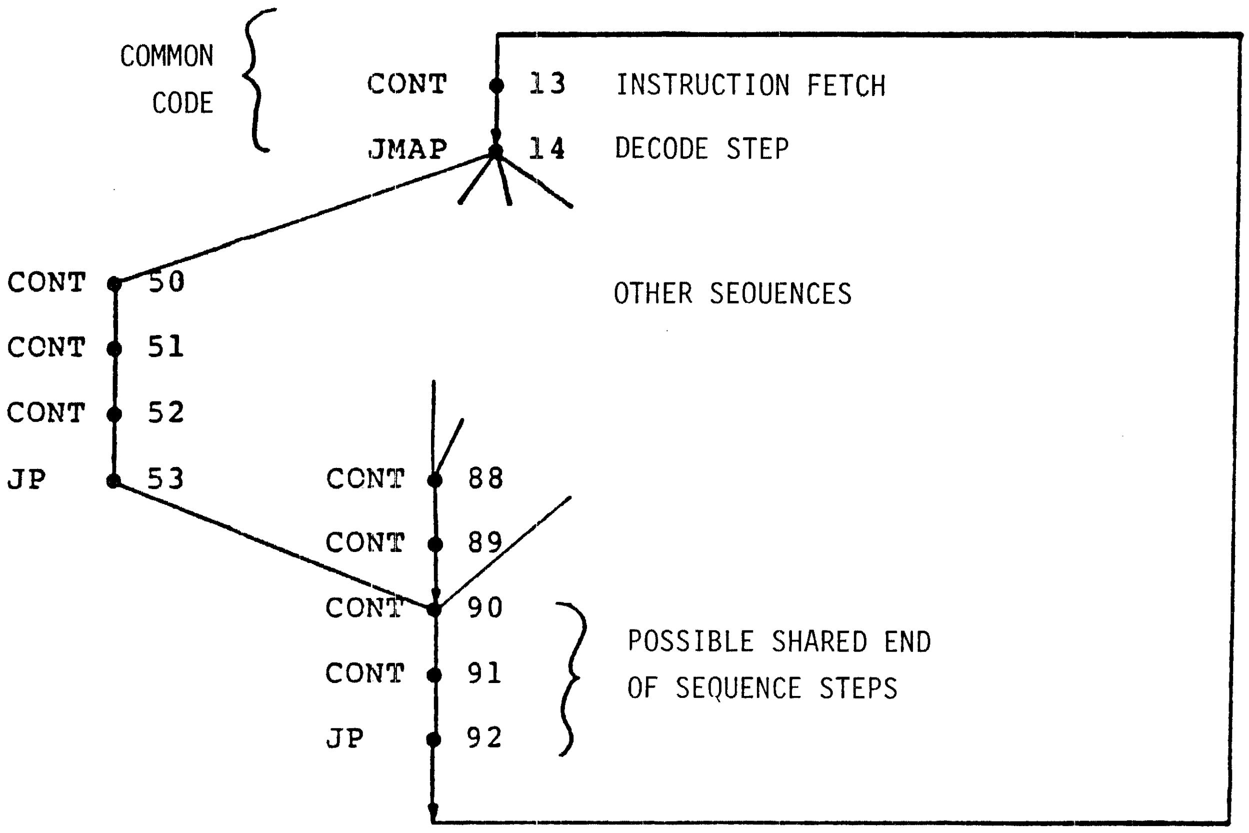
    - Note that the branch address field values are don't care

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 = Ø; a different routine

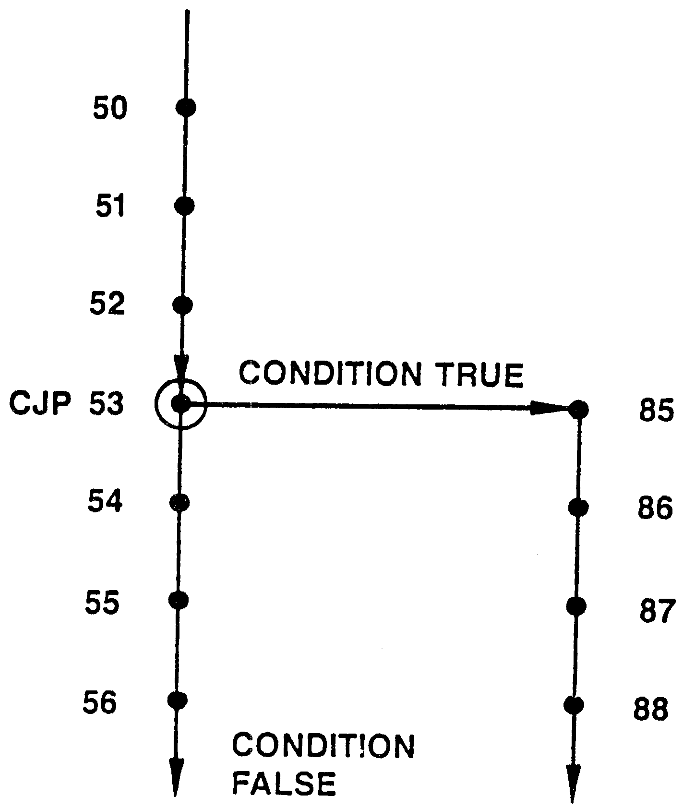
OTHER TESTABLE CONDITIONS MAY INCLUDE:

logical expression			mnemonic
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)


50
51
52
CJP 53
CONDITION TRUE
54
55
56
CONDITION FALSE
85
86
87
88

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

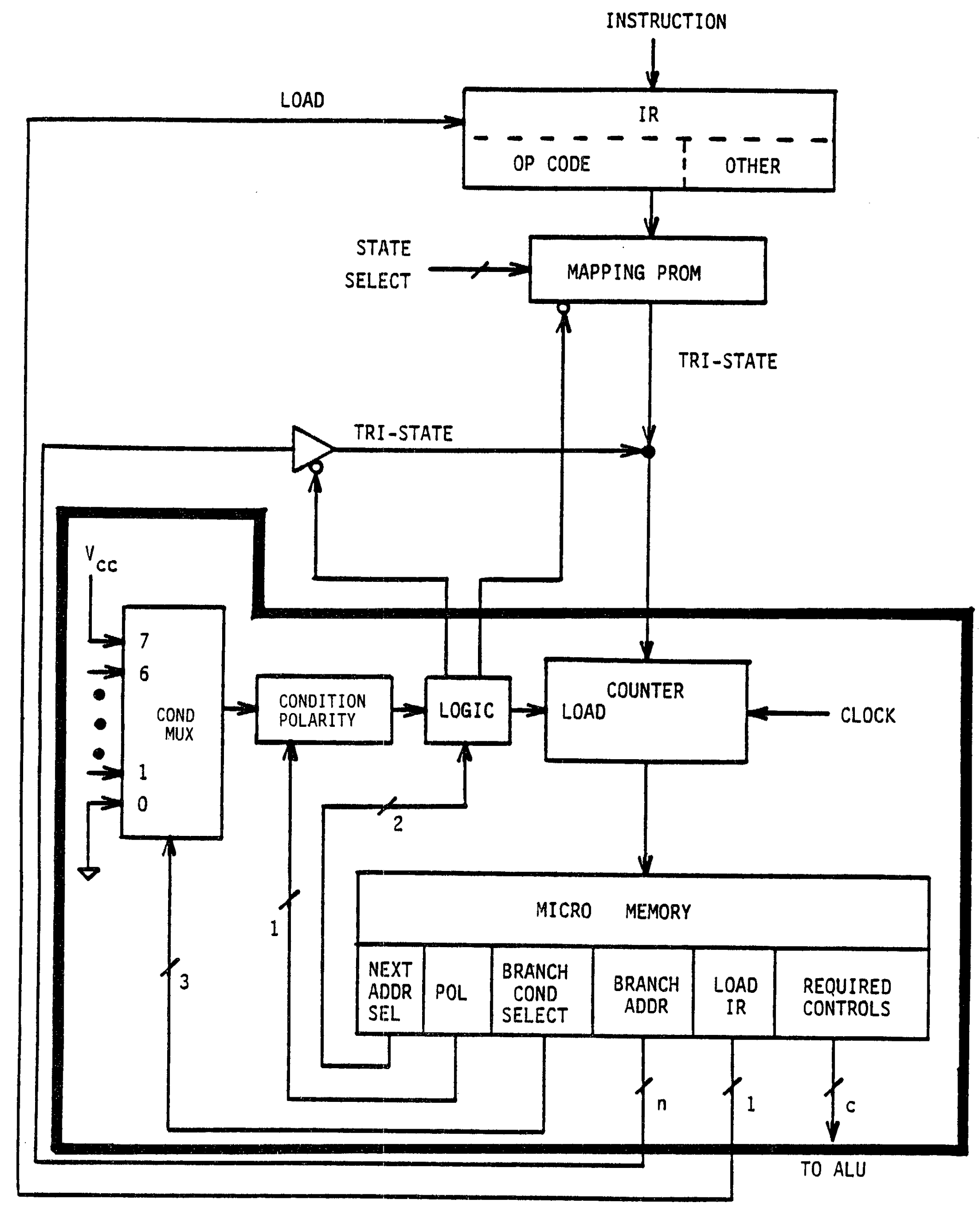
  - **ELSE** (condition code FALSE)

    (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".


INSTRUCTION
LOAD
IR
OP CODE
OTHER
STATE SELECT
MAPPING PROM
TRI-STATE
TRI-STATE
Vcc
7
6
1
0
COND MUX
CONDITION POLARITY
LOGIC
COUNTER
LOAD
CLOCK
2
1
3
MICRO MEMORY
NEXT ADDR SEL
POL
BRANCH COND SELECT
BRANCH ADDR
LOAD IR
REQUIRED CONTROLS
n
1
c
TO ALU

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**

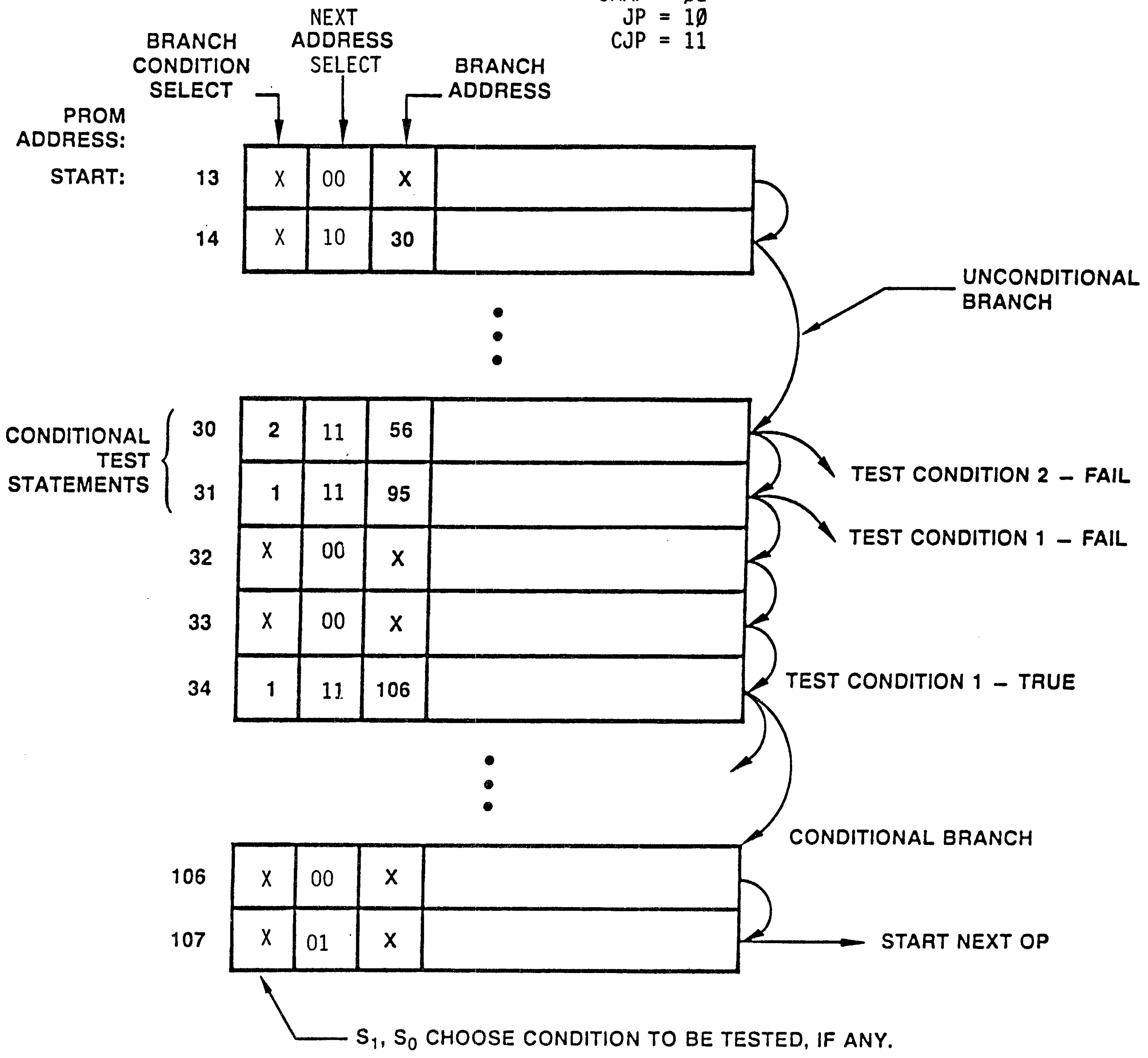
- There are three fields which are important

  - (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 = ØØ
JMAP = Ø1
  JP = 1Ø
 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

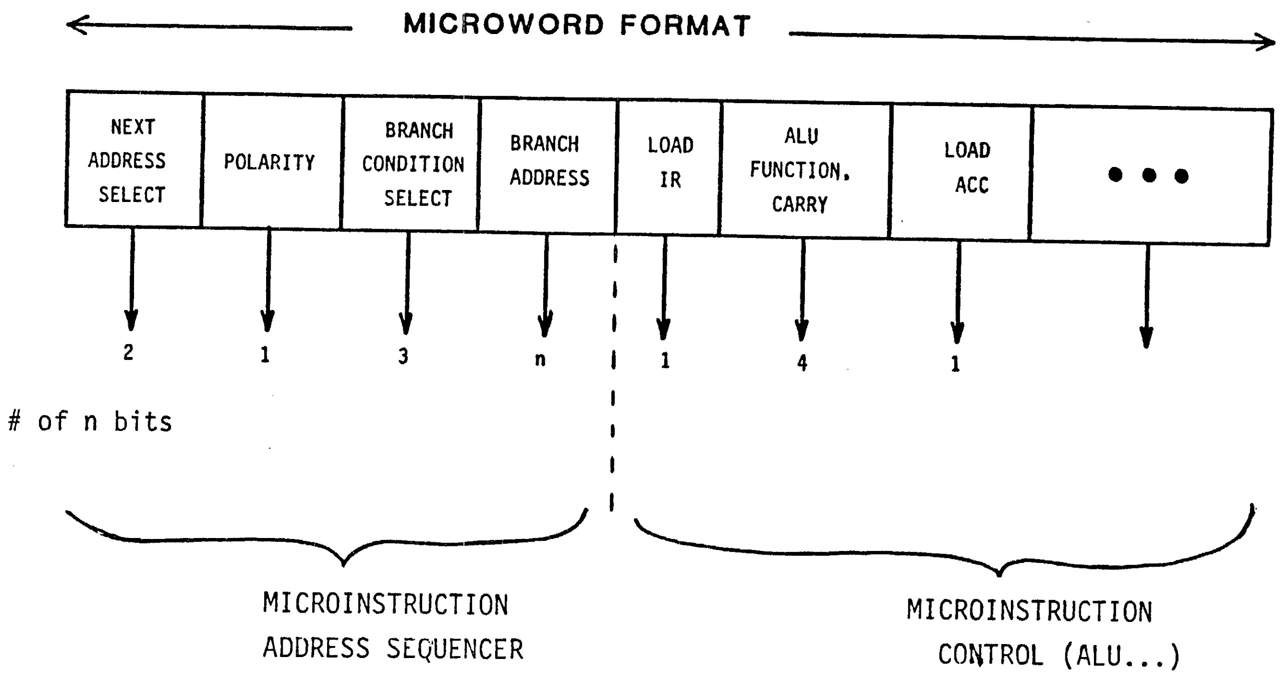
106) CONT

107) JMAP - unconditional jump

 select mapping PROM output

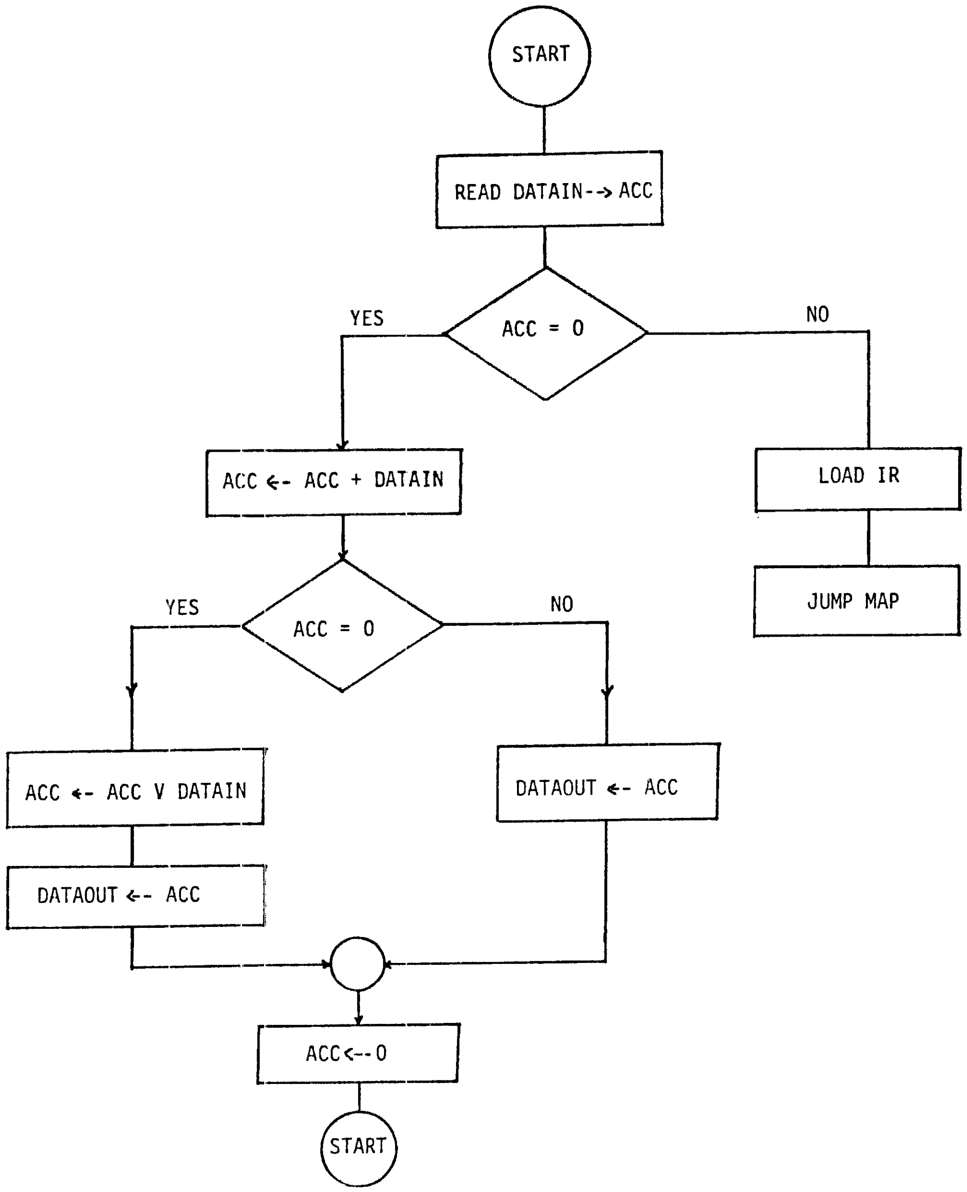
CLASS EXERCISE: MICRO-PROGRAM CONTROL

- 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

FLOW (MM ADDR*)	NEXT ADDR*	POL	COND SEL	BRANCH ADDR*	CONTROL (REGISTER TRANSFER LANGUAGE)
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

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 (microcyle).

- 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)

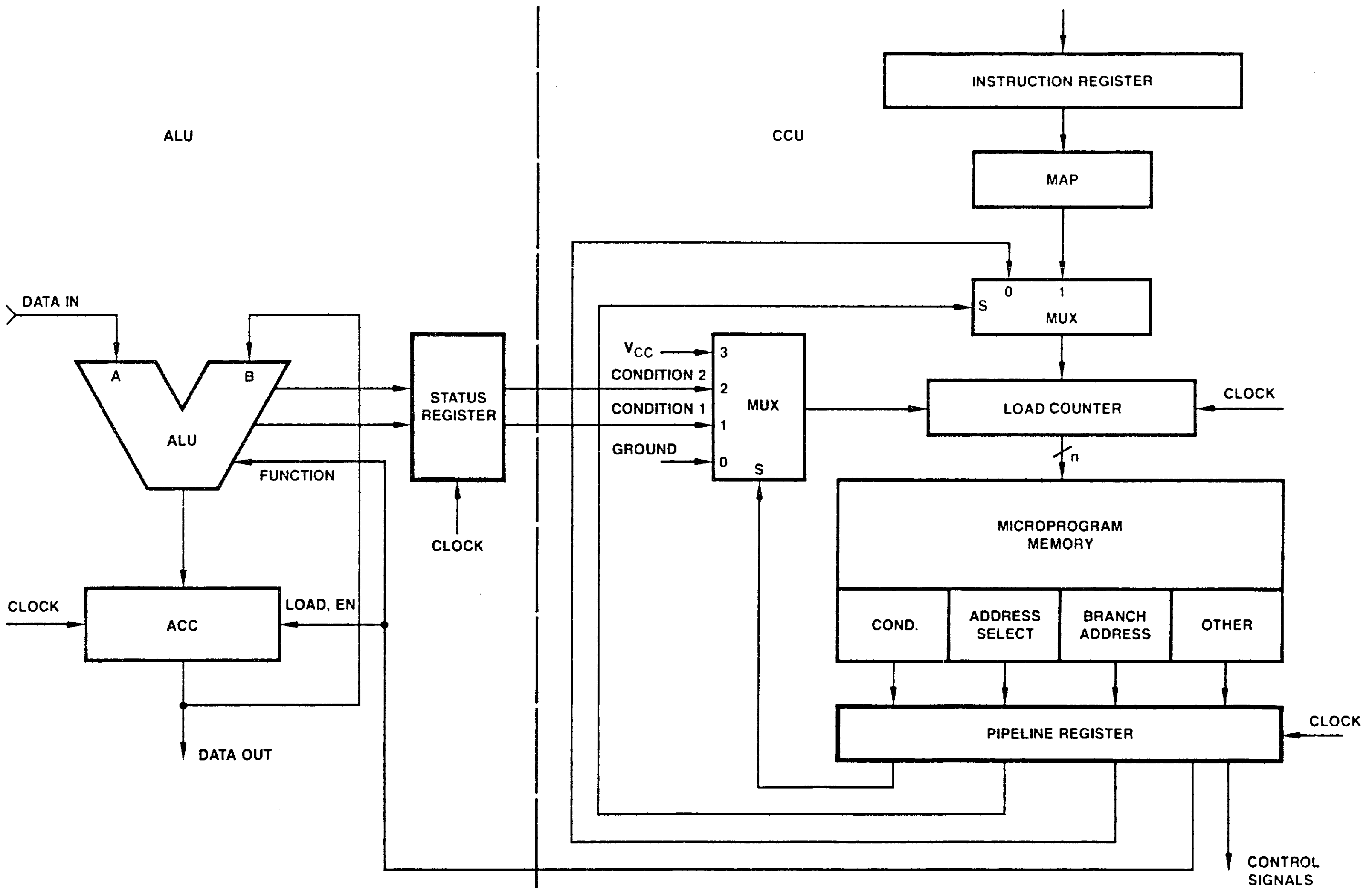
  - in parallel with the above, time through the MUX and set-up time for the counter load

  $t_{CP} = t_{CL \text{ to output}} + t_{\text{read access}} + t_{\text{set-up}}$

  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


ALU
CCU
INSTRUCTION REGISTER
MAP
0
1
S
MUX
DATA IN
A
B
ALU
FUNCTION
STATUS REGISTER
CLOCK
VCC
CONDITION 2
CONDITION 1
GROUND
3
2
1
0
MUX
S
LOAD COUNTER
CLOCK
n
MICROPROGRAM MEMORY
COND.
ADDRESS SELECT
BRANCH ADDRESS
OTHER
PIPELINE REGISTER
CLOCK
CLOCK
ACC
LOAD, EN
DATA OUT
CONTROL SIGNALS

TIMING COMPUTATION

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

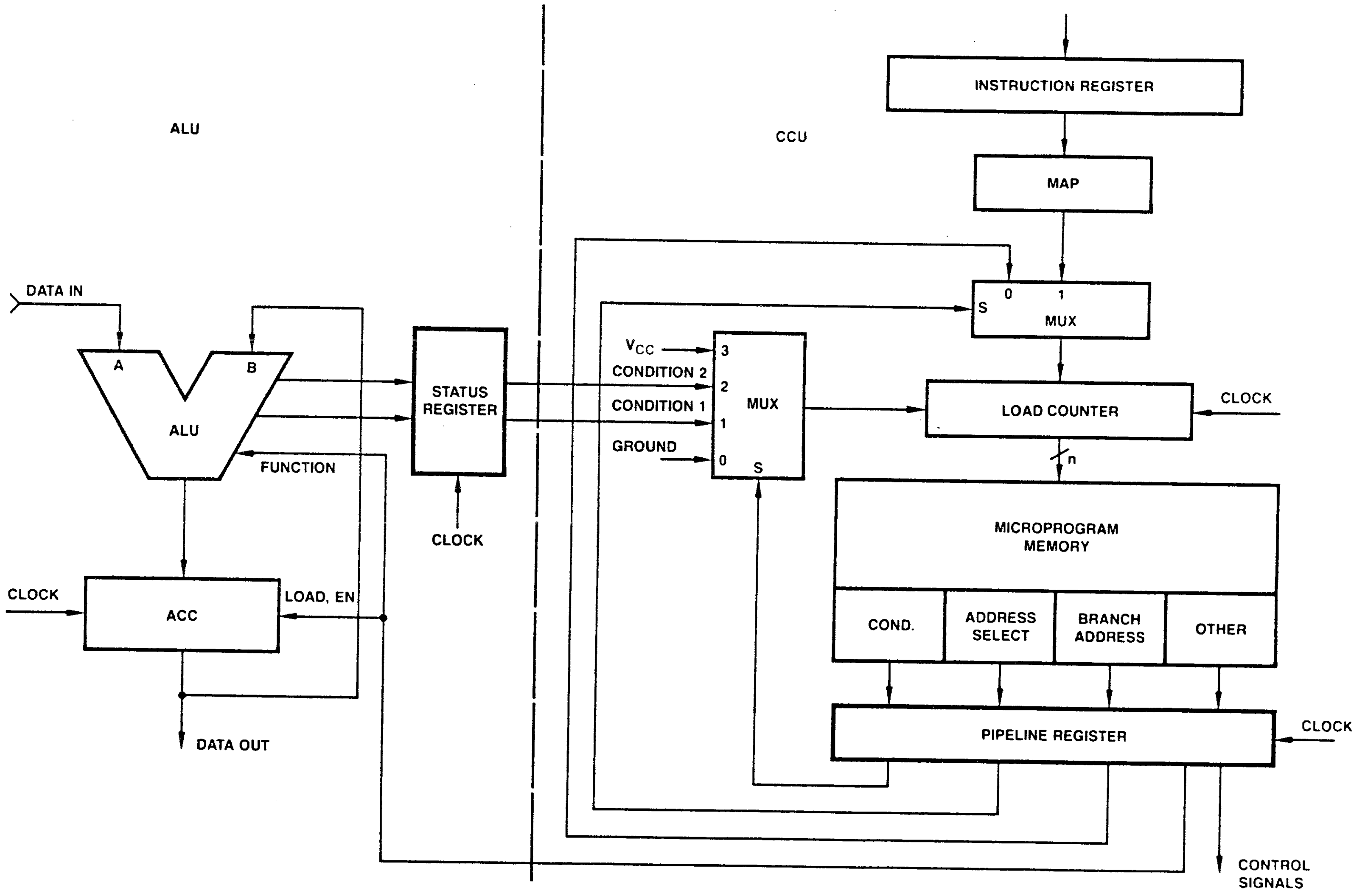
First:

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

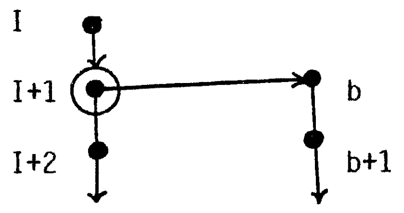

ALU
CCU
DATA IN
A
B
ALU
FUNCTION
STATUS REGISTER
CLOCK
CLOCK
ACC
LOAD, EN
DATA OUT
VCC
CONDITION 2
CONDITION 1
GROUND
3
2
1
0
MUX
S
INSTRUCTION REGISTER
MAP
0
1
S
MUX
LOAD COUNTER
CLOCK
n
MICROPROGRAM MEMORY
COND.
ADDRESS SELECT
BRANCH ADDRESS
OTHER
PIPELINE REGISTER
CLOCK
CONTROL SIGNALS

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.

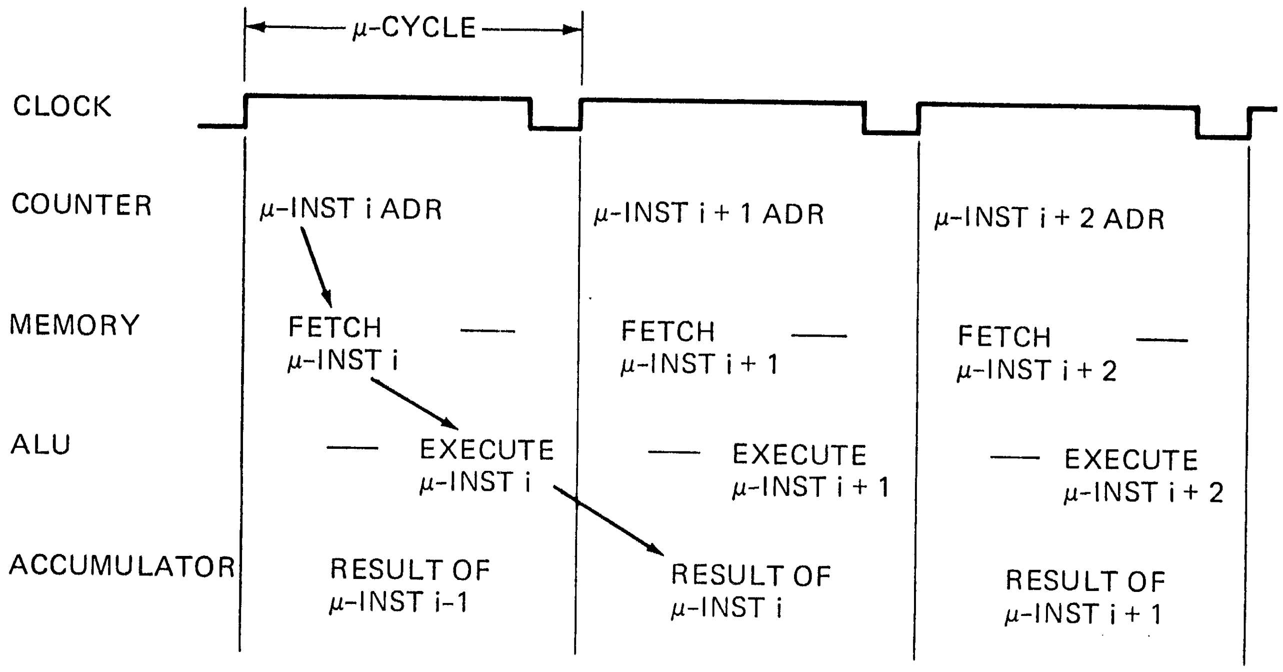
CONDITIONAL BRANCH (CJP)



- 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.

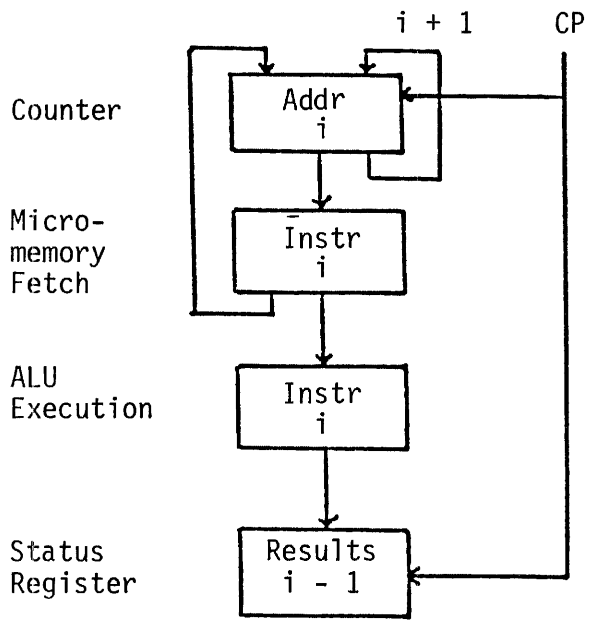
- Note also that there is no difference in flow when the branch is taken as shown in the BRANCH TAKEN diagram.

No Branch

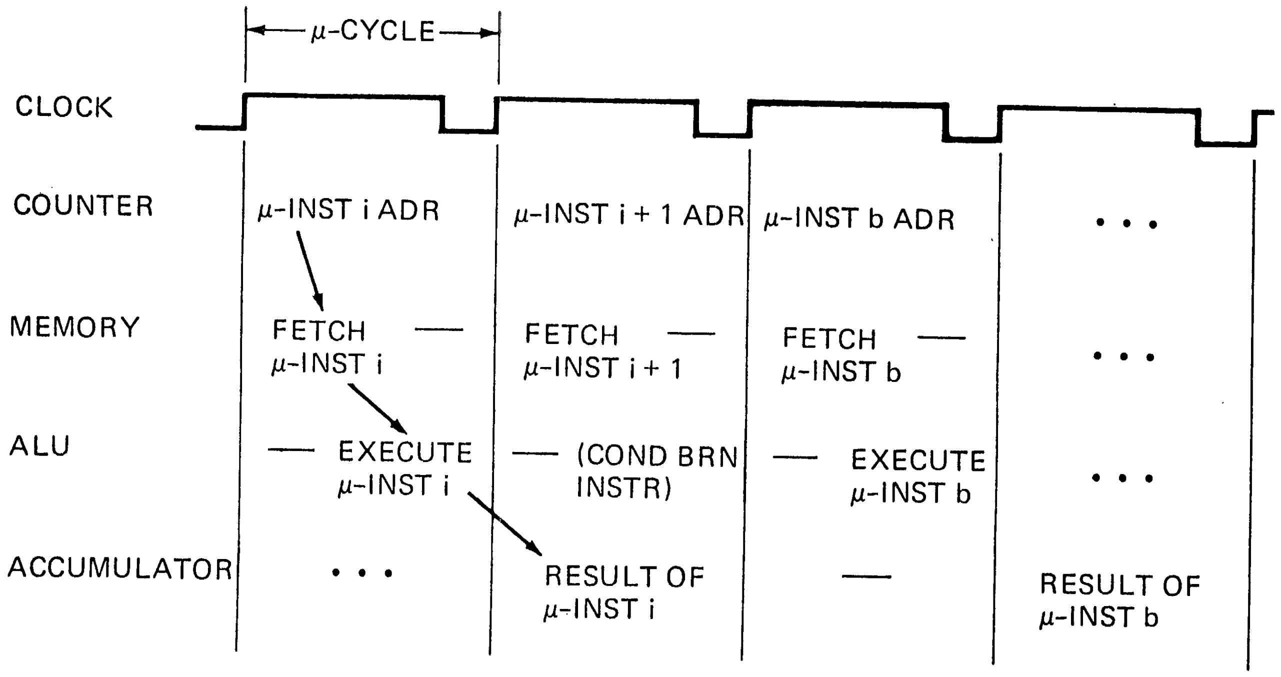

μ-CYCLE
CLOCK
COUNTER
MEMORY
ALU
ACCUMULATOR
μ-INST i ADR
μ-INST i + 1 ADR
μ-INST i + 2 ADR
FETCH μ-INST i
FETCH μ-INST i + 1
FETCH μ-INST i + 2
EXECUTE μ-INST i
EXECUTE μ-INST i + 1
EXECUTE μ-INST i + 2
RESULT OF μ-INST i-1
RESULT OF μ-INST i
RESULT OF μ-INST i + 1

Current Instruction Flow

(No Branch)

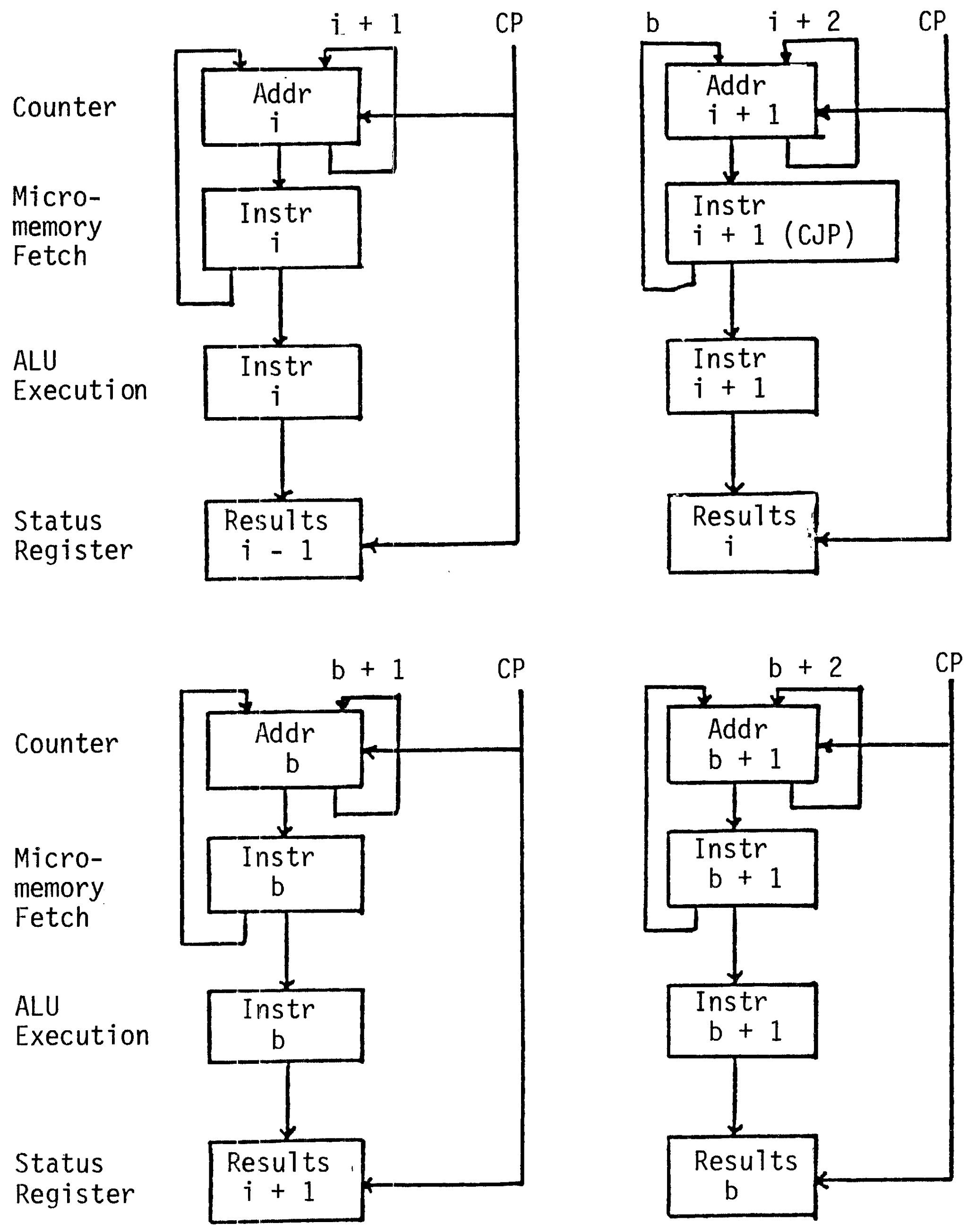


Branch Taken



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)

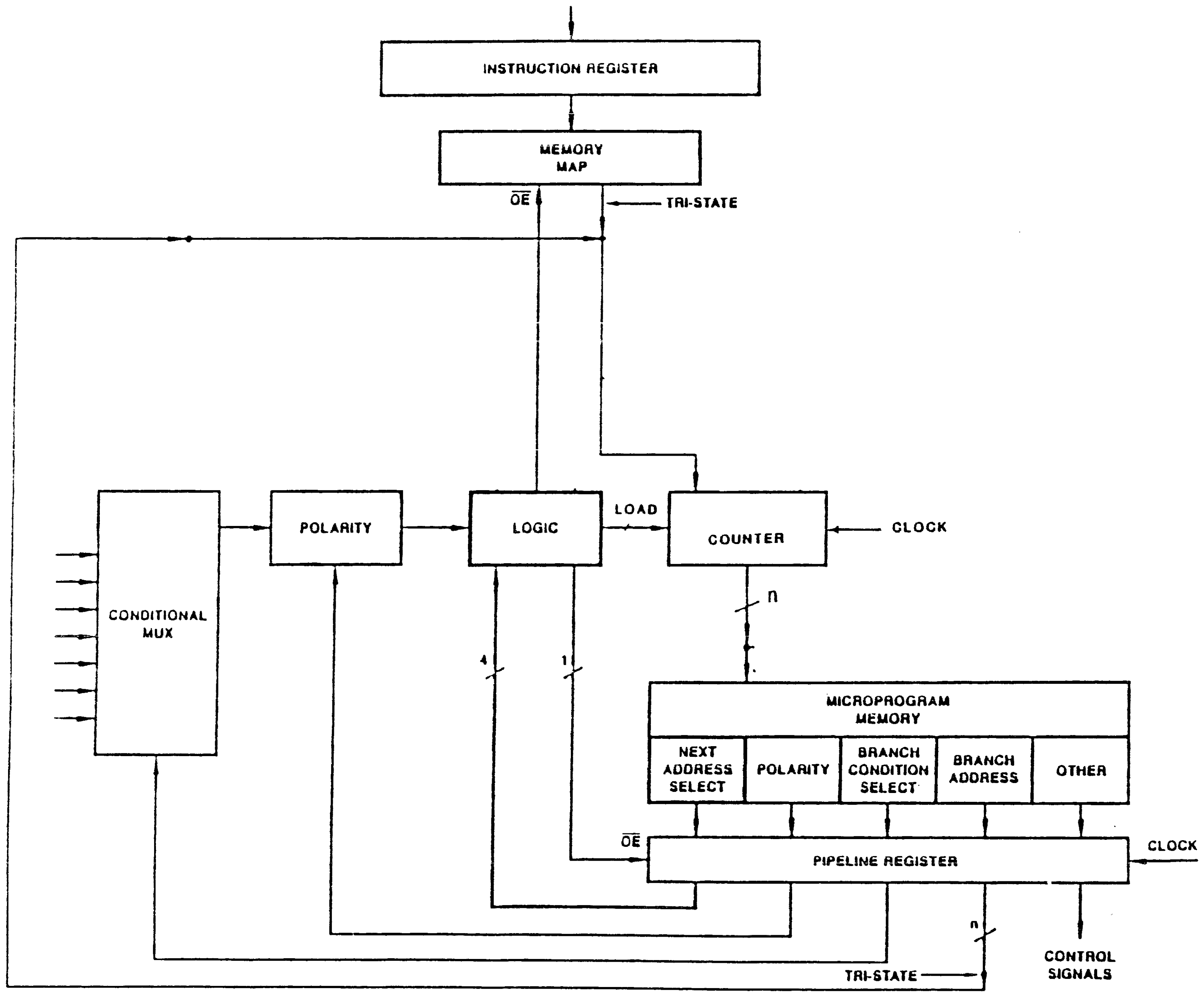
A SOLUTION

- 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.

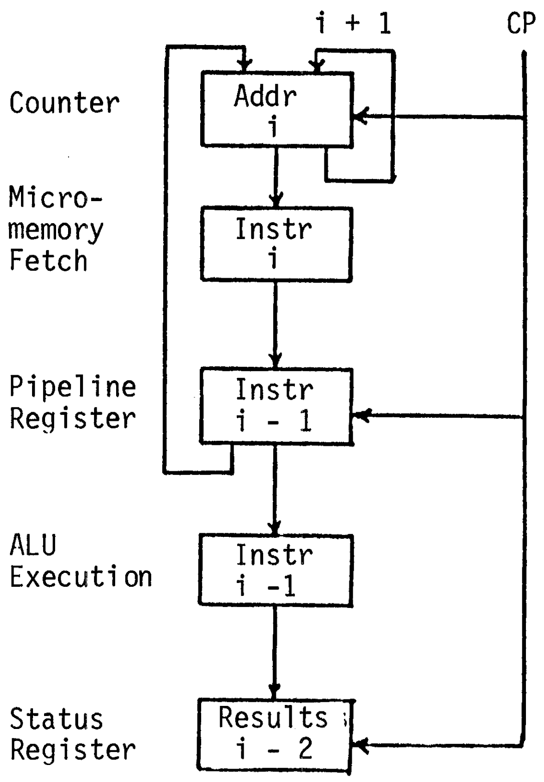
A two-level pipeline results in the current design with:

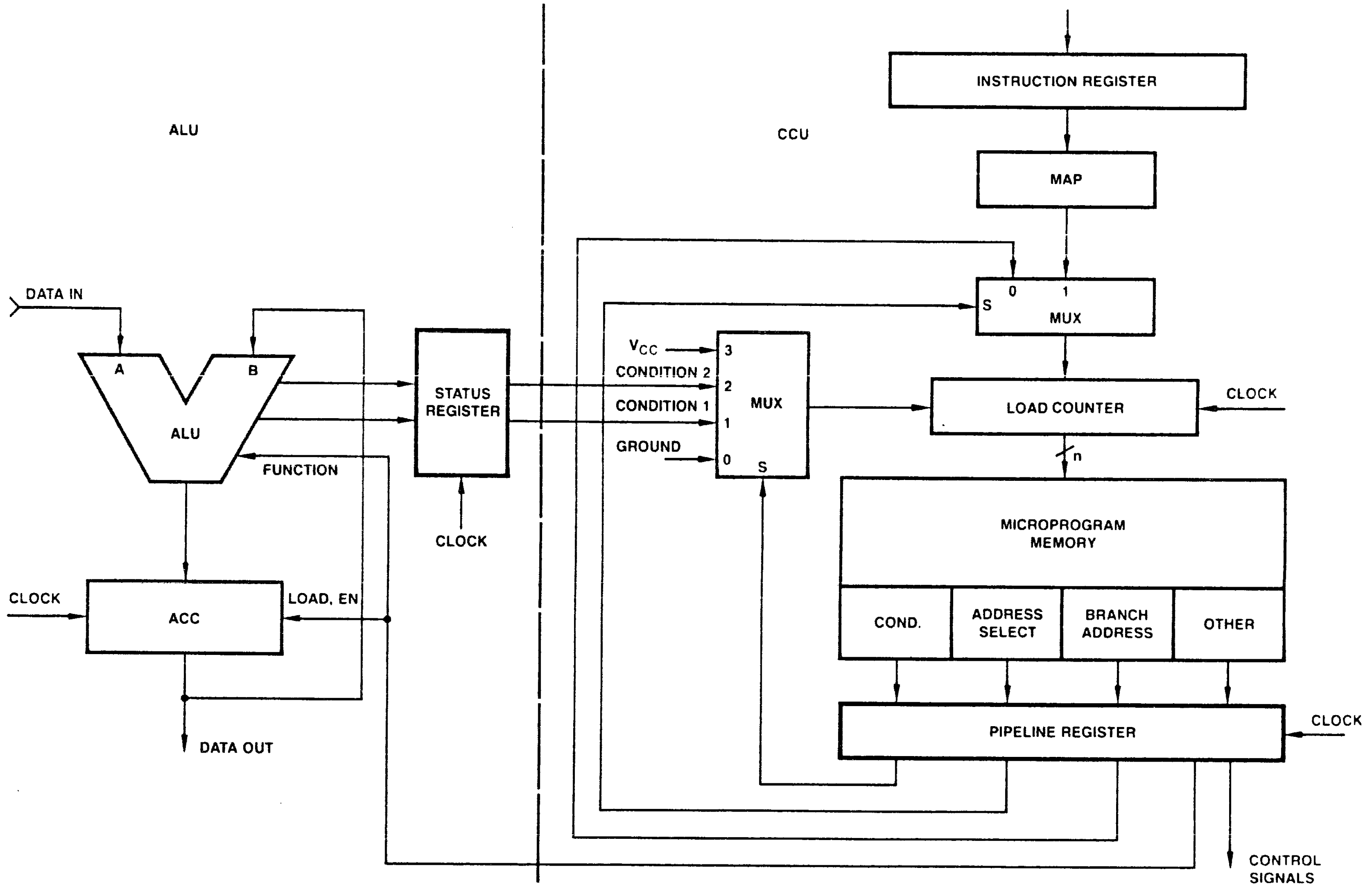
1) counter register

2) pipeline register

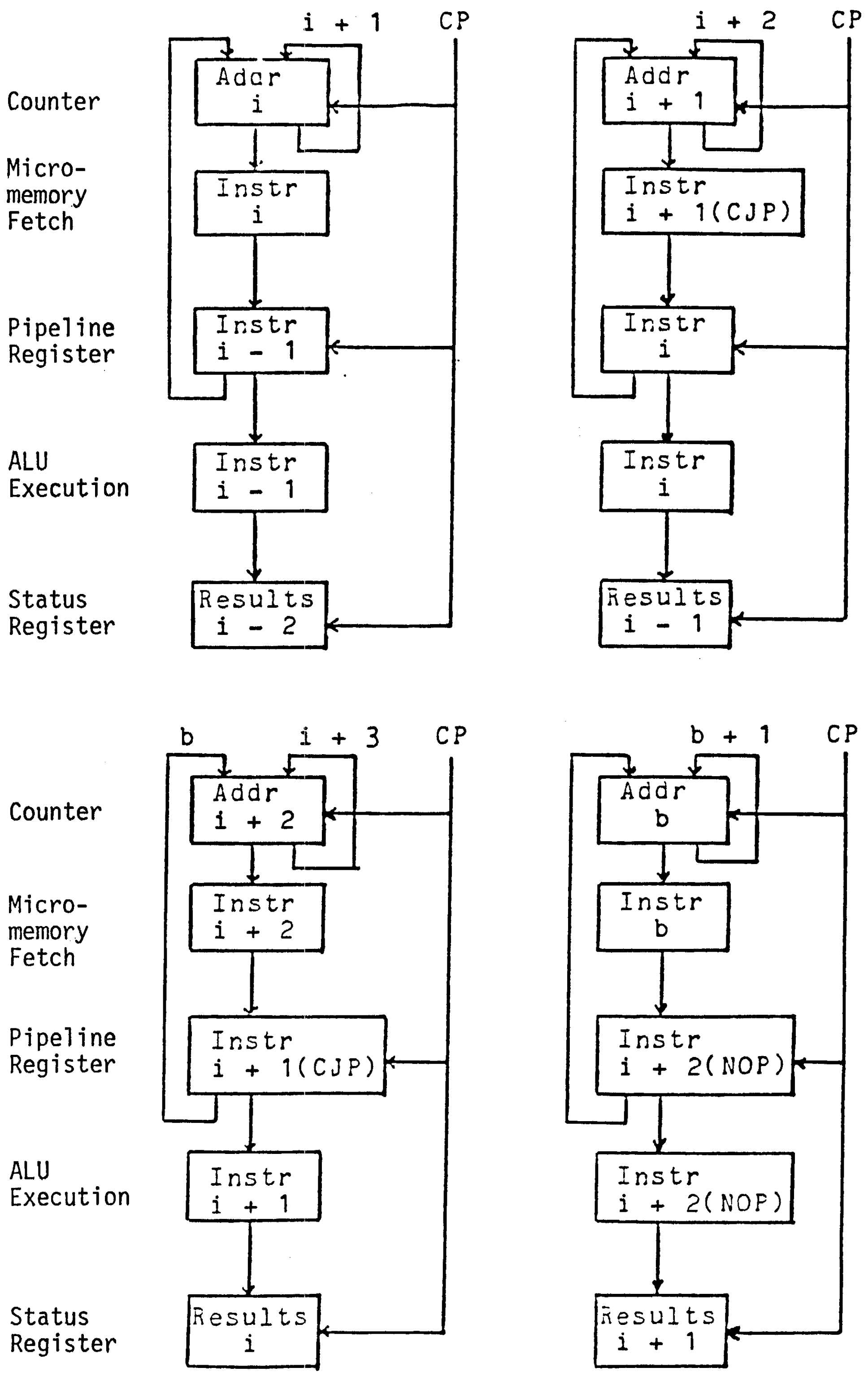

INSTRUCTION REGISTER
MEMORY MAP
$\overline{OE}$
TRI-STATE
CONDITIONAL MUX
POLARITY
LOGIC
LOAD
COUNTER
CLOCK
n
4
1
MICROPROGRAM MEMORY
NEXT ADDRESS SELECT
POLARITY
BRANCH CONDITION SELECT
BRANCH ADDRESS
OTHER
$\overline{OE}$
PIPELINE REGISTER
CLOCK
n
TRI-STATE
CONTROL SIGNALS

Pipeline Concept

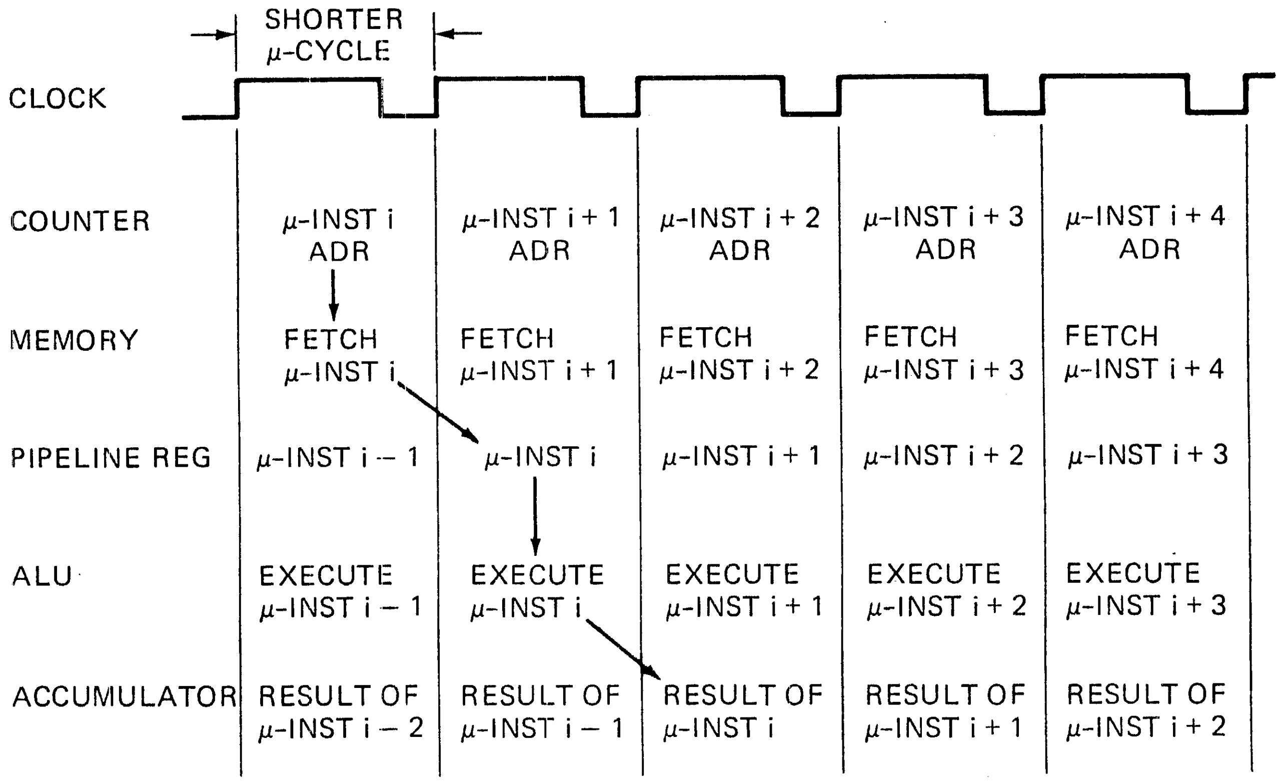



ALU
CCU
DATA IN
A
B
ALU
FUNCTION
STATUS REGISTER
CLOCK
CLOCK
ACC
LOAD, EN
DATA OUT
INSTRUCTION REGISTER
MAP
0
1
S
MUX
VCC
CONDITION 2
CONDITION 1
GROUND
3
2
1
0
MUX
S
LOAD COUNTER
CLOCK
n
MICROPROGRAM MEMORY
COND.
ADDRESS SELECT
BRANCH ADDRESS
OTHER
PIPELINE REGISTER
CLOCK
CONTROL SIGNALS

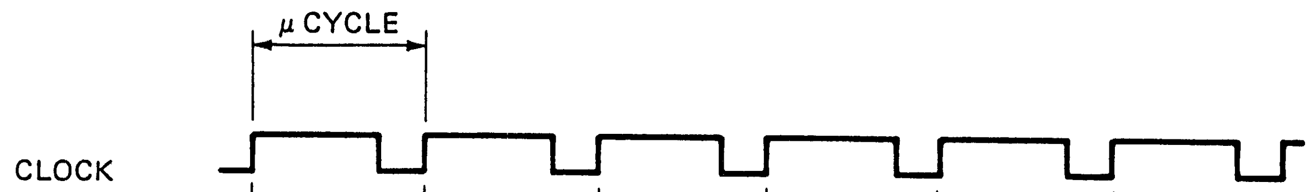
Pipeline with Branch Taken



No Branch


SHORTER
μ-CYCLE
CLOCK
COUNTER
MEMORY
PIPELINE REG
ALU
ACCUMULATOR
μ-INST i ADR
μ-INST i + 1 ADR
μ-INST i + 2 ADR
μ-INST i + 3 ADR
μ-INST i + 4 ADR
FETCH μ-INST i
FETCH μ-INST i + 1
FETCH μ-INST i + 2
FETCH μ-INST i + 3
FETCH μ-INST i + 4
μ-INST i − 1
μ-INST i
μ-INST i + 1
μ-INST i + 2
μ-INST i + 3
EXECUTE μ-INST i − 1
EXECUTE μ-INST i
EXECUTE μ-INST i + 1
EXECUTE μ-INST i + 2
EXECUTE μ-INST i + 3
RESULT OF μ-INST i − 2
RESULT OF μ-INST i − 1
RESULT OF μ-INST i
RESULT OF μ-INST i + 1
RESULT OF μ-INST i + 2

Branch Taken



COUNTER	µ INST i ADR	µ INST i + 1 ADR	µ INST i + 2 ADR	µ INSTR b ADR	µ INSTR b+1 ADR	µ INST b+2 ADR
MEMORY	FETCH µ INST i	FETCH µ INST i + 1	FETCH µ INST i + 2 *****	FETCH µ INST b	——	——
PIPELINE REG.	µ INST i − 1	µ INST i	µ INST i + 1	(HOLD)	µ INST b	——
ALU	EXECUTE µ INST i − 1	EXECUTE µ INST i	(COND BRAN INSTR)	(HOLD)	EXECUTE µ INST b	——
ACCUMULATOR	RESULT OF µ INST i − 2	RESULT OF µ INST i − 1	RESULT OF µ INST i	?		RESULT OF µ INST b

problem!

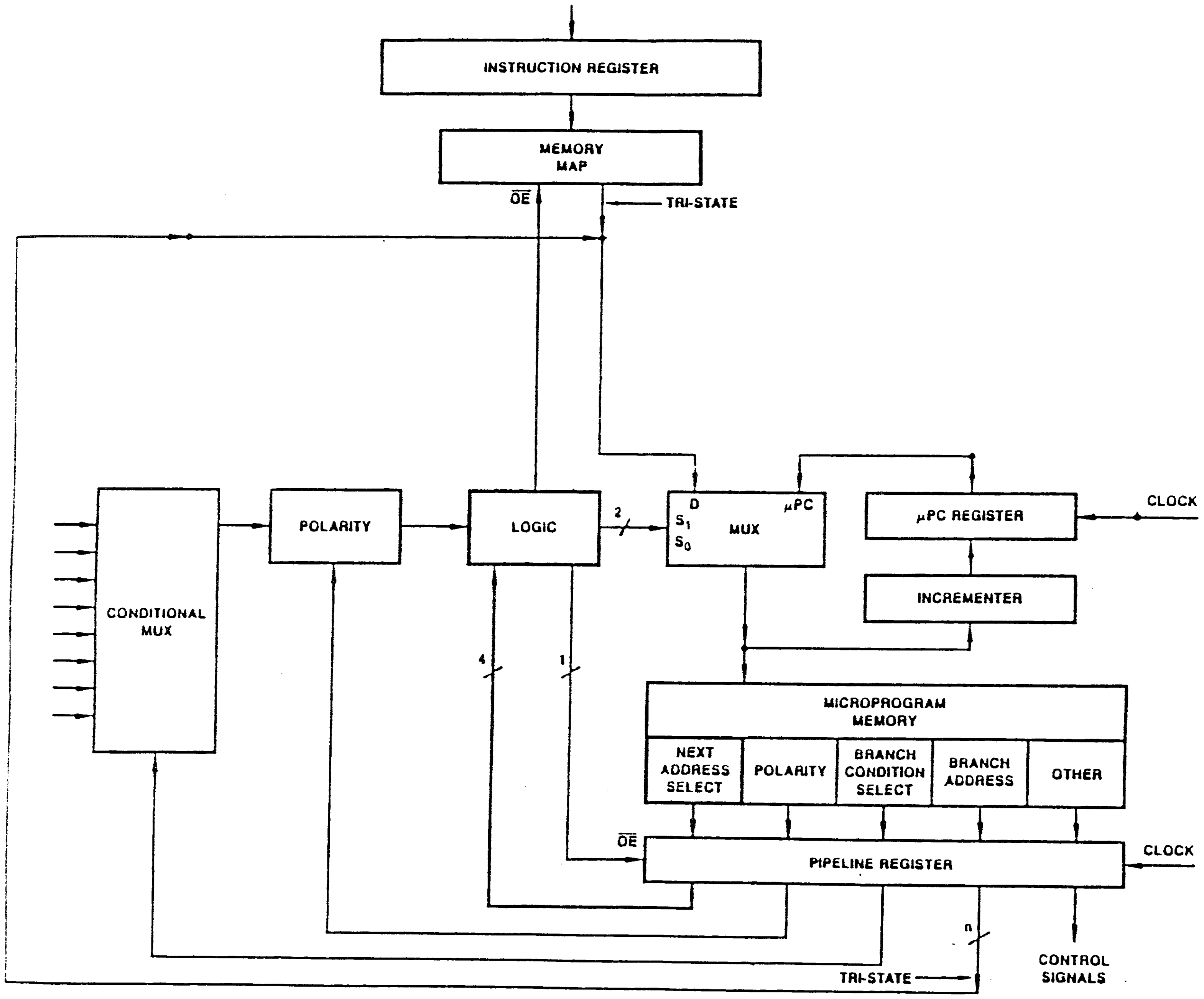
ADDITIONAL ARCHITECTURAL IMPROVEMENTS

- 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.

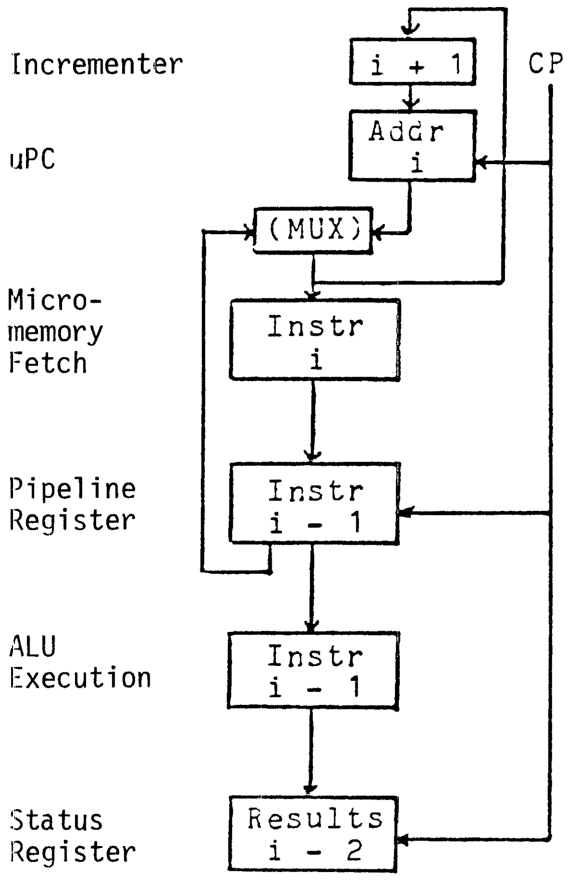
- A multiplexor is added to allow either the micro PC register or the tri-state bus to be selected as the address source to the micro memory.

- Note that the tri-state output on the pipeline is for the branch address field only.

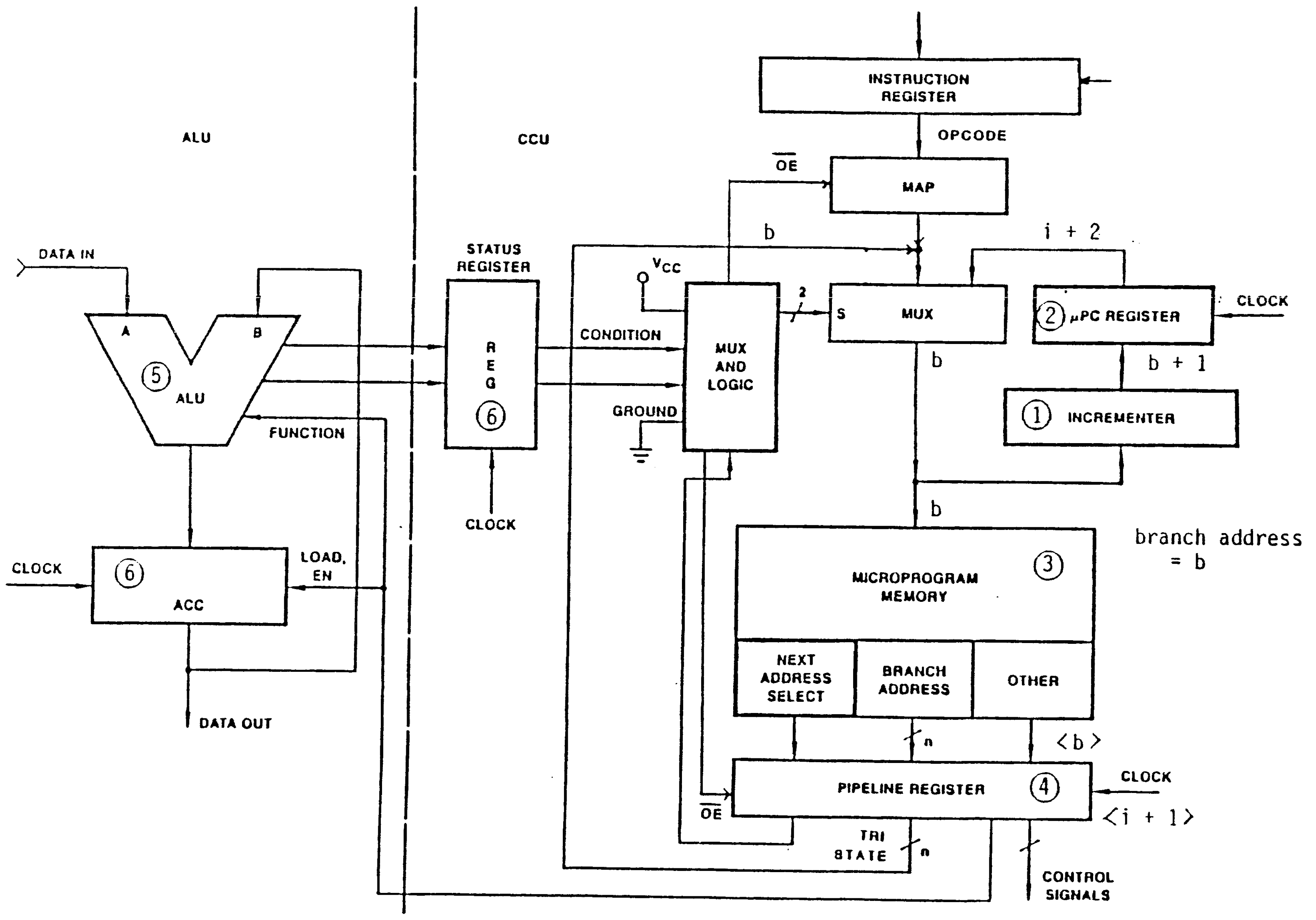
- This architectural change eliminates the problem of a lost cycle when the branch is taken and allows the controller to run at full speed all the time as shown in the following diagrams:


INSTRUCTION REGISTER
MEMORY MAP
OE
TRI-STATE
CONDITIONAL MUX
POLARITY
LOGIC
2
D
S1
S0
MUX
μPC
μPC REGISTER
CLOCK
INCREMENTER
4
1
MICROPROGRAM MEMORY
NEXT ADDRESS SELECT
POLARITY
BRANCH CONDITION SELECT
BRANCH ADDRESS
OTHER
OE
PIPELINE REGISTER
CLOCK
n
TRI-STATE
CONTROL SIGNALS

Pipeline Concept with Incrementer

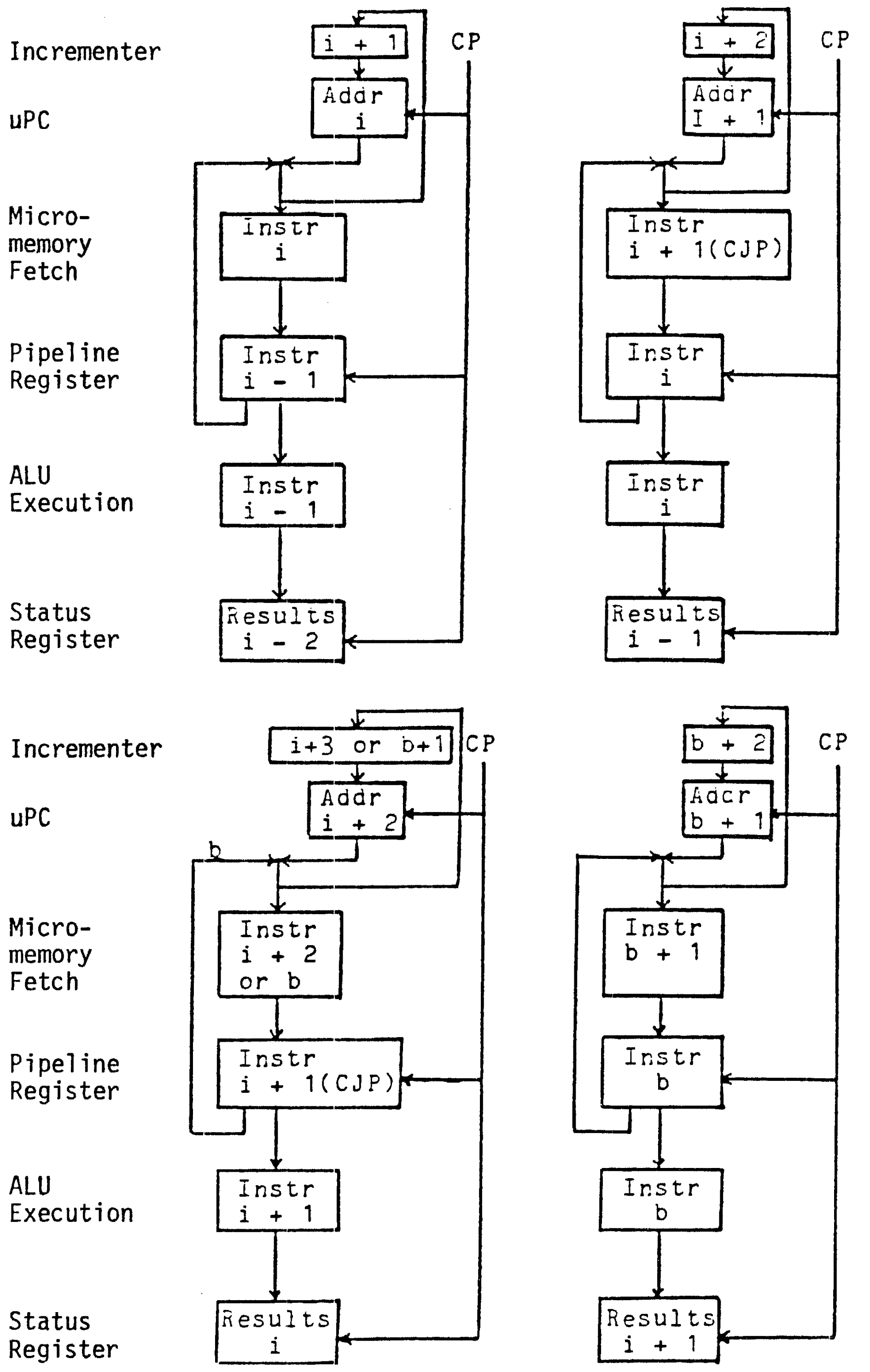


(revised)

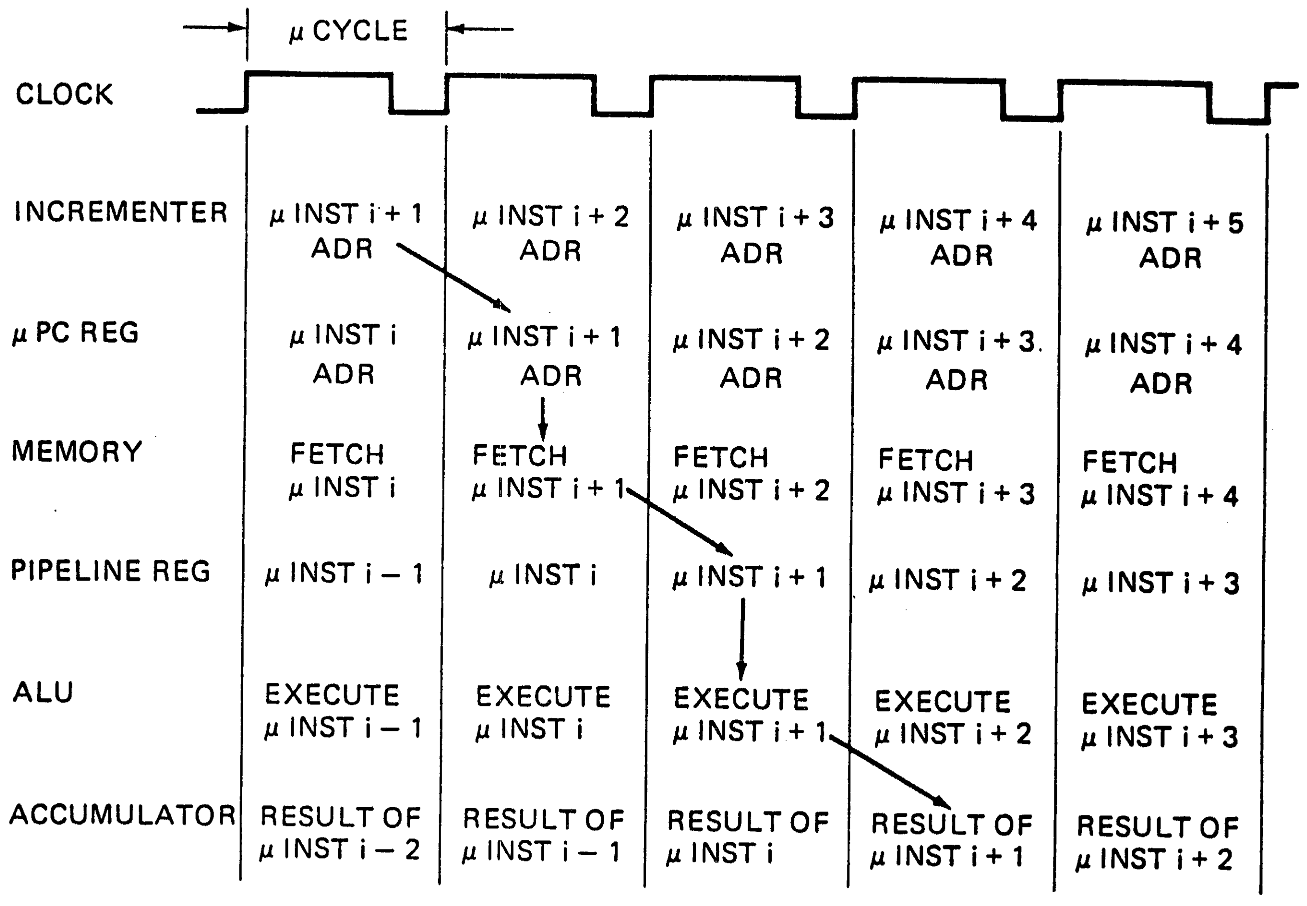


TIMING

Pipeline with Incrementer - Branch Taken

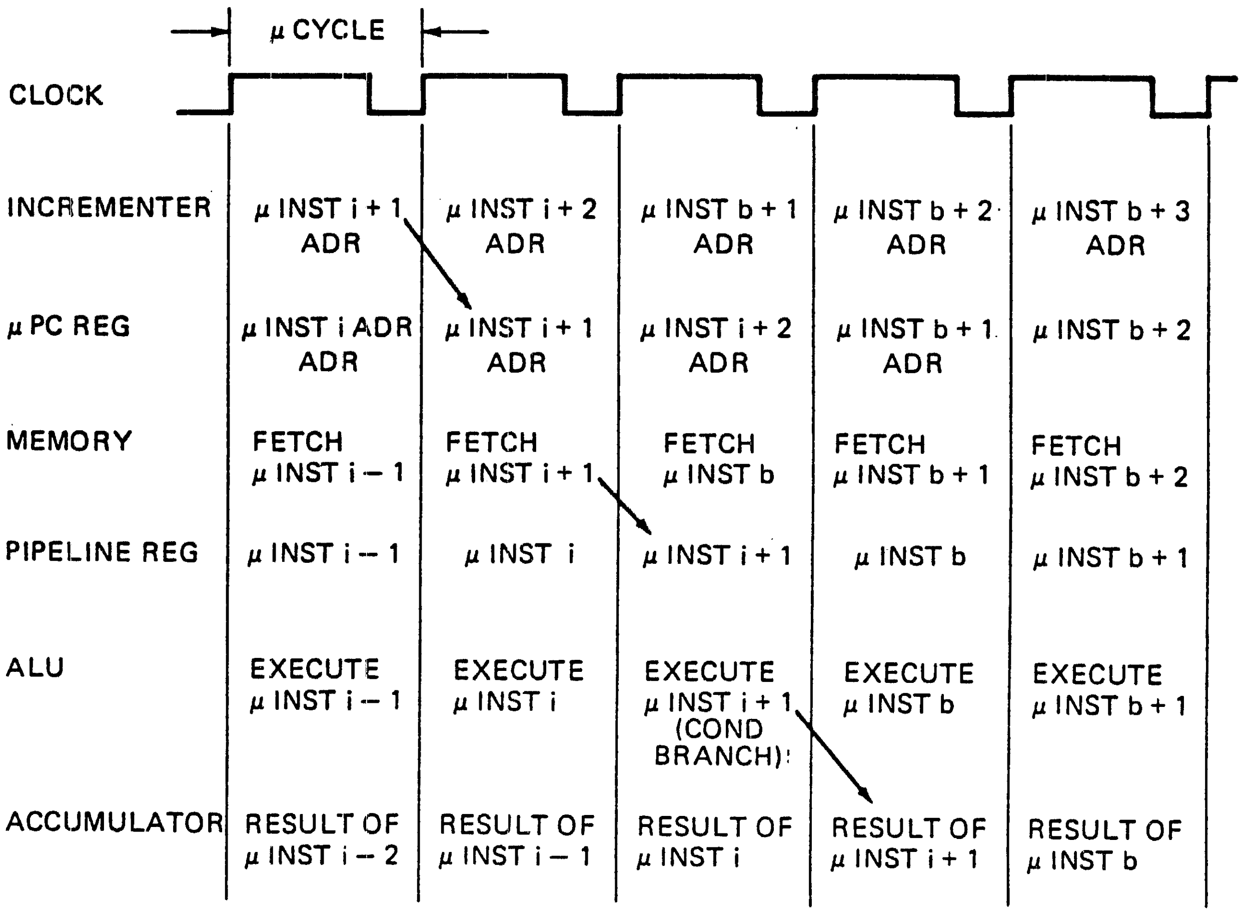


No Branch



Final Version Architecture

Branch Taken - No Penalty



CLOCK	μ CYCLE				
INCREMENTER	μ INST i + 1 ADR	μ INST i + 2 ADR	μ INST b + 1 ADR	μ INST b + 2 ADR	μ INST b + 3 ADR
μ PC REG	μ INST i ADR	μ INST i + 1 ADR	μ INST i + 2 ADR	μ INST b + 1 ADR	μ INST b + 2
MEMORY	FETCH μ INST i – 1	FETCH μ INST i + 1	FETCH μ INST b	FETCH μ INST b + 1	FETCH μ INST b + 2
PIPELINE REG	μ INST i – 1	μ INST i	μ INST i + 1	μ INST b	μ INST b + 1
ALU	EXECUTE μ INST i – 1	EXECUTE μ INST i	EXECUTE μ INST i + 1 (COND BRANCH)	EXECUTE μ INST b	EXECUTE μ INST b + 1
ACCUMULATOR	RESULT OF μ INST i – 2	RESULT OF μ INST i – 1	RESULT OF μ INST i	RESULT OF μ INST i + 1	RESULT OF μ INST b

FURTHER IMPROVEMENTS IN MICROPROGRAM CONTROL

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.

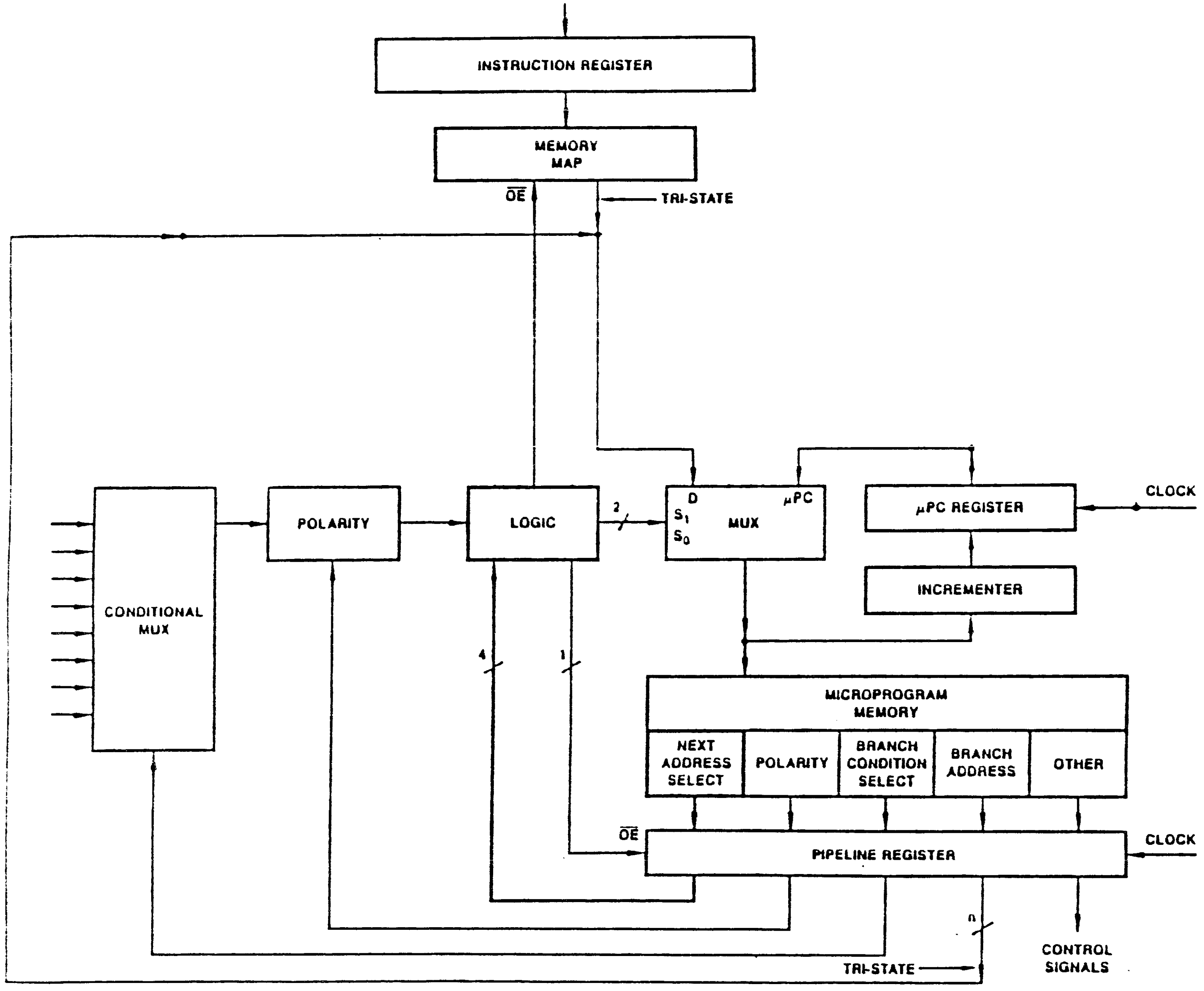
- To facilitate these features, the following capabilities are required to perform a subroutine (a branch and return sequence):

  - 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


INSTRUCTION REGISTER
MEMORY MAP
$\overline{OE}$
TRI-STATE
CONDITIONAL MUX
POLARITY
LOGIC
2
D
S_1
S_0
MUX
μPC
μPC REGISTER
CLOCK
INCREMENTER
4
1
MICROPROGRAM MEMORY
NEXT ADDRESS SELECT
POLARITY
BRANCH CONDITION SELECT
BRANCH ADDRESS
OTHER
$\overline{OE}$
PIPELINE REGISTER
CLOCK
n
TRI-STATE
CONTROL SIGNALS

SUBROUTINES:

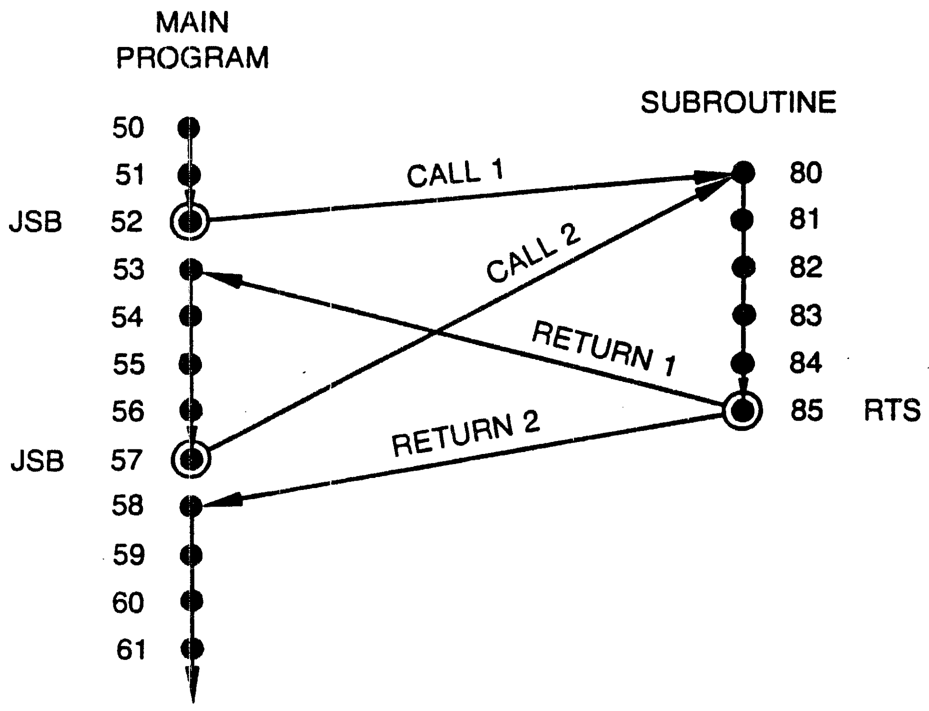
- Subroutines should be callable from anywhere in the microprogram.
- As with jumps/branches, subroutine calls can be conditional or unconditional.
- At the completion of the subroutine, control returns to the main macroprogram statement following the calling statement. This is an unconditional return.
- 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	AFTER 52	AFTER 85	AFTER 57
λ	53	λ	58

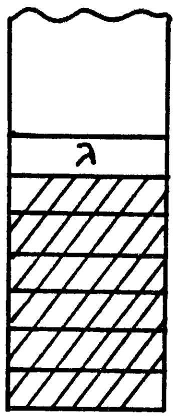
λ means "undefined"

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

"TOP OF STACK" --->

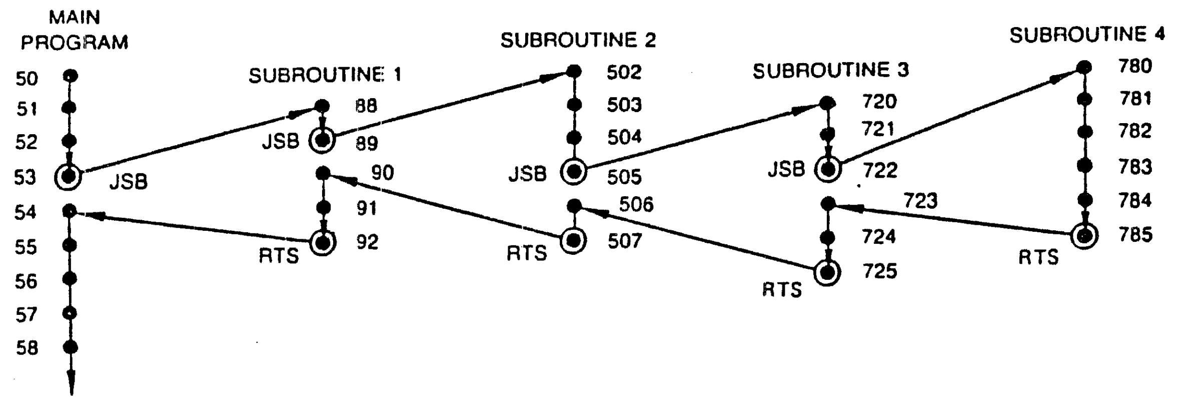
POINTER



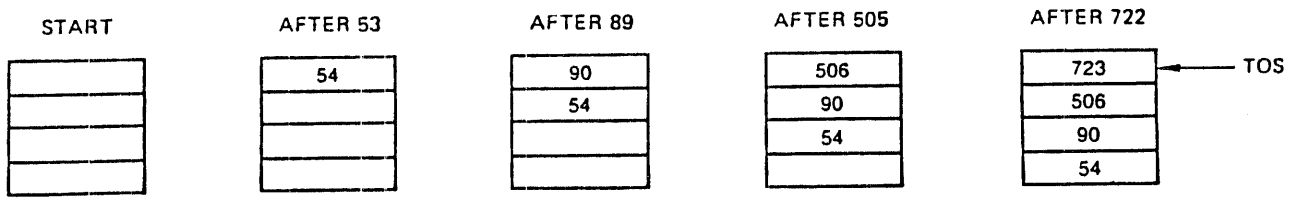
STACK

"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     P  COND   BR
      ADDR     O  MUX    ADDR
      SELECT   L  SEL
------------------------------------------

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"
```

<u>Possible next address controls for our CCU so far:</u>

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** -

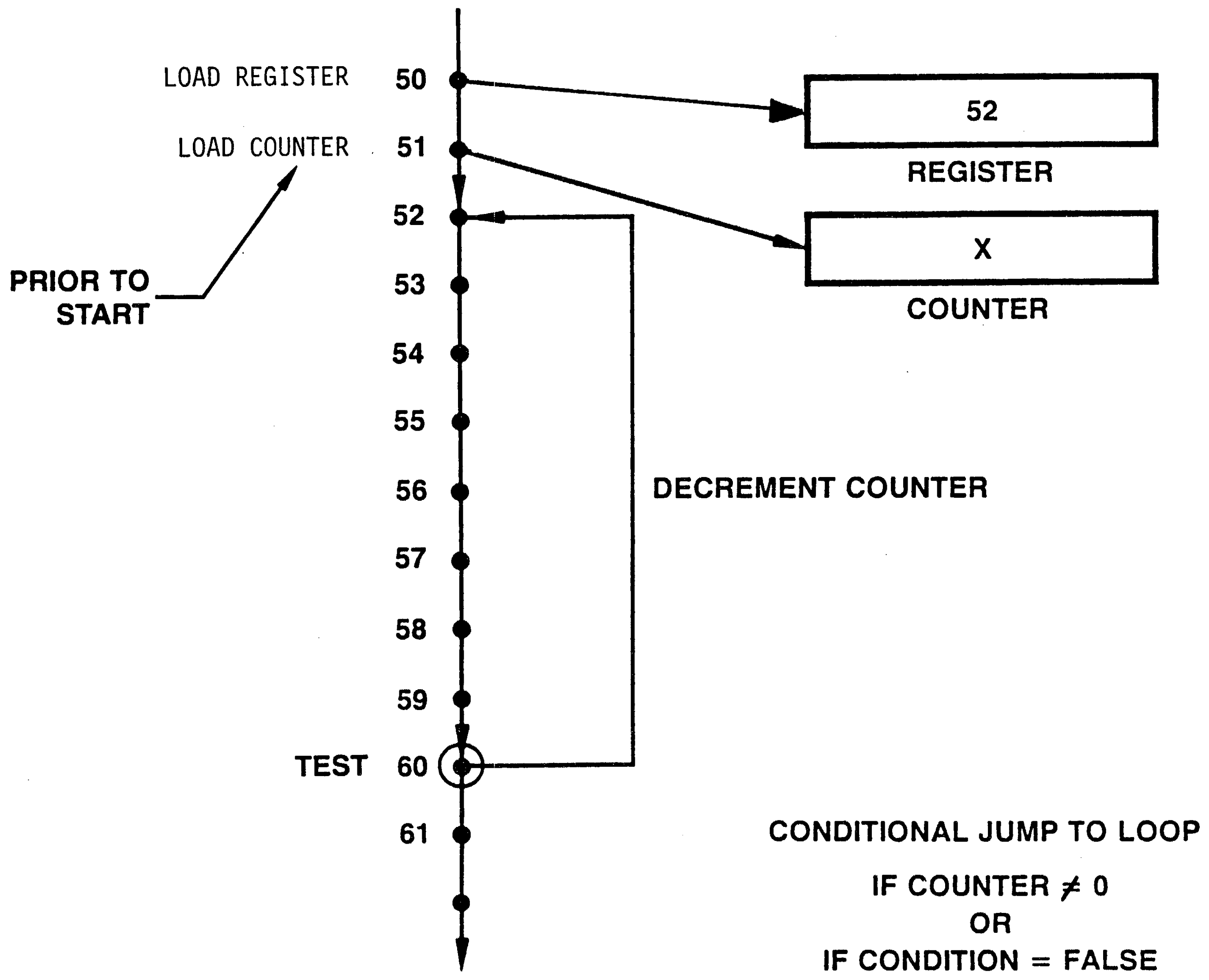
```
    BEGIN LOOP:

        REGISTER  <--- START ADDRESS

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

    END LOOP:

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

        IF COUNTER ≠ Ø 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>
```

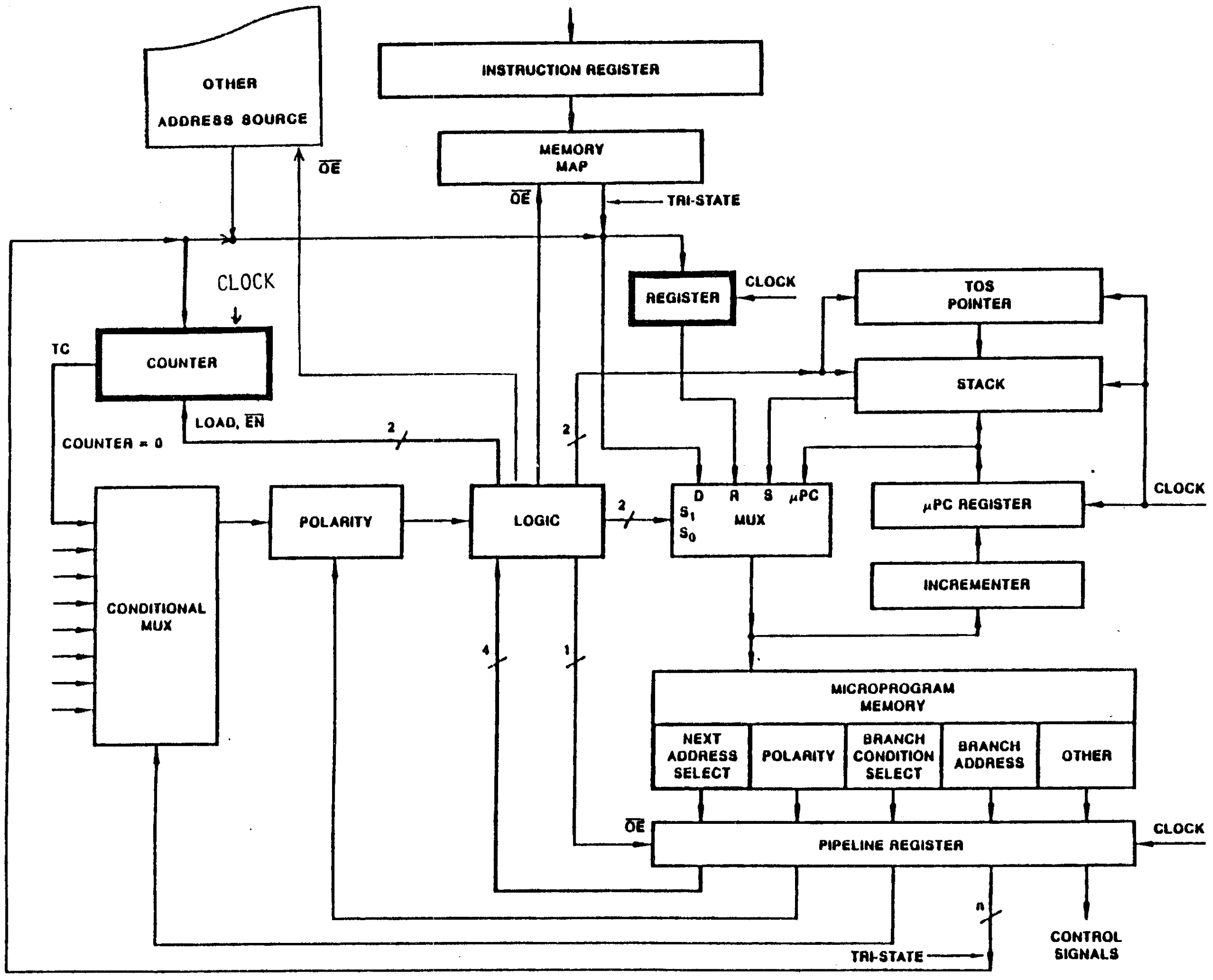
LOOPS


LOAD REGISTER
LOAD COUNTER
PRIOR TO START
TEST
50
51
52
53
54
55
56
57
58
59
60
61
52
REGISTER
X
COUNTER
DECREMENT COUNTER
CONDITIONAL JUMP TO LOOP
IF COUNTER ≠ 0
OR
IF CONDITION = FALSE

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

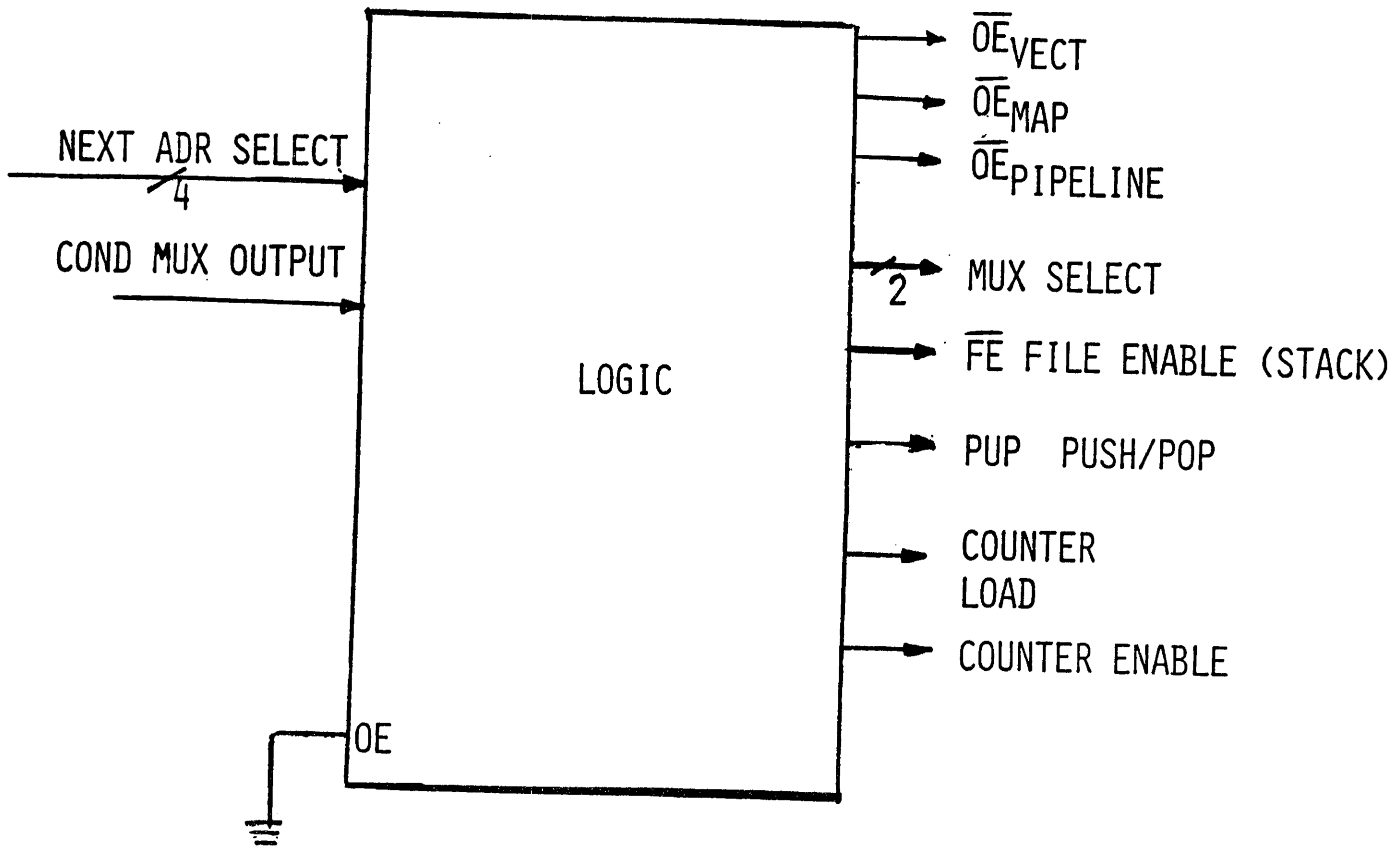
    MAP - for mapping PROM

    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



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

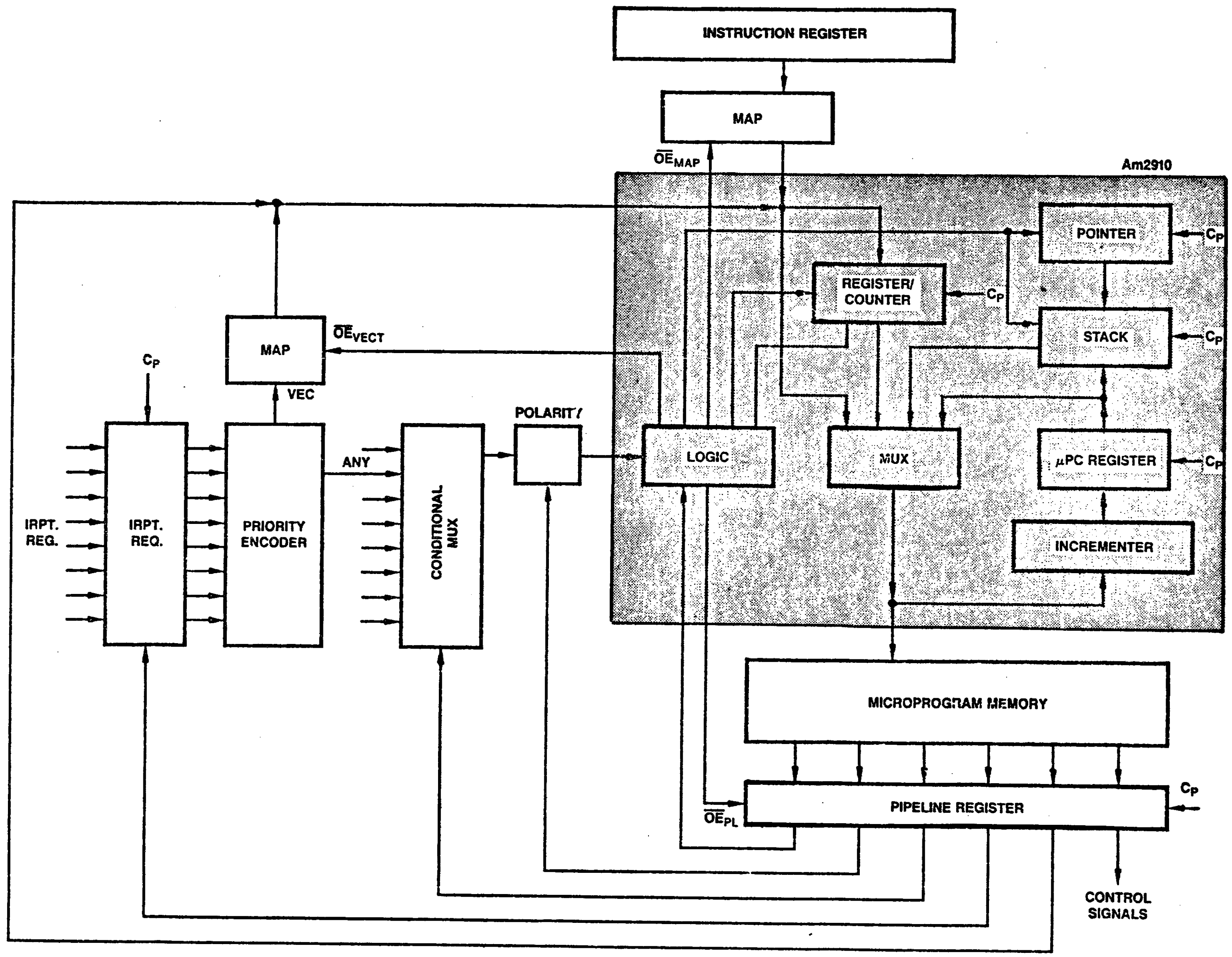
- The Am29112 is similar to the Am2910 in general structure, but is an 8-bit slice expandable to two for addressing 64K of micromemory.
- The Am29112 stack is 33 registers deep.
- The Am29112 also features direct, multiway, relative and program-counter-relative addressing modes, along with vectored interrupts.

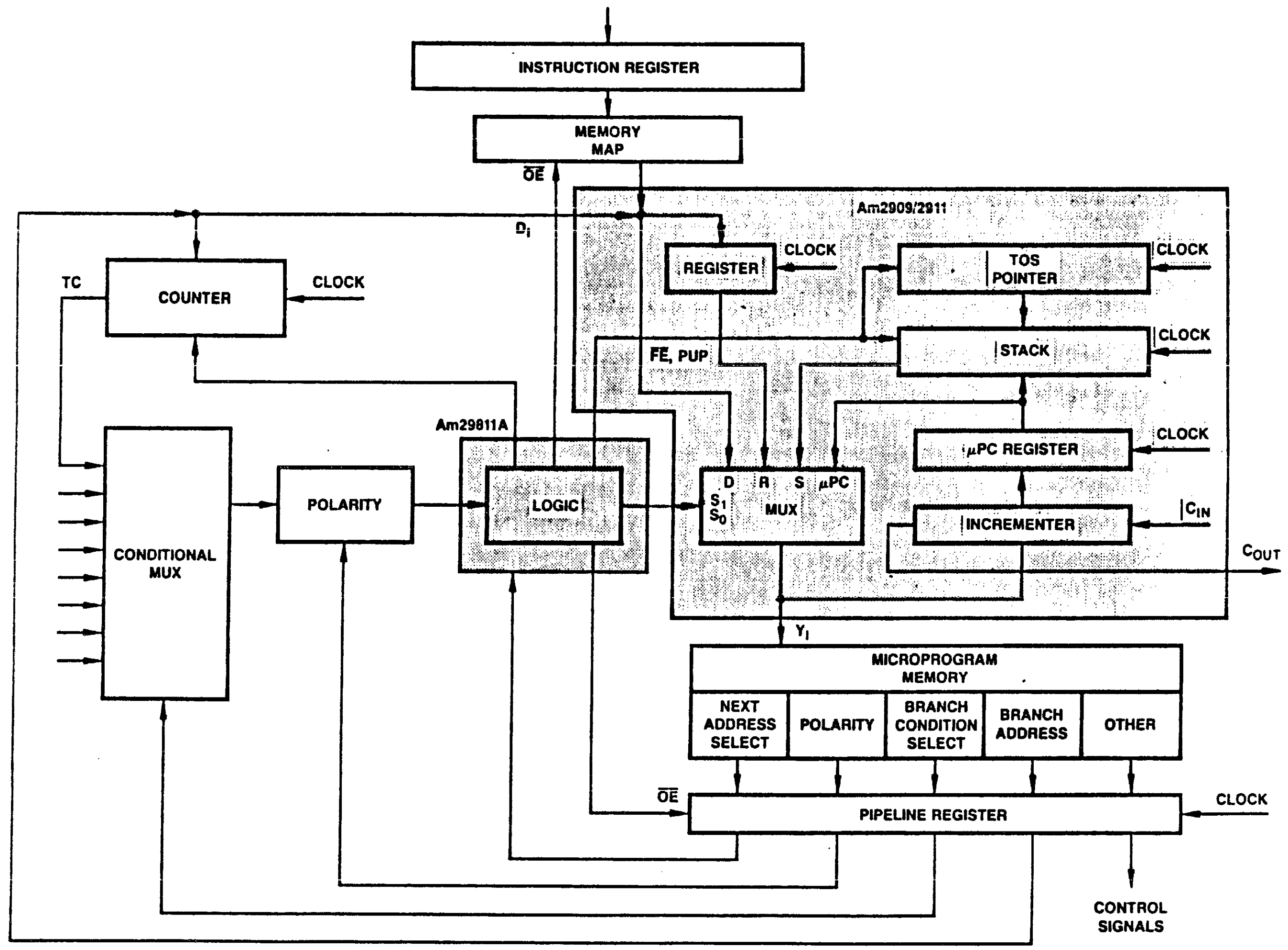
(The Am2910 will be emphasized with possible alternate capabilities discussed with the Am29811 and the Am2909/2911 and the Am29112)

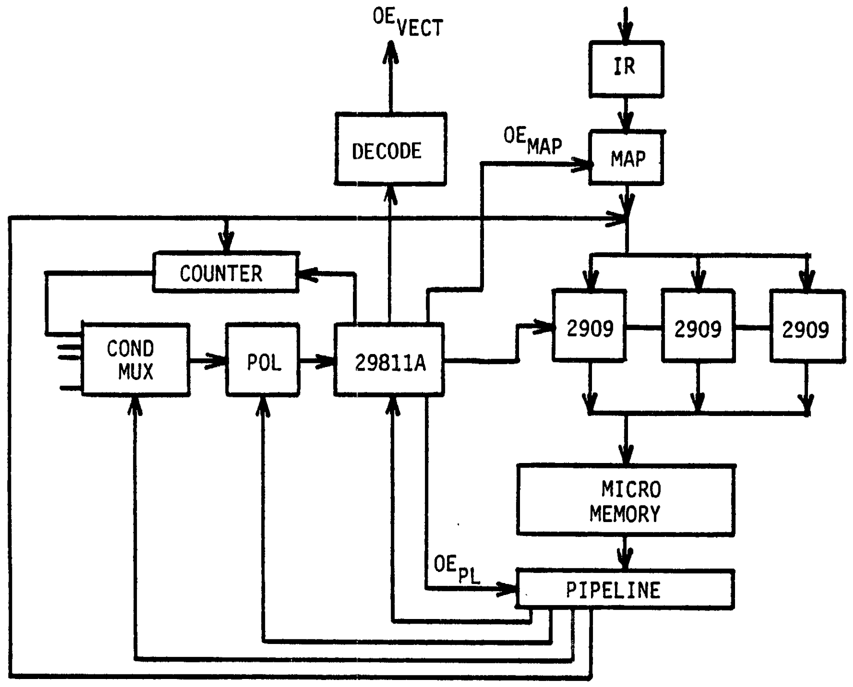
Am2909/2911 SEQUENCERS

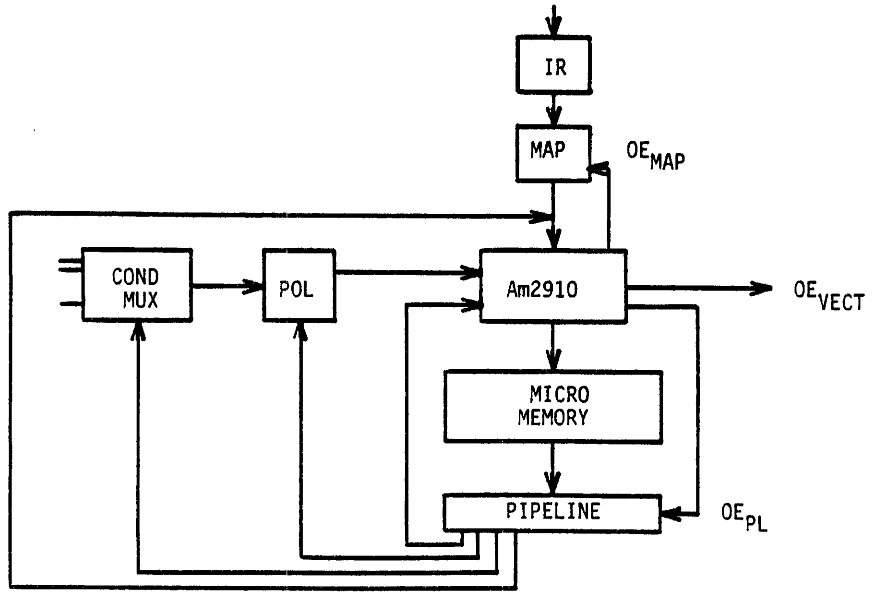
- The Am2909/2911 is a 4-bit sequencer slice, allowing any width of microprogramming addressing and requires next address control logic.
- The Am2909 has four input bits OR'ed with its output for use with the Am29803 for doing 16-way branches (case statement).
- The Am29811 next address control logic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

Am2910

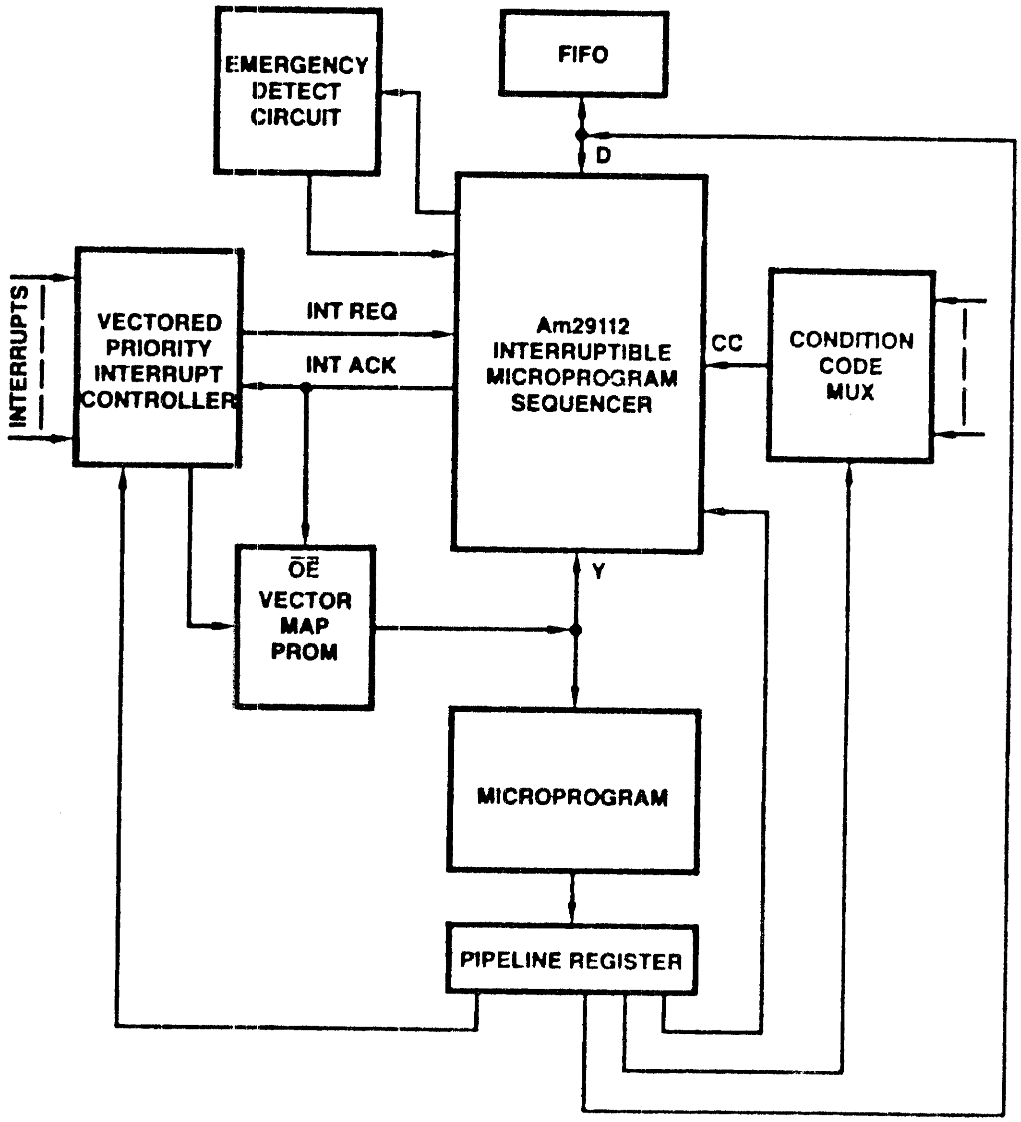

INSTRUCTION REGISTER
MAP
$\overline{OE}_{MAP}$
Am2910
POINTER
C_P
REGISTER/ COUNTER
C_P
STACK
C_P
MAP
$\overline{OE}_{VECT}$
VEC
C_P
POLARITY
ANY
LOGIC
MUX
μPC REGISTER
C_P
IRPT. REG.
IRPT. REQ.
PRIORITY ENCODER
CONDITIONAL MUX
INCREMENTER
MICROPROGRAM MEMORY
C_P
PIPELINE REGISTER
$\overline{OE}_{PL}$
CONTROL SIGNALS


INSTRUCTION REGISTER
MEMORY MAP
OE
Am2909/2911
Dᵢ
REGISTER
CLOCK
TOS POINTER
CLOCK
STACK
CLOCK
FE, PUP
TC
COUNTER
CLOCK
Am29811A
LOGIC
μPC REGISTER
CLOCK
D R S μPC
S₁ S₀
MUX
INCREMENTER
C_IN
C_OUT
CONDITIONAL MUX
POLARITY
Yᵢ
MICROPROGRAM MEMORY
NEXT ADDRESS SELECT
POLARITY
BRANCH CONDITION SELECT
BRANCH ADDRESS
OTHER
OE
PIPELINE REGISTER
CLOCK
CONTROL SIGNALS


OE$_{VECT}$
IR
DECODE
OE$_{MAP}$
MAP
COUNTER
COND MUX
POL
29811A
2909
2909
2909
MICRO MEMORY
OE$_{PL}$
PIPELINE


IR
MAP
OE$_{MAP}$
COND MUX
POL
Am2910
OE$_{VECT}$
MICRO MEMORY
PIPELINE
OE$_{PL}$

Am29112 in a Single Pipelined System



SUPERSEQUENCER

Am2910

Am2910 DISTINCTIVE CHARACTERISTICS

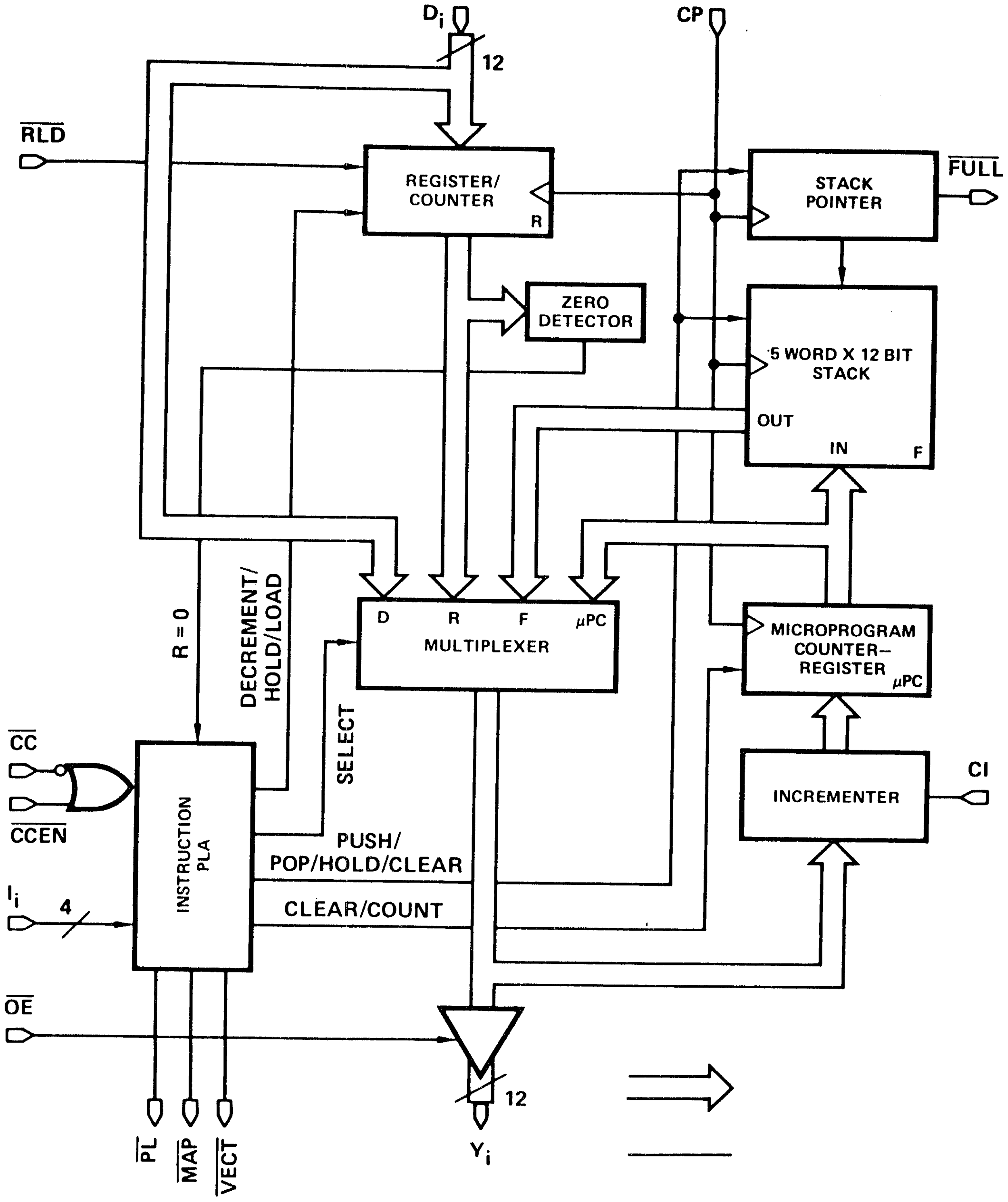
- Twelve bit address output
- Four address sources - D, R, File (Stack output), uPC
- Internal loop counter
- Five deep subroutine stack - Am2910, nine deep - Am2910A
- Conditional test input
- Sixteen powerful microinstructions
- $\overline{\text{OE}}$ for three next address jump sources
- Fast microprogram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

  RLD - register latch

  CCEN - for forced pass

  CI - for inhibiting incrementer

Am2910



D_i
12
CP
RLD
REGISTER/ COUNTER
R
STACK POINTER
FULL
ZERO DETECTOR
5 WORD X 12 BIT STACK
OUT
IN
F
R = 0
DECREMENT/ HOLD/LOAD
D
R
F
μPC
MULTIPLEXER
MICROPROGRAM COUNTER– REGISTER
μPC
SELECT
CC
CCEN
INSTRUCTION PLA
INCREMENTER
CI
PUSH/ POP/HOLD/CLEAR
I_i
4
CLEAR/COUNT
OE
12
PL
MAP
VECT
Y_i

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

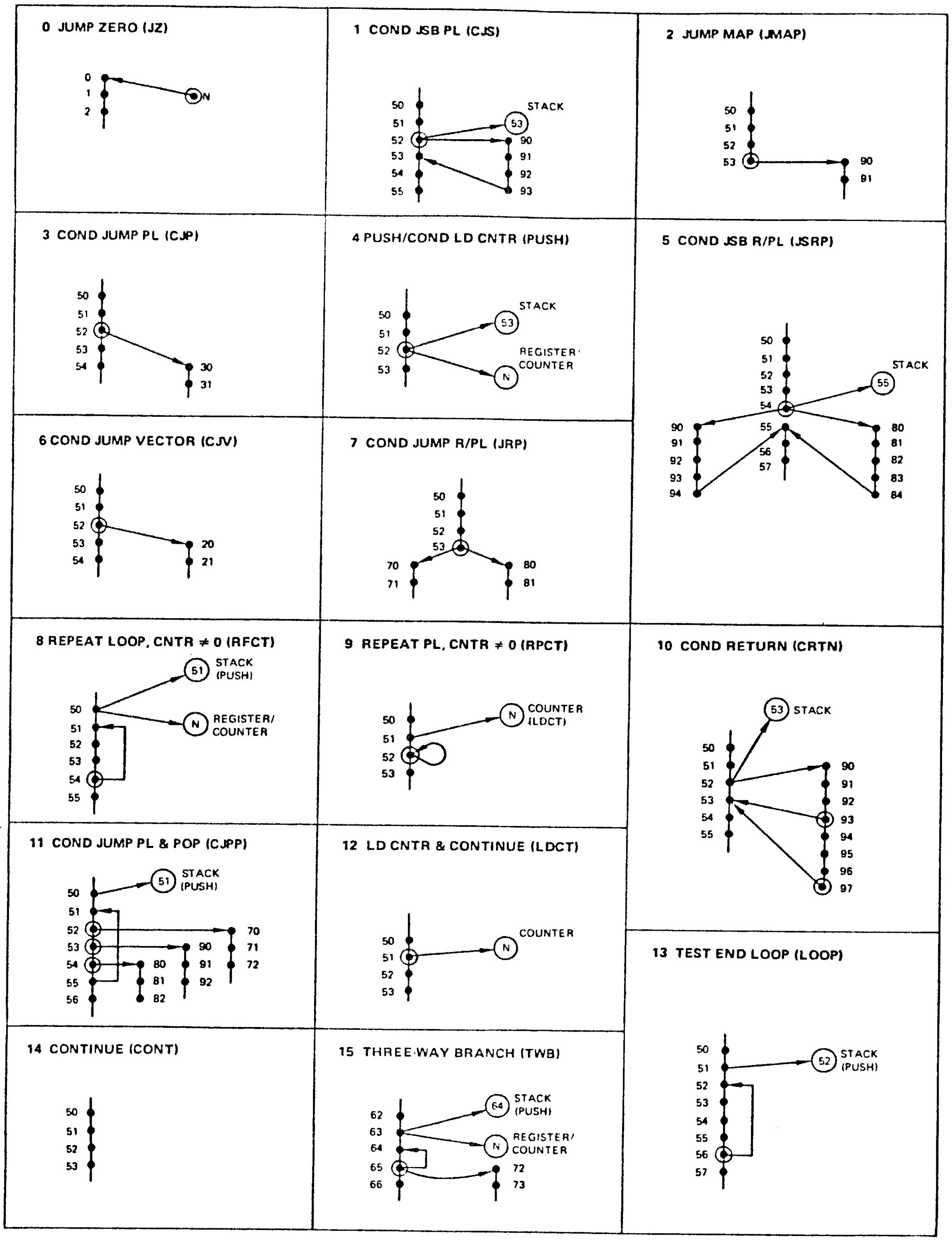
LOOPING:

LDCT	Load Counter and Continue
PUSH	Push Micro-PC on Stack, Conditional Load Counter and Continue
RPCT	Repeat Loop if Counter = Ø, Start Address on Stack
LOOP	Repeat Loop until TEST = TRUE, Start Address on Stack
TWB	Repeat Loop if TEST = FALSE <u>and</u> Counter = Ø

ELSE IF TEST = FALSE and COUNTER = Ø, Go to Pipeline

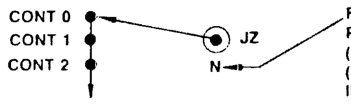
ELSE IF TEST = TRUE Continue

Am2910


0 JUMP ZERO (JZ)
0
1
2
N
1 COND JSB PL (CJS)
50
51
52
53
54
55
STACK
53
90
91
92
93
2 JUMP MAP (JMAP)
50
51
52
53
90
91
3 COND JUMP PL (CJP)
50
51
52
53
54
30
31
4 PUSH/COND LD CNTR (PUSH)
50
51
52
53
STACK
53
REGISTER/
COUNTER
N
5 COND JSB R/PL (JSRP)
50
51
52
53
54
55
56
57
STACK
55
90
91
92
93
94
80
81
82
83
84
6 COND JUMP VECTOR (CJV)
50
51
52
53
54
20
21
7 COND JUMP R/PL (JRP)
50
51
52
53
70
71
80
81
8 REPEAT LOOP, CNTR ≠ 0 (RFCT)
51
STACK
(PUSH)
N
REGISTER/
COUNTER
50
51
52
53
54
55
9 REPEAT PL, CNTR ≠ 0 (RPCT)
N
COUNTER
(LDCT)
50
51
52
53
10 COND RETURN (CRTN)
53
STACK
50
51
52
53
54
55
90
91
92
93
94
95
96
97
11 COND JUMP PL & POP (CJPP)
51
STACK
(PUSH)
50
51
52
53
54
55
56
70
71
72
90
91
92
80
81
82
12 LD CNTR & CONTINUE (LDCT)
COUNTER
N
50
51
52
53
13 TEST END LOOP (LOOP)
50
51
52
53
54
55
56
57
52
STACK
(PUSH)
14 CONTINUE (CONT)
50
51
52
53
15 THREE-WAY BRANCH (TWB)
62
63
64
65
66
64
STACK
(PUSH)
N
REGISTER/
COUNTER
72
73

JZ Jump to Address Zero

```
ADDRESS   LABEL    2910   COND   BRANCH
 (HEX)             INSTR  MUX    ADDRESS
_______________________________________________

   0      START:   CONT   #      #        <--------- Start Address
   1               CONT   #      #
   2               CONT   #      #
   .
   .
   .
   n               JZ     #      #     <------- Hardwired Start Address
```

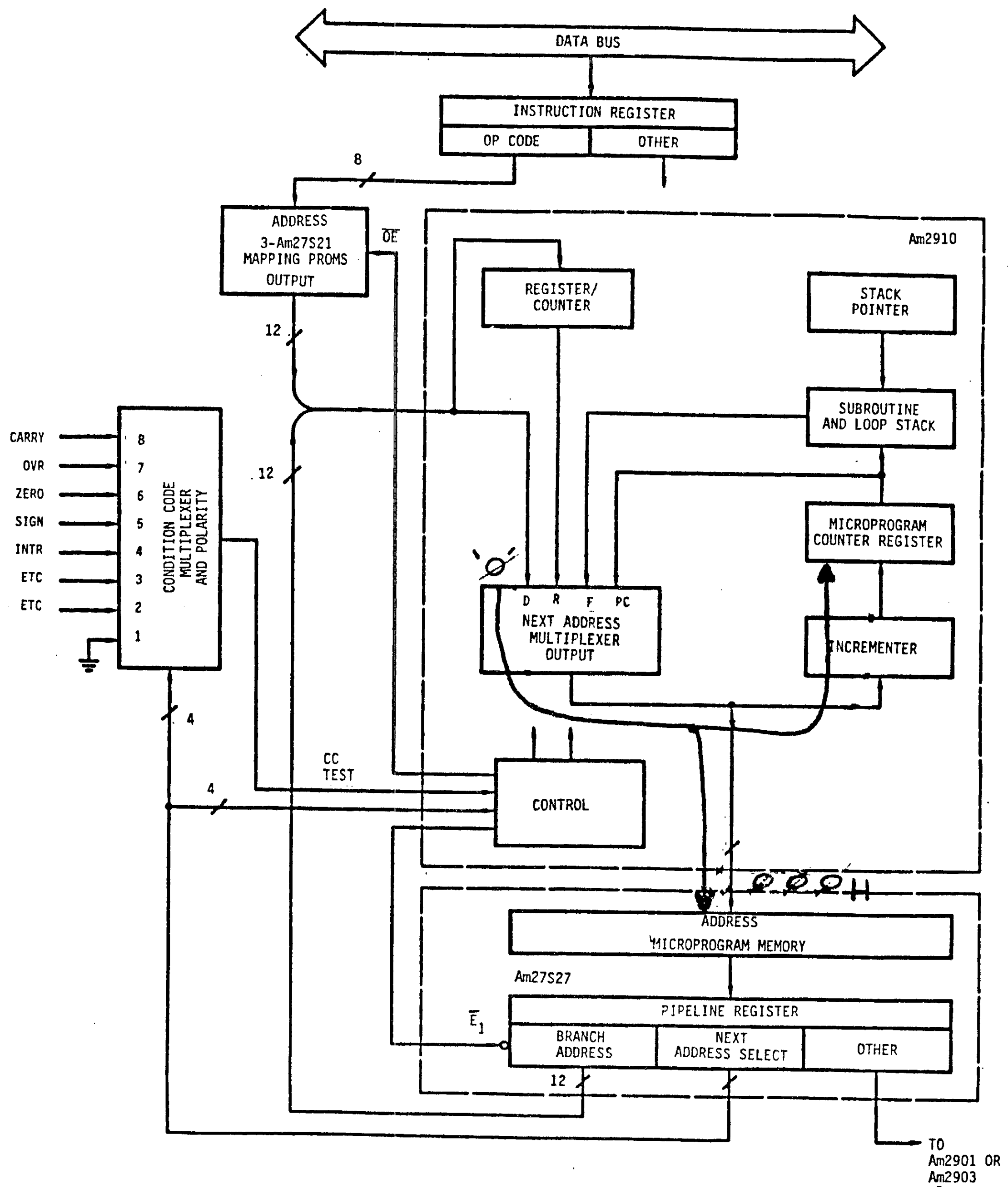


FROM SPECIAL ADDRESS OR RESET OF PIPELINE REGISTER. EITHER SEND 000 (JZ) TO Am2910 OR THE INITIALIZATION (START, RESET) COULD SEND ADDRESS FFF INTO MICROMEMORY. JZ SHOULD BE PLACED THERE. JZ RESETS THE STACK AND SHOULD BE EXECUTED FIRST.

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

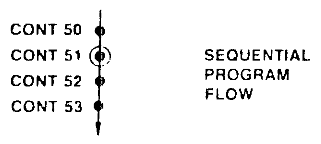
Figure 4-9. Jump zero (JZ, 0).

JZ


DATA BUS
INSTRUCTION REGISTER
OP CODE
OTHER
8
ADDRESS
3-Am27S21
MAPPING PROMS
OUTPUT
OE
12
12
Am2910
REGISTER/
COUNTER
STACK
POINTER
SUBROUTINE
AND LOOP STACK
MICROPROGRAM
COUNTER REGISTER
INCREMENTER
D R F PC
NEXT ADDRESS
MULTIPLEXER
OUTPUT
CARRY
OVR
ZERO
SIGN
INTR
ETC
ETC
8 7 6 5 4 3 2 1
CONDITION CODE
MULTIPLEXER
AND POLARITY
4
4
CC
TEST
CONTROL
000H
ADDRESS
MICROPROGRAM MEMORY
Am27S27
PIPELINE REGISTER
E1
BRANCH
ADDRESS
NEXT
ADDRESS SELECT
OTHER
12
TO
Am2901 OR
Am2903

CONT Continue to Next Instruction in Sequence

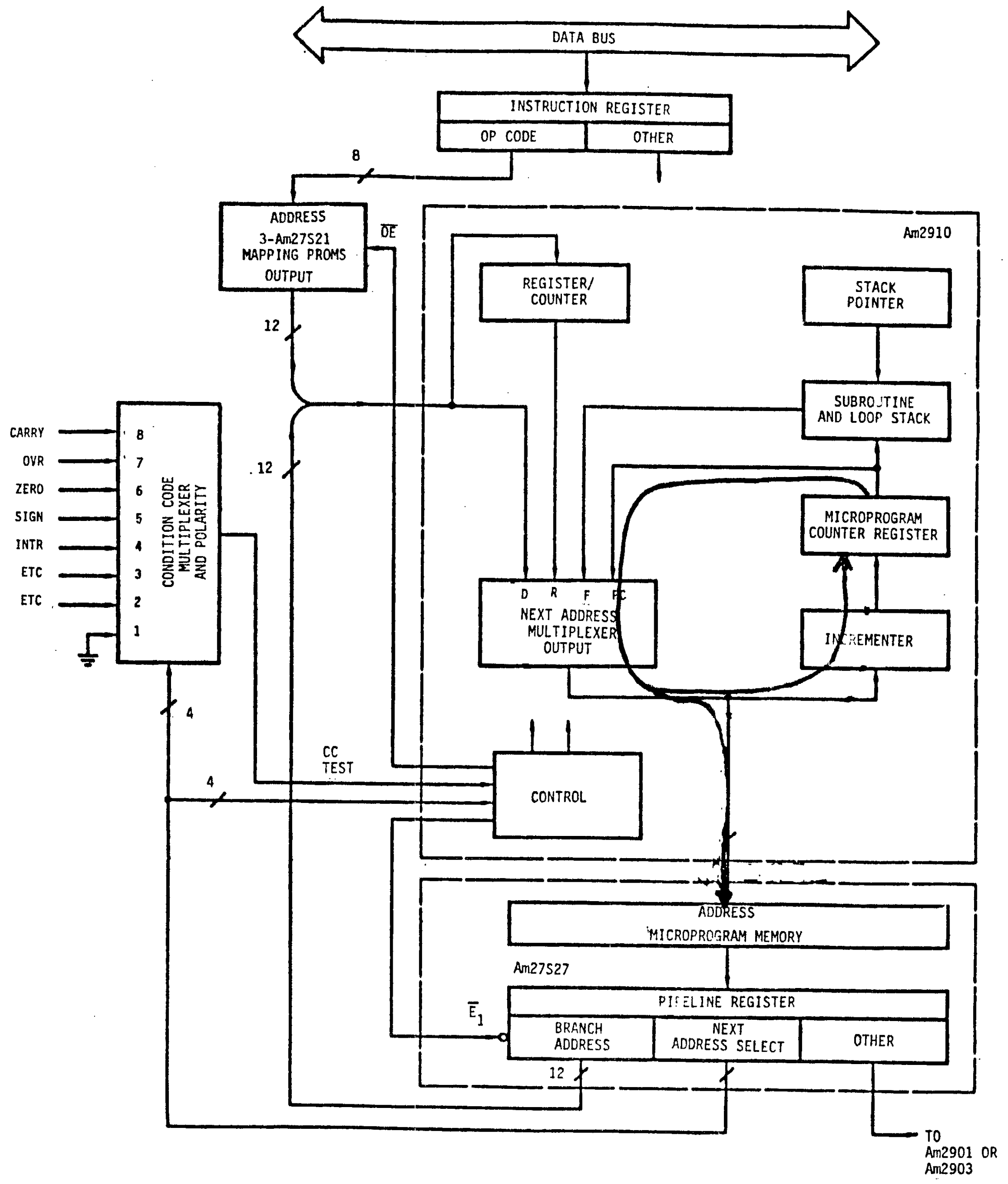
```
ADDRESS  LABEL   2910   COND   BRANCH
 (HEX)           INSTR  MUX    ADDRESS
-----------------------------------------------

  50             CONT   #      #
  51             CONT   #      #
  52             CONT   #      #           Sequential Program Flow
  53             CONT   #      #
```



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

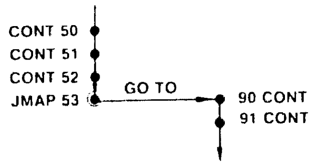
Figure 4–10. Continue (CONT, E).

CONT


DATA BUS
INSTRUCTION REGISTER
OP CODE
OTHER
8
ADDRESS
3-Am27S21
MAPPING PROMS
OUTPUT
$\overline{OE}$
Am2910
REGISTER/
COUNTER
STACK
POINTER
12
SUBROUTINE
AND LOOP STACK
12
CARRY
OVR
ZERO
SIGN
INTR
ETC
ETC
8
7
6
5
4
3
2
1
CONDITION CODE
MULTIPLEXER
AND POLARITY
MICROPROGRAM
COUNTER REGISTER
D R F PC
NEXT ADDRESS
MULTIPLEXER
OUTPUT
INCREMENTER
4
CC
TEST
4
CONTROL
ADDRESS
MICROPROGRAM MEMORY
Am27S27
$\overline{E}_1$
PIPELINE REGISTER
BRANCH
ADDRESS
NEXT
ADDRESS SELECT
OTHER
12
TO
Am2901 OR
Am2903

JMAP **Jump to Start Address (Enable Mapping PROM)**

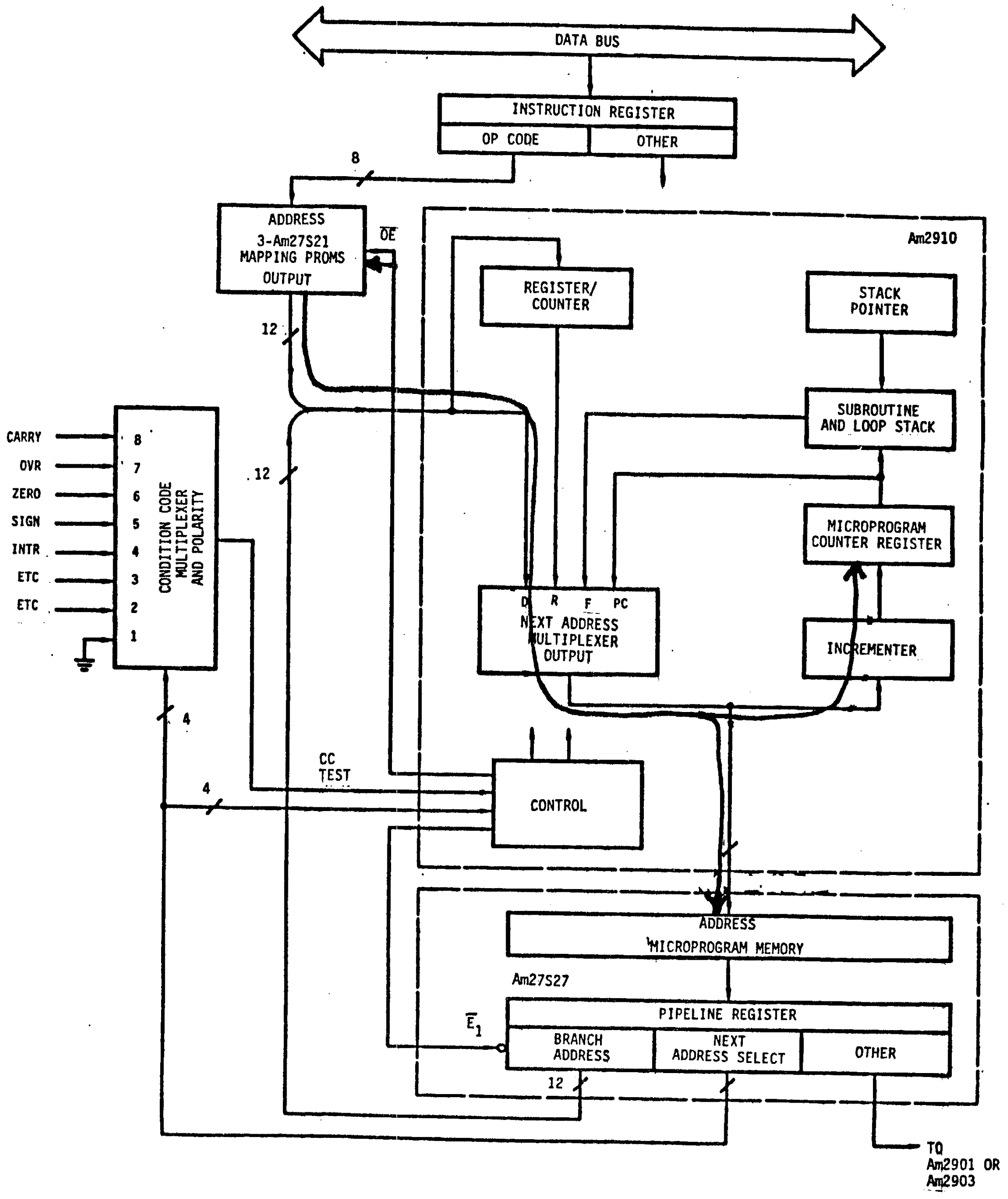
```
ADDRESS  LABEL   2910   COND   BRANCH
 (HEX)           INSTR  MUX    ADDRESS
_______________________________________________

  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

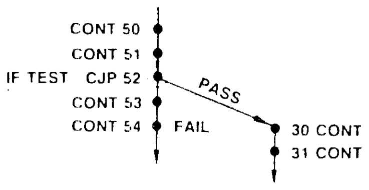
Figure 4–11. Jump map (JMAP. 2).

JMAP


DATA BUS
INSTRUCTION REGISTER
OP CODE
OTHER
8
ADDRESS
3-Am27S21
MAPPING PROMS
OUTPUT
OE
12
Am2910
REGISTER/
COUNTER
STACK
POINTER
SUBROUTINE
AND LOOP STACK
MICROPROGRAM
COUNTER REGISTER
INCREMENTER
D R F PC
NEXT ADDRESS
MULTIPLEXER
OUTPUT
CARRY
OVR
ZERO
SIGN
INTR
ETC
ETC
8 7 6 5 4 3 2 1
CONDITION CODE
MULTIPLEXER
AND POLARITY
12
4
CC
TEST
4
CONTROL
ADDRESS
MICROPROGRAM MEMORY
Am27S27
$\overline{E}_1$
PIPELINE REGISTER
BRANCH
ADDRESS
NEXT
ADDRESS SELECT
OTHER
12
TO
Am2901 OR
Am2903

CJP Conditional Jump to Branch Address (Pipeline)

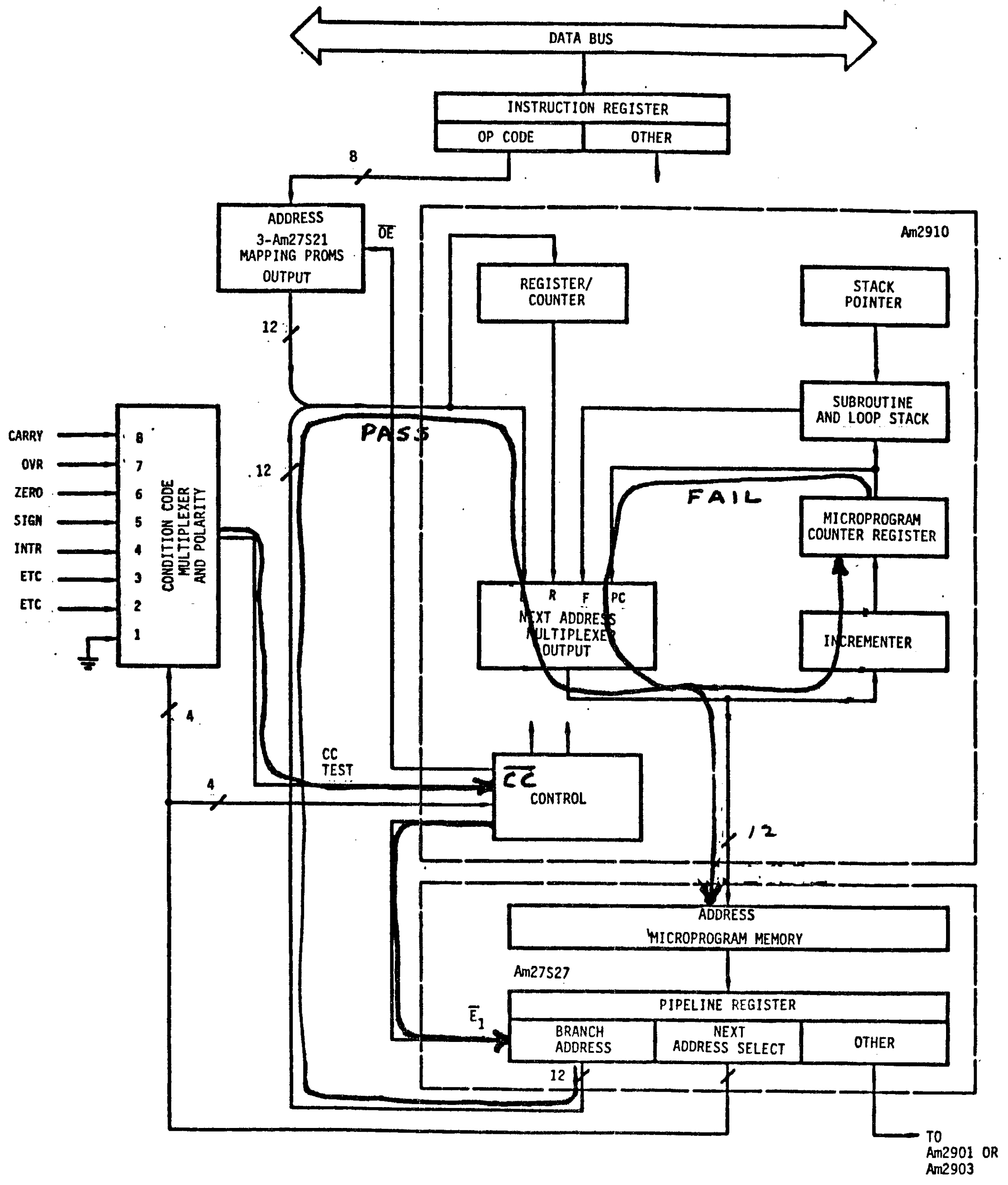
ADDRESS (HEX)	LABEL	2910 INSTR	COND MUX	BRANCH ADDRESS
30	LABELA:	CONT	#	#
31		CONT	#	#
.				
.				
.				
50		CONT	#	#
51		CONT	#	#
52		CJP	TESTA	LABELA
53		CONT	#	#
54		CONT	#	#



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

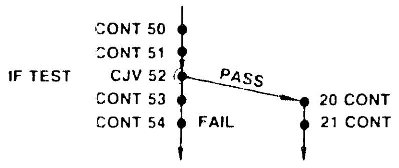
Figure 4–12. Conditional jump pipeline (CJP, 3).

CJP


DATA BUS
INSTRUCTION REGISTER
OP CODE
OTHER
8
ADDRESS
3-Am27S21
MAPPING PROMS
OUTPUT
OE
12
Am2910
REGISTER/ COUNTER
STACK POINTER
SUBROUTINE AND LOOP STACK
MICROPROGRAM COUNTER REGISTER
INCREMENTER
PASS
FAIL
R F PC
NEXT ADDRESS MULTIPLEXER OUTPUT
CARRY
OVR
ZERO
SIGN
INTR
ETC
ETC
8 7 6 5 4 3 2 1
CONDITION CODE MULTIPLEXER AND POLARITY
12
4
CC TEST
4
CC
CONTROL
12
ADDRESS
MICROPROGRAM MEMORY
Am27S27
PIPELINE REGISTER
E1
BRANCH ADDRESS
NEXT ADDRESS SELECT
OTHER
12
TO Am2901 OR Am2903

CJV Conditional Jump to Vector Map Output

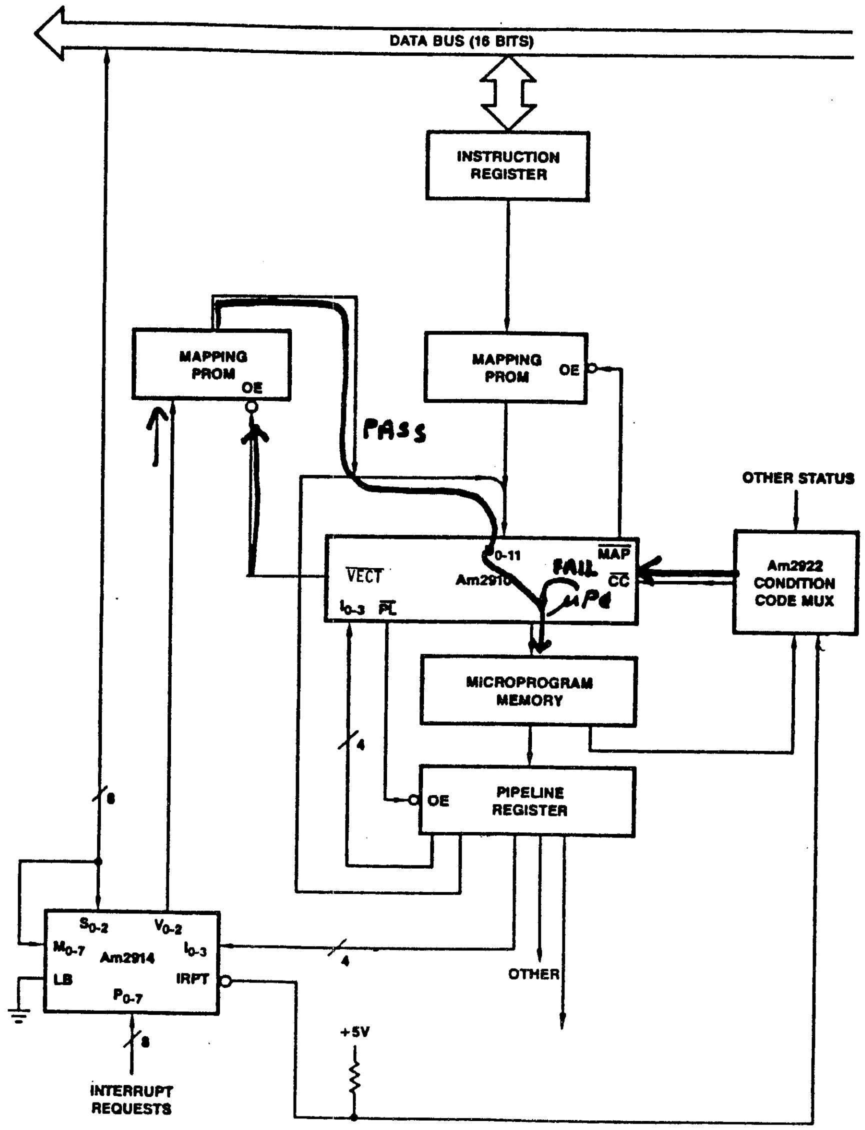
```
ADDRESS   LABEL   2910   COND   BRANCH
 (HEX)            INSTR  MUX    ADDRESS
________________________________________________

  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

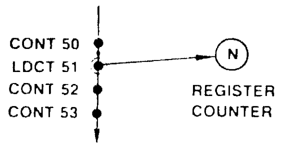
Figure 4-13. Conditional jump vector (CJV, 6).

CJV


DATA BUS (16 BITS)
INSTRUCTION REGISTER
MAPPING PROM
OE
MAPPING PROM
OE
PASS
OTHER STATUS
VECT
Am2910
0-11
MAP
CC
FAIL
μPC
I_{0-3} PL
Am2922 CONDITION CODE MUX
MICROPROGRAM MEMORY
4
8
OE
PIPELINE REGISTER
S_{0-2}
V_{0-2}
M_{0-7}
Am2914
I_{0-3}
LB
IRPT
P_{0-7}
4
OTHER
+5V
8
INTERRUPT REQUESTS

LDCT Load the Register/Counter and Continue

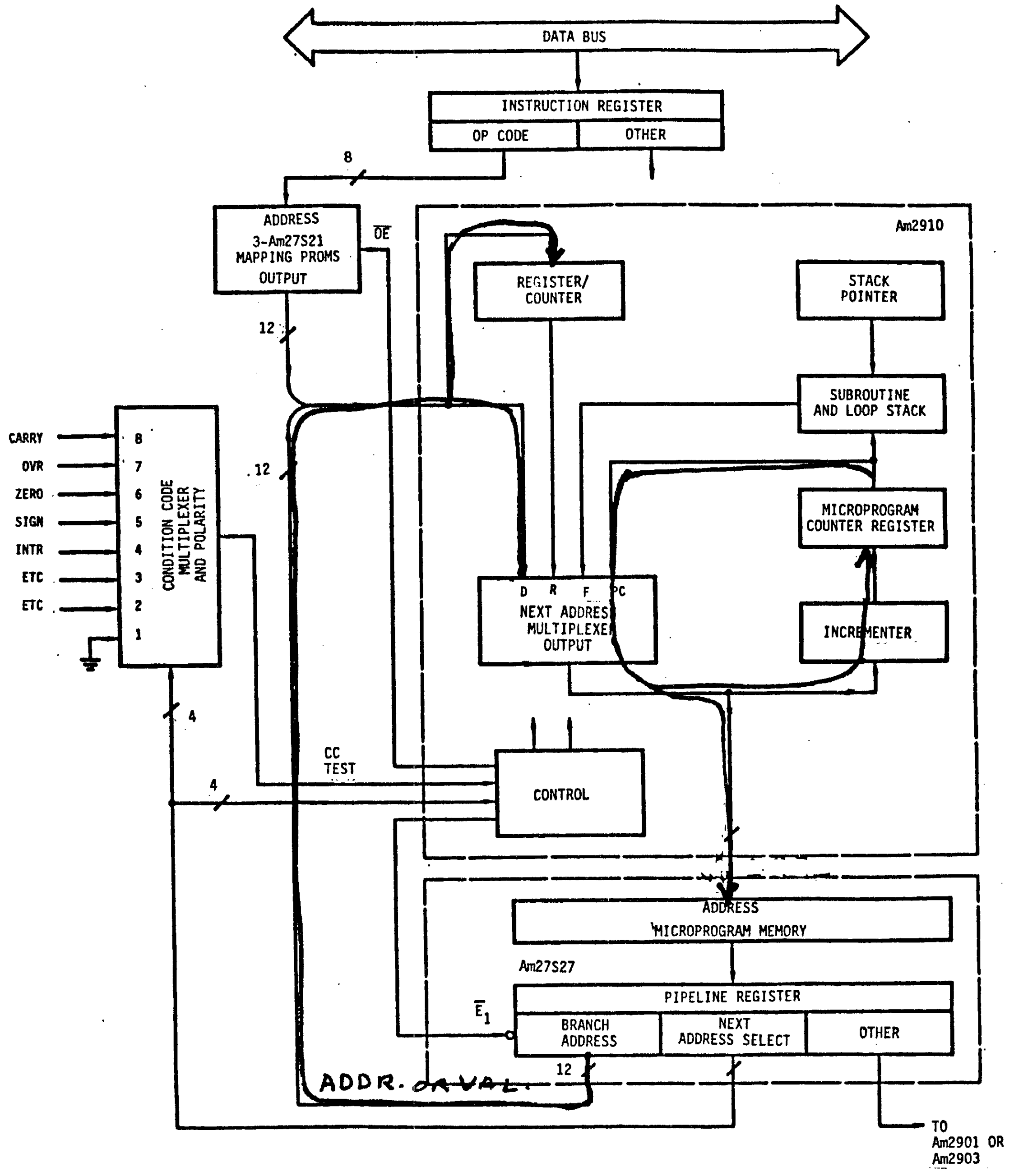
ADDRESS (HEX)	LABEL	2910 INSTR	COND MUX	BRANCH ADDRESS
50		CONT	#	#
51		LDCT	#	VALUE-1
52		CONT	#	#
53		CONT	#	#



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

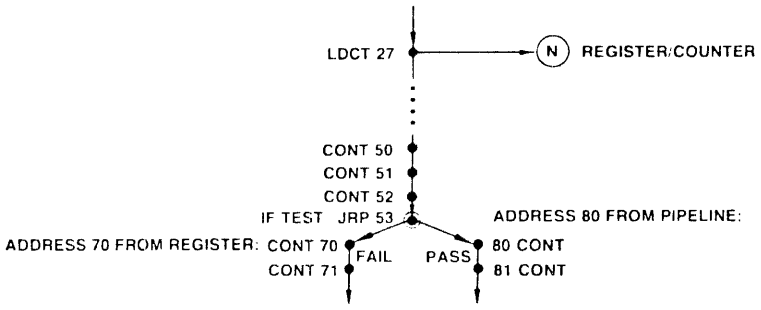
Figure 4-14. Load counter and continue (LDCT. C). This instruction must be executed before a loop instruction or a jump which used the register.

LDCT


DATA BUS
INSTRUCTION REGISTER
OP CODE
OTHER
8
ADDRESS
3-Am27S21
MAPPING PROMS
OUTPUT
$\overline{OE}$
12
Am2910
REGISTER/
COUNTER
STACK
POINTER
SUBROUTINE
AND LOOP STACK
MICROPROGRAM
COUNTER REGISTER
INCREMENTER
D R F PC
NEXT ADDRESS
MULTIPLEXER
OUTPUT
CARRY
OVR
ZERO
SIGN
INTR
ETC
ETC
8
7
6
5
4
3
2
1
CONDITION CODE
MULTIPLEXER
AND POLARITY
12
4
CC
TEST
4
CONTROL
ADDRESS
MICROPROGRAM MEMORY
Am27S27
PIPELINE REGISTER
$\overline{E}_1$
BRANCH
ADDRESS
NEXT
ADDRESS SELECT
OTHER
12
ADDR. ~~or VAL.~~
TO
Am2901 OR
Am2903

JRP Conditional Jump to Register or Branch Address (Pipeline)

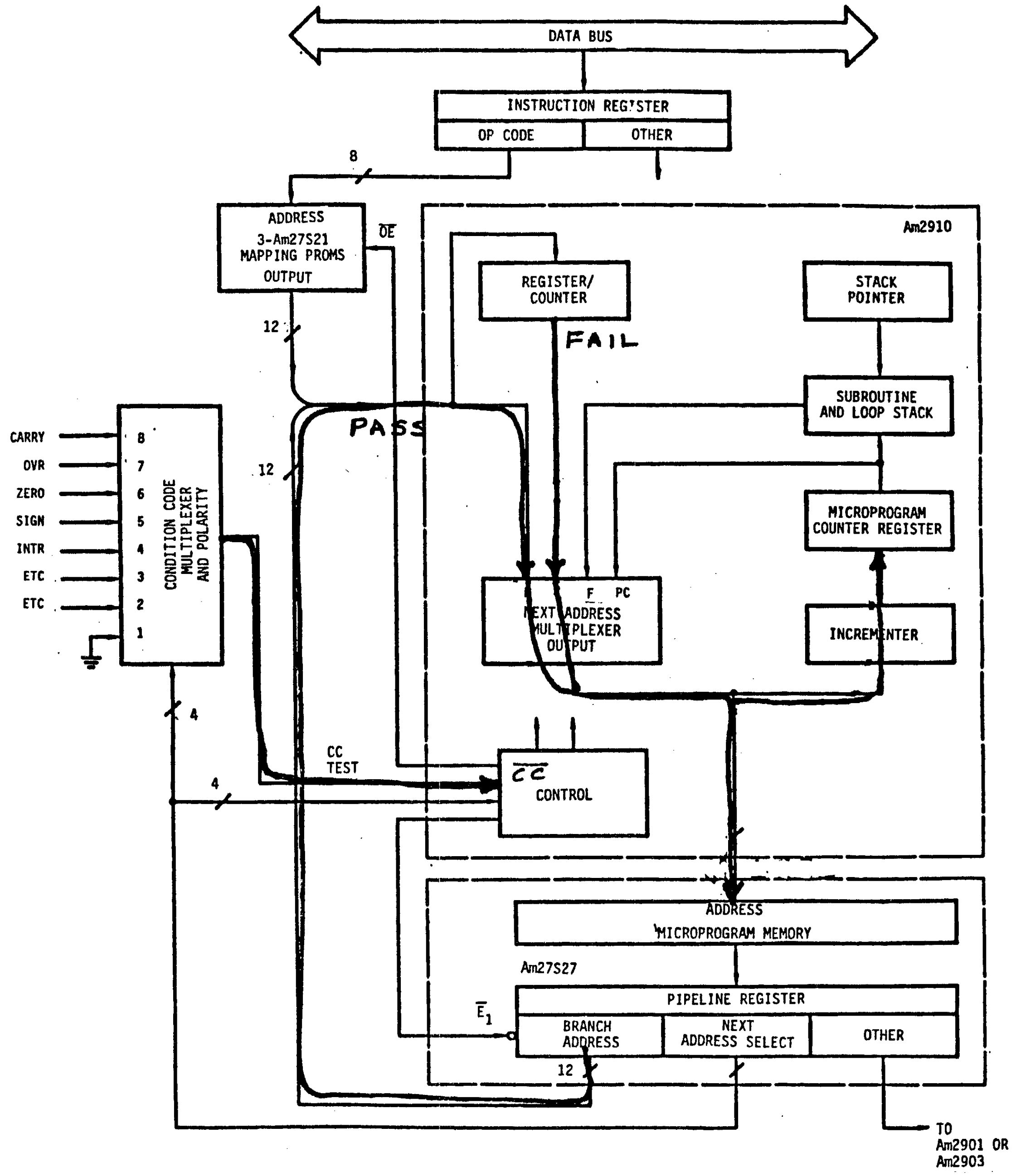
```
ADDRESS   LABEL    2910   COND   BRANCH
 (HEX)             INSTR  MUX    ADDRESS
_____________________________________________

  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   #      #
```



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

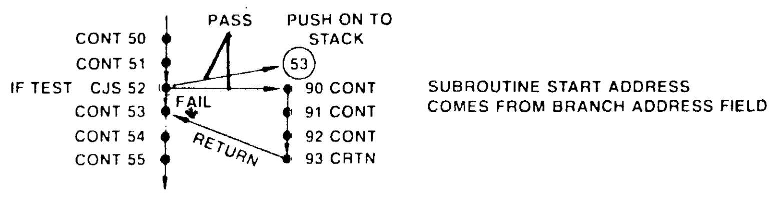
Figure 4-15. Conditional jump register/pipeline (JRP, 7). LDCT must have been executed somewhere ahead of JRP.

JRP


DATA BUS
INSTRUCTION REGISTER
OP CODE
OTHER
8
ADDRESS
3-Am27S21
MAPPING PROMS
OUTPUT
OE
12
Am2910
REGISTER/
COUNTER
FAIL
STACK
POINTER
SUBROUTINE
AND LOOP STACK
PASS
CARRY
OVR
ZERO
SIGN
INTR
ETC
ETC
8
7
6
5
4
3
2
1
CONDITION CODE
MULTIPLEXER
AND POLARITY
12
MICROPROGRAM
COUNTER REGISTER
F
PC
NEXT ADDRESS
MULTIPLEXER
OUTPUT
INCREMENTER
4
CC
TEST
4
CC
CONTROL
ADDRESS
MICROPROGRAM MEMORY
Am27S27
PIPELINE REGISTER
E1
BRANCH
ADDRESS
NEXT
ADDRESS SELECT
OTHER
12
TO
Am2901 OR
Am2903

CJS Conditional Jump to Subroutine Address

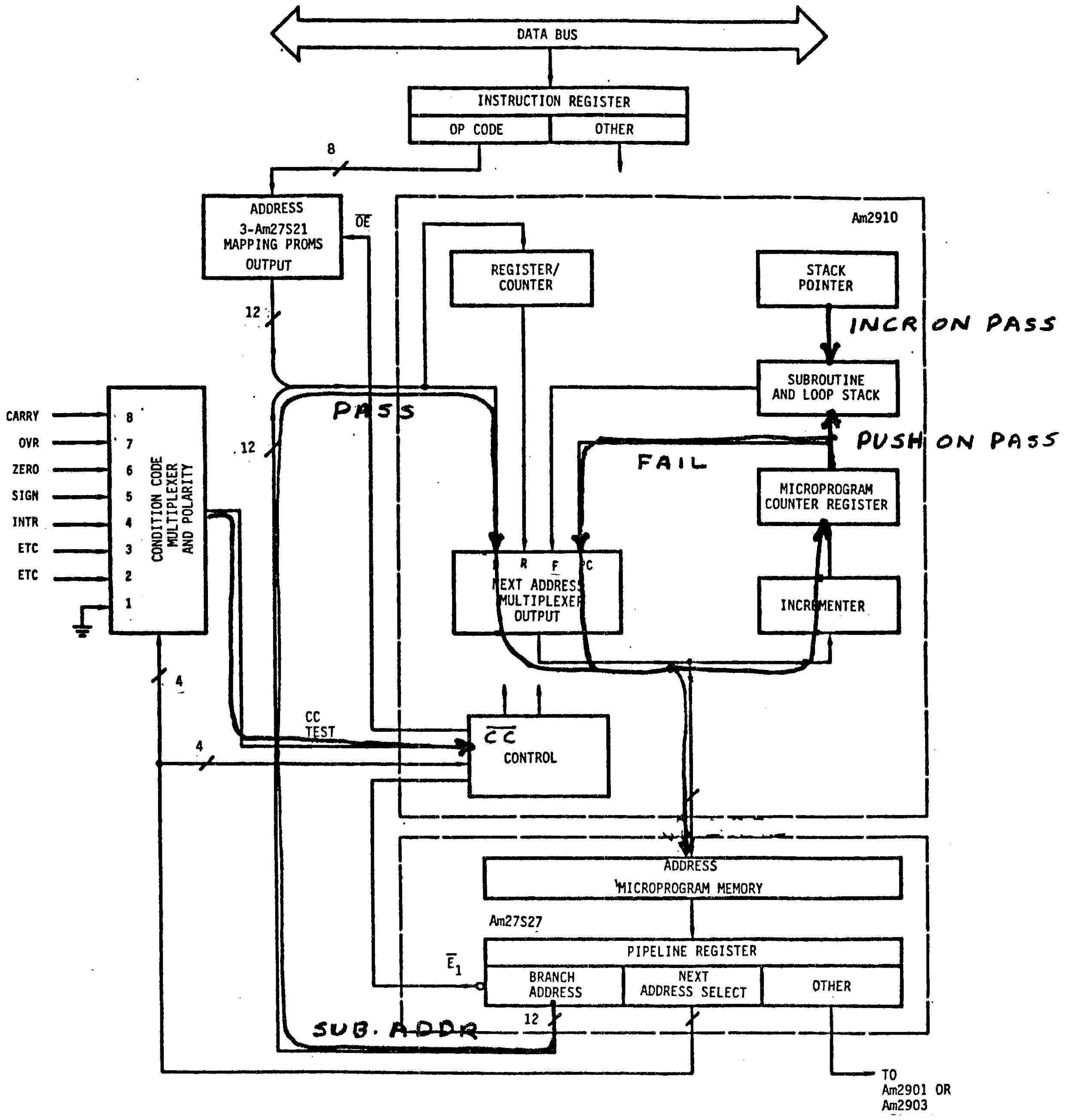
```
ADDRESS   LABEL     2910   COND    BRANCH
 (HEX)              INSTR  MUX     ADDRESS
________________________________________________________

  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
```



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

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

CJS


DATA BUS
INSTRUCTION REGISTER
OP CODE
OTHER
8
ADDRESS
3-Am27S21
MAPPING PROMS
OUTPUT
OE
12
Am2910
REGISTER/
COUNTER
STACK
POINTER
INCR ON PASS
SUBROUTINE
AND LOOP STACK
PUSH ON PASS
PASS
FAIL
MICROPROGRAM
COUNTER REGISTER
CARRY
OVR
ZERO
SIGN
INTR
ETC
ETC
CONDITION CODE
MULTIPLEXER
AND POLARITY
NEXT ADDRESS
MULTIPLEXER
OUTPUT
R
F
PC
INCREMENTER
4
CC
TEST
CC
CONTROL
ADDRESS
MICROPROGRAM MEMORY
Am27S27
PIPELINE REGISTER
E1
BRANCH
ADDRESS
NEXT
ADDRESS SELECT
OTHER
SUB. ADDR
TO
Am2901 OR
Am2903

JSRP Conditional Jump to Subroutine (Register or Pipeline)

```
ADDRESS   LABEL     2910   COND    BRANCH
 (HEX)              INSTR  MUX     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
```

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

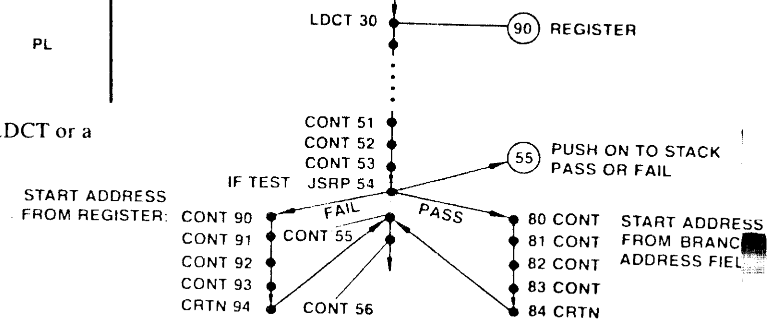
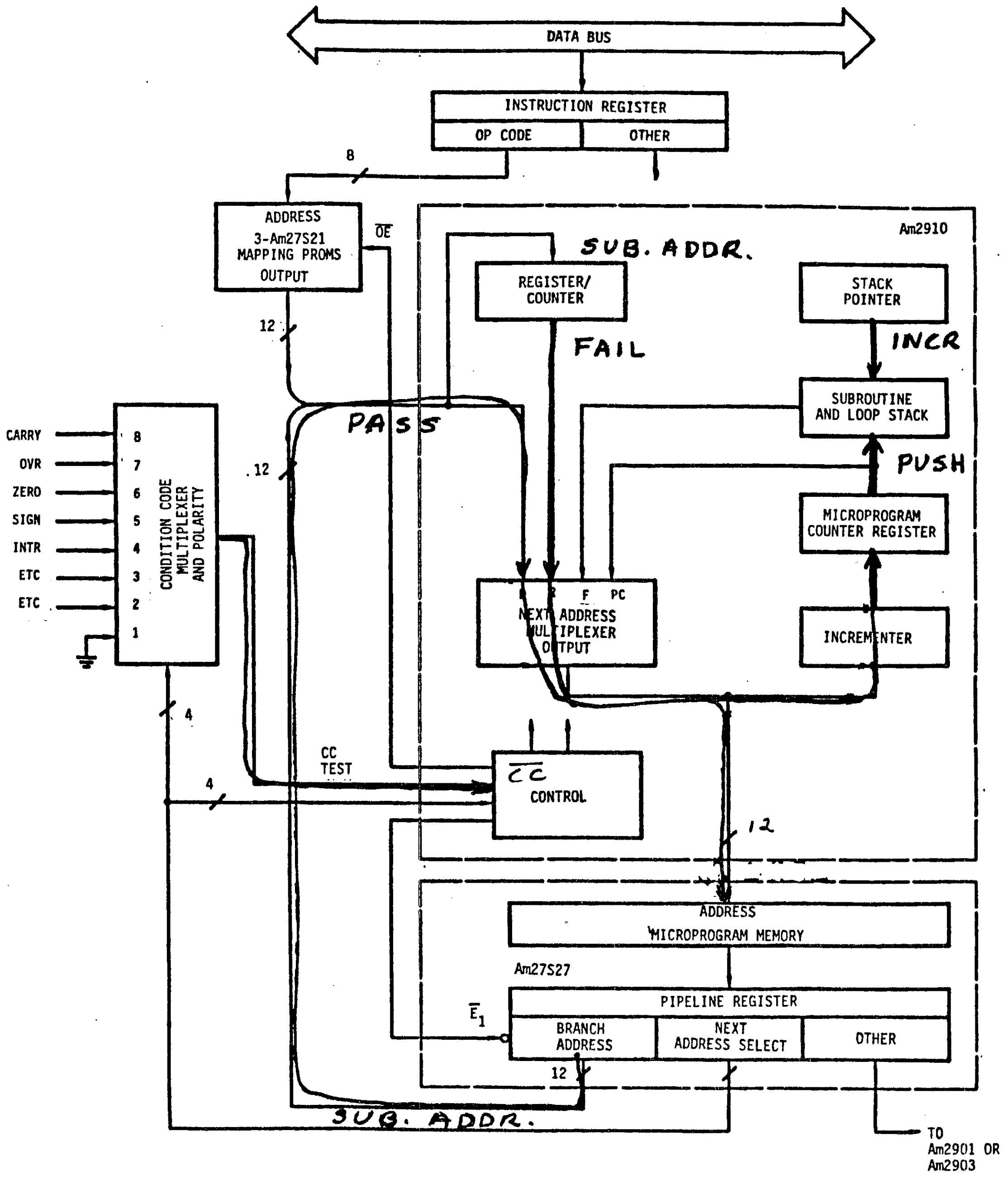


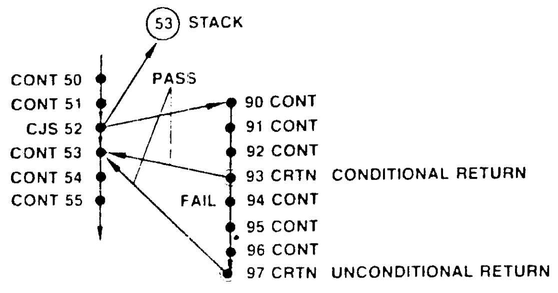
Figure 4–17. Conditional jump subroutine register/pipeline (JSRP, 5). LDCT or a register load must occur somewhere prior to JSRP.

JSRP


DATA BUS
INSTRUCTION REGISTER
OP CODE
OTHER
8
ADDRESS
3-Am27S21
MAPPING PROMS
OUTPUT
OE
12
Am2910
SUB. ADDR.
REGISTER/
COUNTER
STACK
POINTER
FAIL
INCR
SUBROUTINE
AND LOOP STACK
PASS
PUSH
MICROPROGRAM
COUNTER REGISTER
CARRY
OVR
ZERO
SIGN
INTR
ETC
ETC
8
7
6
5
4
3
2
1
CONDITION CODE
MULTIPLEXER
AND POLARITY
12
D
R
F
PC
NEXT ADDRESS
MULTIPLEXER
OUTPUT
INCREMENTER
4
CC
TEST
4
CC
CONTROL
12
ADDRESS
MICROPROGRAM MEMORY
Am27S27
PIPELINE REGISTER
E1
BRANCH
ADDRESS
NEXT
ADDRESS SELECT
OTHER
12
SUB. ADDR.
TO
Am2901 OR
Am2903

CRTN Conditional Return from Subroutine

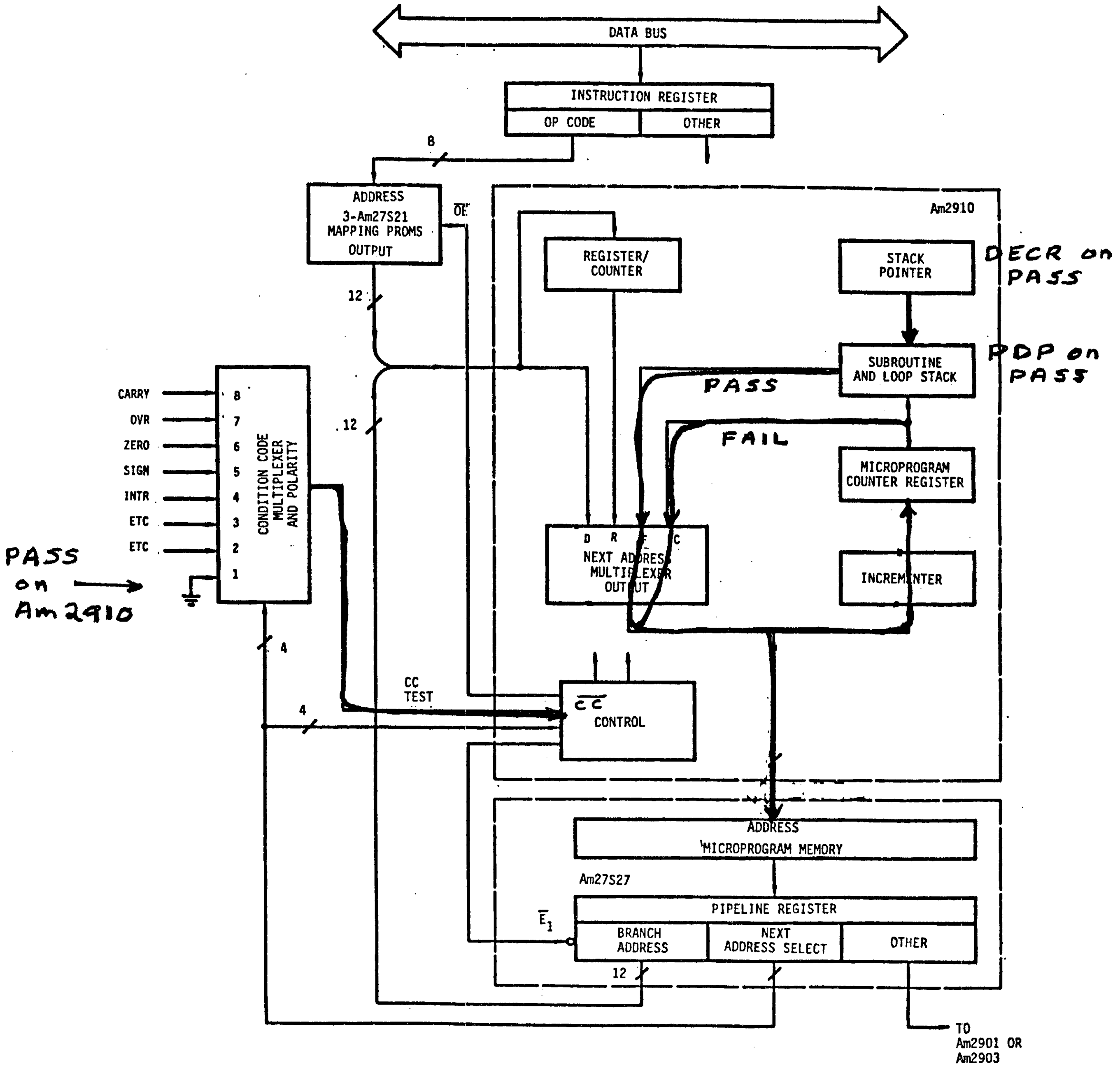
```
ADDRESS   LABEL    2910   COND    BRANCH
 (HEX)             INSTR  MUX     ADDRESS
_____________________________________________

  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	X	POP	STACK	NC	PL
FAIL		NC	μPC		
DISABLE ($\overline{CCEN}$ = H OR $\overline{CC}$ = L)	X	POP	STACK	NC	PL

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

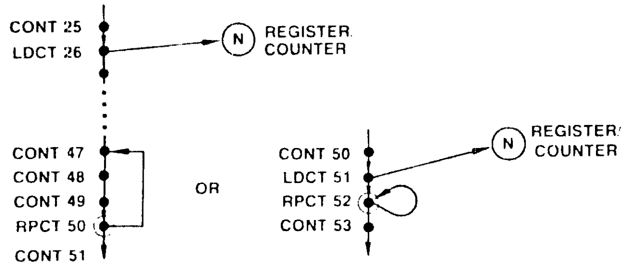
CRTN



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

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

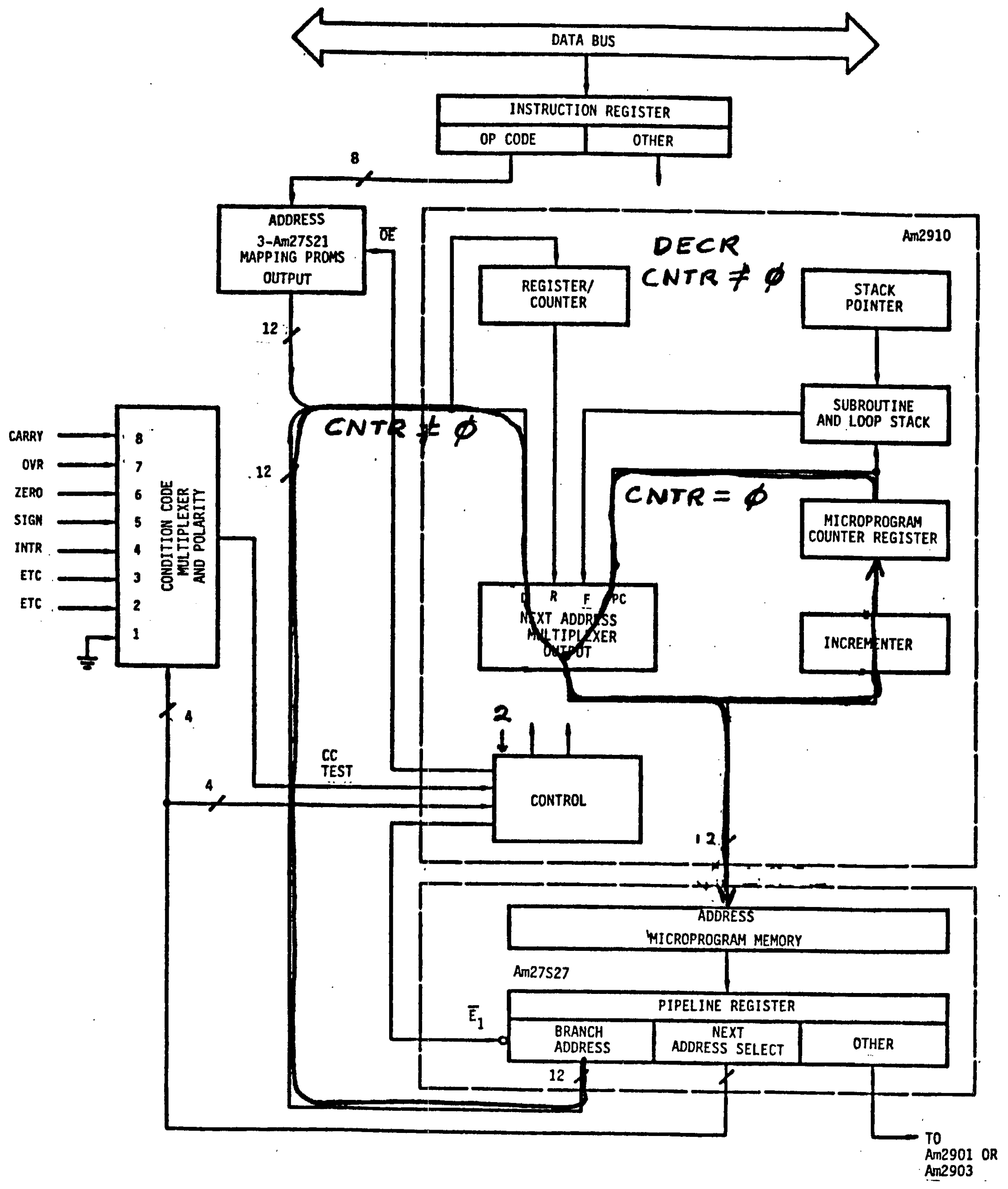
```
ADDRESS   LABEL    2910   COND   BRANCH
 (HEX)             INSTR  MUX    ADDRESS
-------------------------------------------------
  25               CONT   #      #
  26               LDCT   #      VAL-1
  27               CONT   #      #
   .
   .
   .
  47      BEGIN:   CONT   #      #
  48  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

RPCT


DATA BUS
INSTRUCTION REGISTER
OP CODE
OTHER
8
ADDRESS
3-Am27S21
MAPPING PROMS
OUTPUT
OE
12
Am2910
DECR
CNTR ≠ ∅
REGISTER/
COUNTER
STACK
POINTER
SUBROUTINE
AND LOOP STACK
CNTR ≠ ∅
CNTR = ∅
MICROPROGRAM
COUNTER REGISTER
CARRY
OVR
ZERO
SIGN
INTR
ETC
ETC
8
7
6
5
4
3
2
1
CONDITION CODE
MULTIPLEXER
AND POLARITY
D R F PC
NEXT ADDRESS
MULTIPLEXER
OUTPUT
INCREMENTER
4
2
CC
TEST
4
CONTROL
12
ADDRESS
MICROPROGRAM MEMORY
Am27S27
PIPELINE REGISTER
E1
BRANCH
ADDRESS
NEXT
ADDRESS SELECT
OTHER
12
TO
Am2901 OR
Am2903

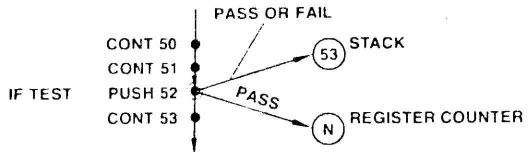
PUSH Push Microprocessor to TOS and Continue; Load Register/Counter Maybe

```
ADDRESS   LABEL   2910   COND   BRANCH
 (HEX)            INSTR  MUX    ADDRESS
__________________________________________

  50              CONT   #      #
  51              CONT   #      #
  52              PUSH   TESTH  VAL-1   <--- The result of TESTH only
  53              CONT   #      #             controls the Register Load
```

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.

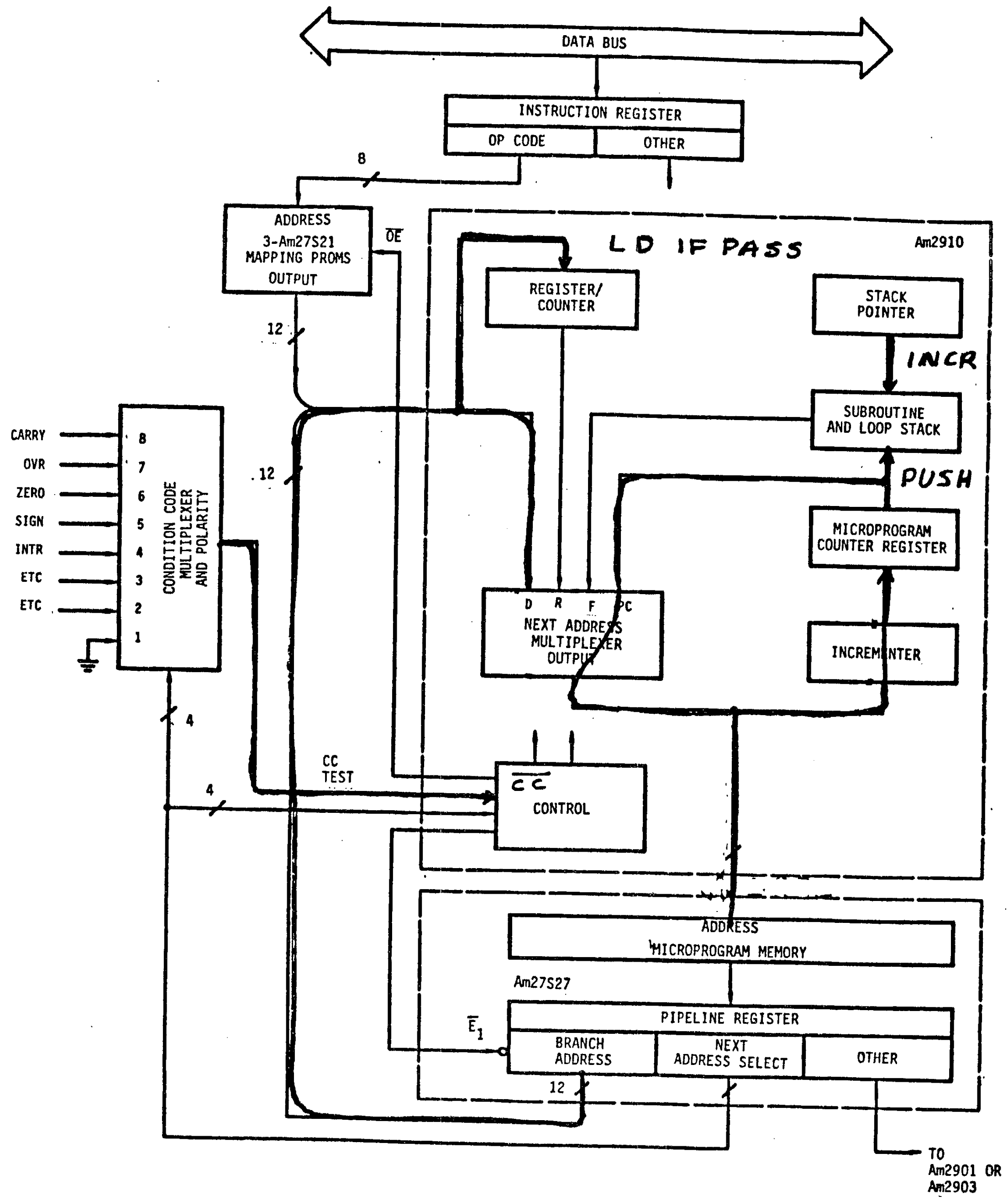
Instruction execution then continues.



$\overline{CC}$	COUNTER = 0 LINE	STACK	ADDRESS SOURCE	REGISTER/ COUNTER	$\overline{OE}$
PASS FAIL	X	PUSH	μPC	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.

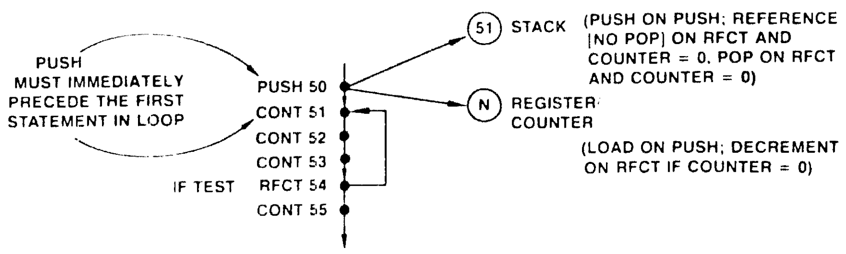
PUSH


DATA BUS
INSTRUCTION REGISTER
OP CODE
OTHER
8
ADDRESS
3-Am27S21
MAPPING PROMS
OUTPUT
OE
12
LD IF PASS
Am2910
REGISTER/
COUNTER
STACK
POINTER
INCR
SUBROUTINE
AND LOOP STACK
PUSH
MICROPROGRAM
COUNTER REGISTER
INCREMENTER
D R F PC
NEXT ADDRESS
MULTIPLEXER
OUTPUT
CARRY
OVR
ZERO
SIGN
INTR
ETC
ETC
8 7 6 5 4 3 2 1
CONDITION CODE
MULTIPLEXER
AND POLARITY
4
CC
TEST
CC
CONTROL
ADDRESS
MICROPROGRAM MEMORY
Am27S27
E1
PIPELINE REGISTER
BRANCH
ADDRESS
NEXT
ADDRESS SELECT
OTHER
12
TO
Am2901 OR
Am2903

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

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

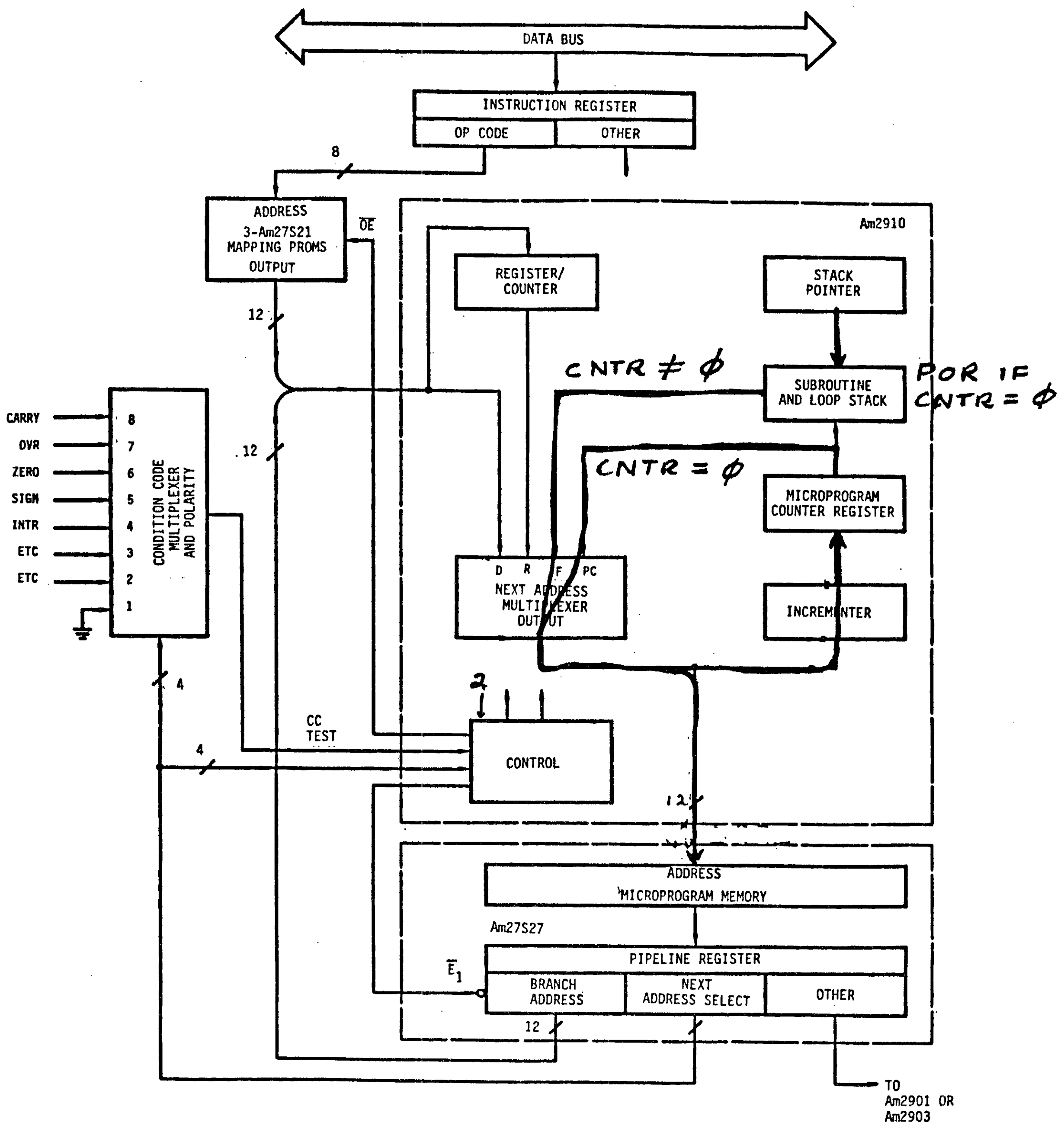
RFCT can also be used to form one-line loops.


PUSH
MUST IMMEDIATELY PRECEDE THE FIRST STATEMENT IN LOOP
PUSH 50
CONT 51
CONT 52
CONT 53
IF TEST
RFCT 54
CONT 55
51
STACK
(PUSH ON PUSH; REFERENCE (NO POP) ON RFCT AND COUNTER ≠ 0. POP ON RFCT AND COUNTER = 0)
N
REGISTER COUNTER
(LOAD ON PUSH; DECREMENT ON RFCT IF COUNTER ≠ 0)

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

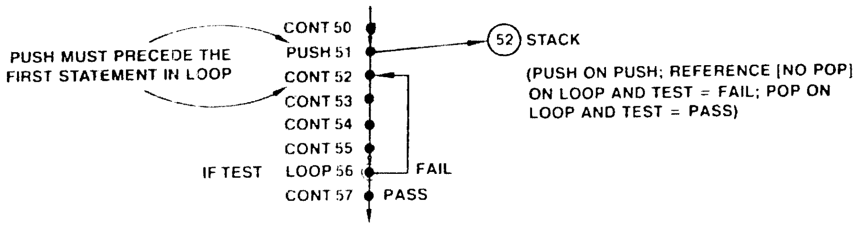
Figure 4-21. Repeat loop from stack if counter ≠ 0 (RFCT, 8).

RFCT


DATA BUS
INSTRUCTION REGISTER
OP CODE
OTHER
8
ADDRESS
3-Am27S21
MAPPING PROMS
OUTPUT
OE
Am2910
REGISTER/
COUNTER
STACK
POINTER
12
CNTR ≠ ϕ
SUBROUTINE
AND LOOP STACK
POR IF
CNTR = ϕ
CARRY
OVR
ZERO
SIGN
INTR
ETC
ETC
8
7
6
5
4
3
2
1
CONDITION CODE
MULTIPLEXER
AND POLARITY
12
CNTR = ϕ
MICROPROGRAM
COUNTER REGISTER
D R F PC
NEXT ADDRESS
MULTIPLEXER
OUTPUT
INCREMENTER
4
2
CC
TEST
4
CONTROL
12
ADDRESS
MICROPROGRAM MEMORY
Am27S27
E1
PIPELINE REGISTER
BRANCH
ADDRESS
NEXT
ADDRESS SELECT
OTHER
12
TO
Am2901 OR
Am2903

LOOP **Repeat Loop until TEST = TRUE**

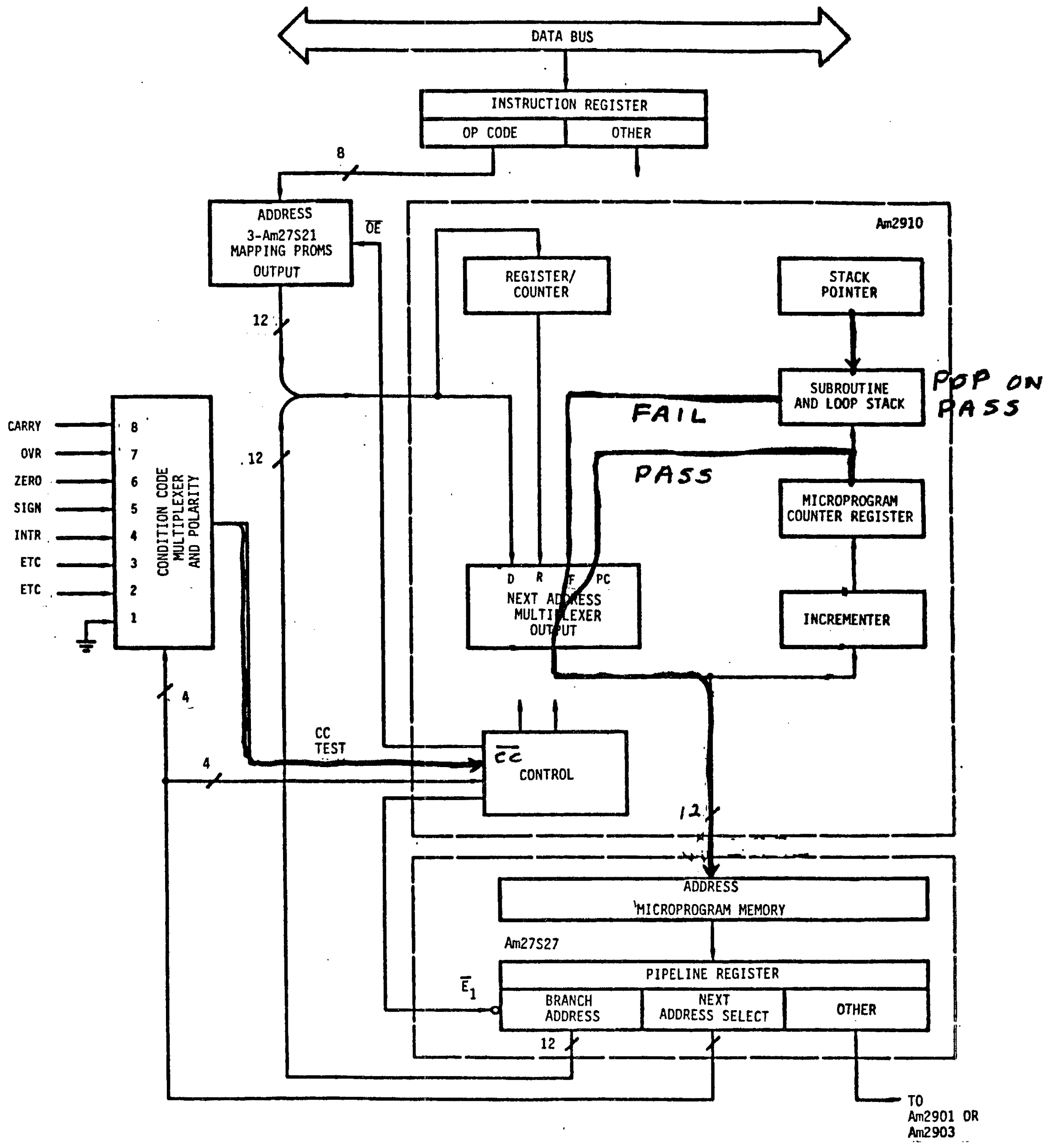
```
ADDRESS   LABEL   2910   COND   BRANCH
 (HEX)            INSTR  MUX    ADDRESS
_____________________________________________

  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   #      #
```


PUSH MUST PRECEDE THE FIRST STATEMENT IN LOOP
CONT 50
PUSH 51
CONT 52
CONT 53
CONT 54
CONT 55
IF TEST
LOOP 56
CONT 57
52
STACK
(PUSH ON PUSH; REFERENCE [NO POP] ON LOOP AND TEST = FAIL; POP ON LOOP AND TEST = PASS)
FAIL
PASS

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

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

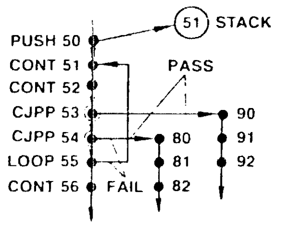
LOOP


DATA BUS
INSTRUCTION REGISTER
OP CODE
OTHER
8
ADDRESS
3-Am27S21
MAPPING PROMS
OUTPUT
OE
12
Am2910
REGISTER/
COUNTER
STACK
POINTER
SUBROUTINE
AND LOOP STACK
POP ON
PASS
FAIL
PASS
MICROPROGRAM
COUNTER REGISTER
INCREMENTER
CARRY
OVR
ZERO
SIGN
INTR
ETC
ETC
8
7
6
5
4
3
2
1
CONDITION CODE
MULTIPLEXER
AND POLARITY
12
D
R
F
PC
NEXT ADDRESS
MULTIPLEXER
OUTPUT
4
CC
TEST
4
CC
CONTROL
12
ADDRESS
MICROPROGRAM MEMORY
Am27S27
PIPELINE REGISTER
E1
BRANCH
ADDRESS
NEXT
ADDRESS SELECT
OTHER
12
TO
Am2901 OR
Am2903

CJPP Conditional Jump to Pipeline and POP TOS

(Use to exit from a loop which uses the stack)

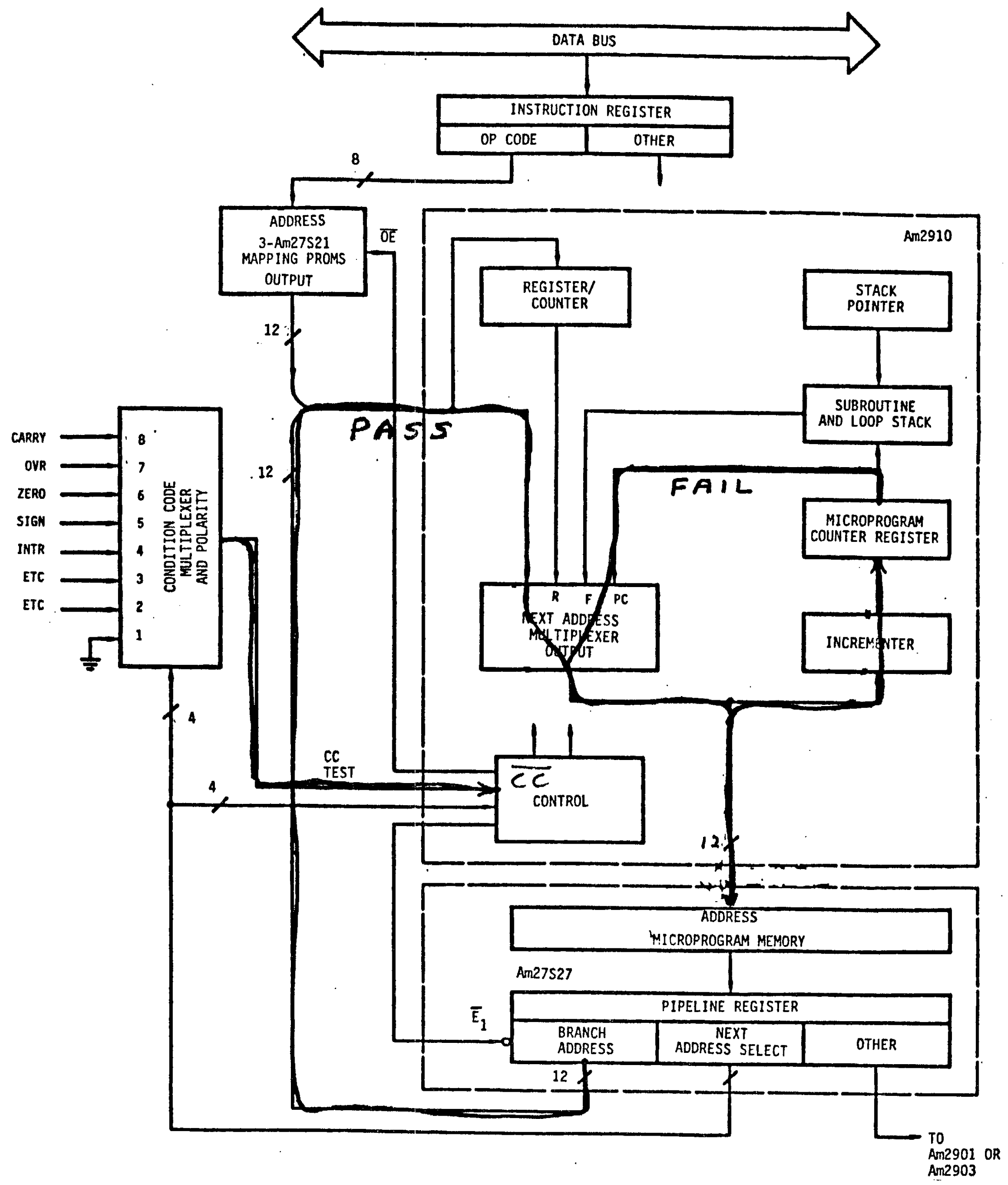
```
ADDRESS   LABEL  2910   COND   BRANCH
 (HEX)           INSTR  MUX    ADDRESS
_____________________________________________________

  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 μPC	NC	PL

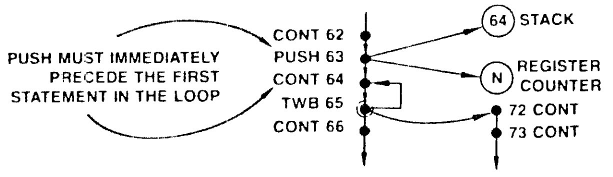
Figure 4–23. Conditional jump pipeline and POP (CJPP, B).

CJPP


DATA BUS
INSTRUCTION REGISTER
OP CODE
OTHER
8
ADDRESS
3-Am27S21
MAPPING PROMS
OUTPUT
OE
12
Am2910
REGISTER/
COUNTER
STACK
POINTER
SUBROUTINE
AND LOOP STACK
PASS
FAIL
MICROPROGRAM
COUNTER REGISTER
CARRY
OVR
ZERO
SIGN
INTR
ETC
ETC
8
7
6
5
4
3
2
1
CONDITION CODE
MULTIPLEXER
AND POLARITY
12
R
F
PC
NEXT ADDRESS
MULTIPLEXER
OUTPUT
INCREMENTER
4
CC
TEST
CC
CONTROL
4
12
ADDRESS
MICROPROGRAM MEMORY
Am27S27
PIPELINE REGISTER
E1
BRANCH
ADDRESS
NEXT
ADDRESS SELECT
OTHER
12
TO
Am2901 OR
Am2903

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

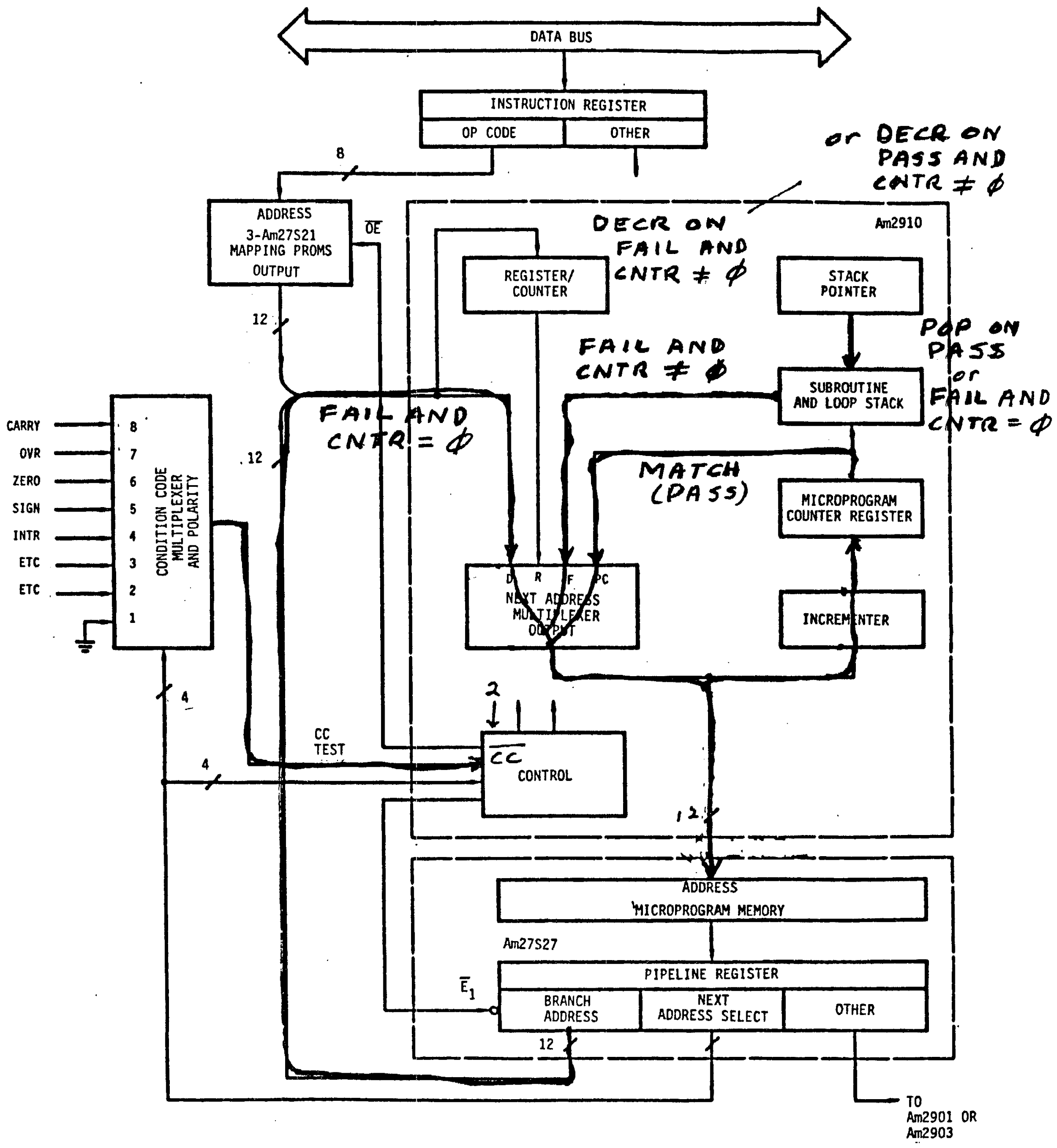
```
ADDRESS  LABEL   2910   COND   BRANCH
 (HEX)           INSTR  MUX    ADDRESS
-------------------------------------------

  62             CONT   #      #
  63             PUSH   PASS   VAL-1
  64     BEGIN:  CONT   #      #
  65             TWB    TESTM  ADRM
  66             CONT   #      #
  .
  .
  .
  72     ADRM:   CONT   #      #
  73             CONT   #      #
```



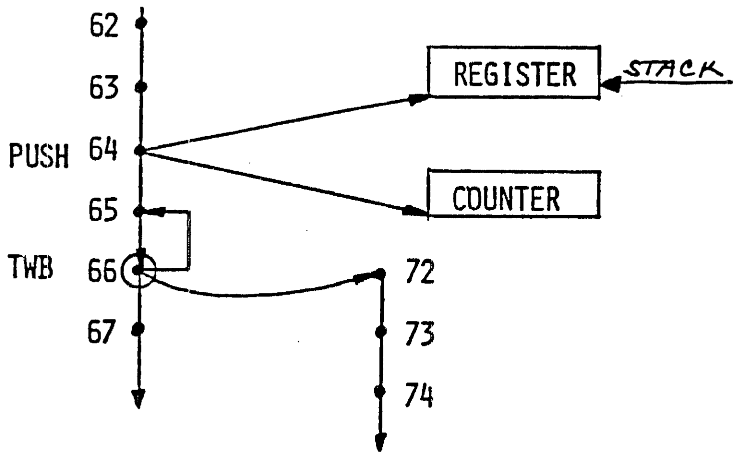
$\overline{CC}$	COUNTER = 0 LINE	STACK	ADDRESS SOURCE	REGISTER COUNTER	$\overline{OE}$
PASS	= 0	POP	μPC	NC	PL
	≠ 0			DECREMENT	
FAIL	= 0	POP	D	NC	
	≠ 0	NC	STACK	DECREMENT	

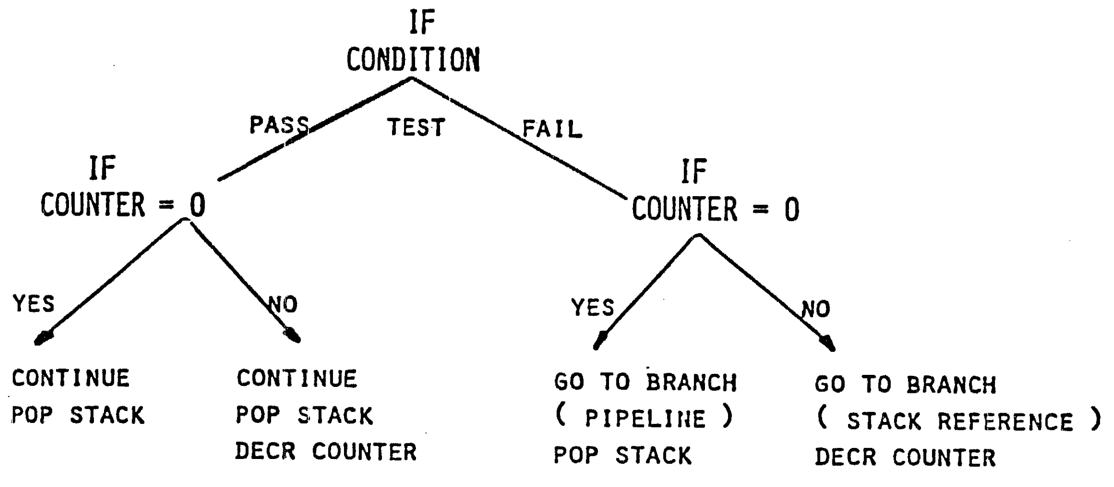
Figure 4–24. Three-way branch (TWB. F).

TWB

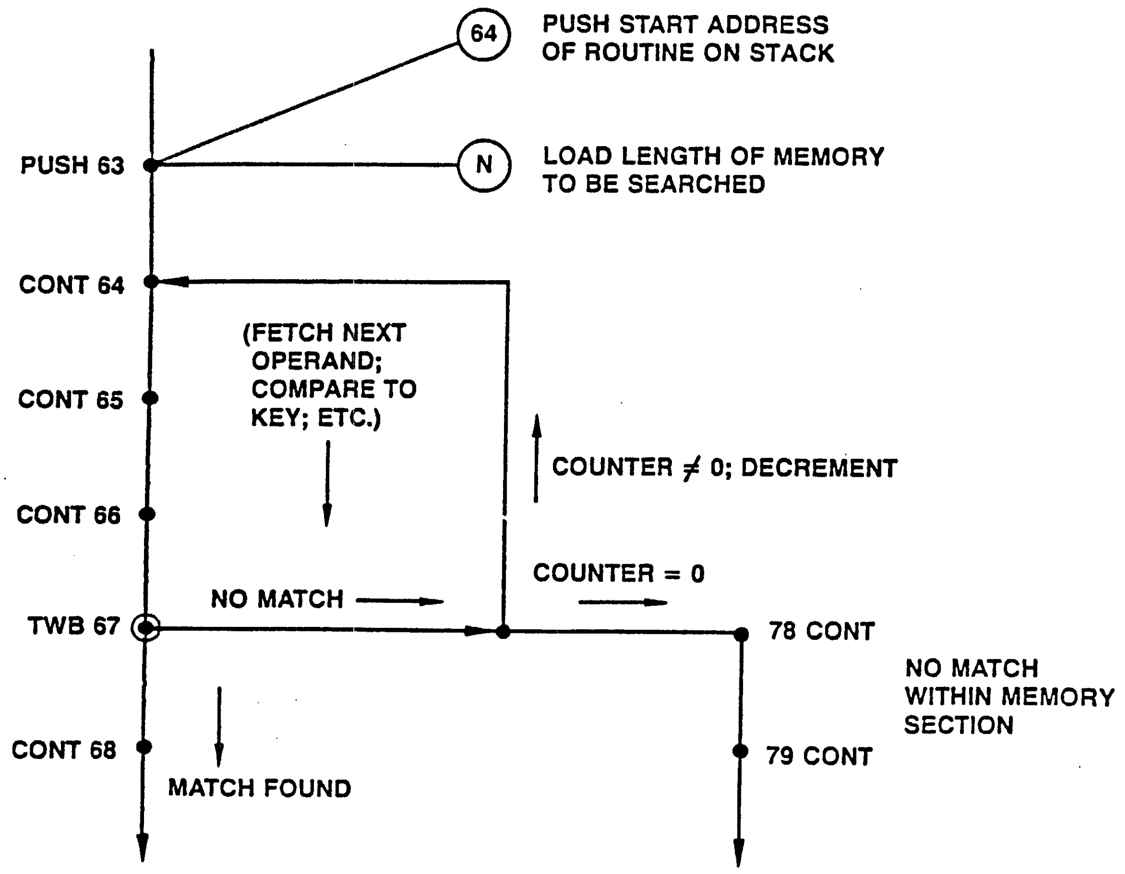

DATA BUS
INSTRUCTION REGISTER
OP CODE
OTHER
8
ADDRESS 3-Am27S21 MAPPING PROMS OUTPUT
OE
12
or DECR ON PASS AND CNTR ≠ ∅
DECR ON FAIL AND CNTR ≠ ∅
Am2910
REGISTER/ COUNTER
STACK POINTER
FAIL AND CNTR ≠ ∅
POP ON PASS or FAIL AND CNTR = ∅
SUBROUTINE AND LOOP STACK
FAIL AND CNTR = ∅
MATCH (PASS)
MICROPROGRAM COUNTER REGISTER
CARRY
OVR
ZERO
SIGN
INTR
ETC
ETC
8
7
6
5
4
3
2
1
CONDITION CODE MULTIPLEXER AND POLARITY
D R F PC
NEXT ADDRESS MULTIPLEXER OUTPUT
INCREMENTER
4
2
CC TEST
CC
CONTROL
4
12
ADDRESS
MICROPROGRAM MEMORY
Am27S27
PIPELINE REGISTER
E1
BRANCH ADDRESS
NEXT ADDRESS SELECT
OTHER
12
TO Am2901 OR Am2903

POWERFUL THREE-WAY BRANCHING





EXAMPLE OF THREE-WAY BRANCH


64
PUSH START ADDRESS OF ROUTINE ON STACK
N
LOAD LENGTH OF MEMORY TO BE SEARCHED
PUSH 63
CONT 64
(FETCH NEXT OPERAND; COMPARE TO KEY; ETC.)
CONT 65
COUNTER ≠ 0; DECREMENT
CONT 66
COUNTER = 0
NO MATCH
TWB 67
78 CONT
NO MATCH WITHIN MEMORY SECTION
CONT 68
79 CONT
MATCH FOUND

Special Pins on Am2910

$\overline{\text{RLD}}$ **Register Load**

- For the basic instruction set, $\overline{\text{RLD}}$ is held high
- For causing the register to load on the ↑ clock transition, regardless of the instruction, $\overline{\text{RLD}}$ 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 (<u>not normally under pipeline control</u> or you may have an infinite loop!)

Special Pins on Am2910

$\overline{\text{CCEN}}$	**Condition Code Enable** • $\overline{\text{CCEN}}$ = LOW; enables $\overline{\text{CC}}$ (TEST) input to operate normally • $\overline{\text{CCEN}}$ = HIGH; all conditional instructions are <u>unconditionally true</u> (TEST = PASS)
$\overline{\text{OE}}$	Tri-state control of Y_I outputs
$\overline{\text{FULL}}$	Five items are on stack; use in diagnostic test programs; debug

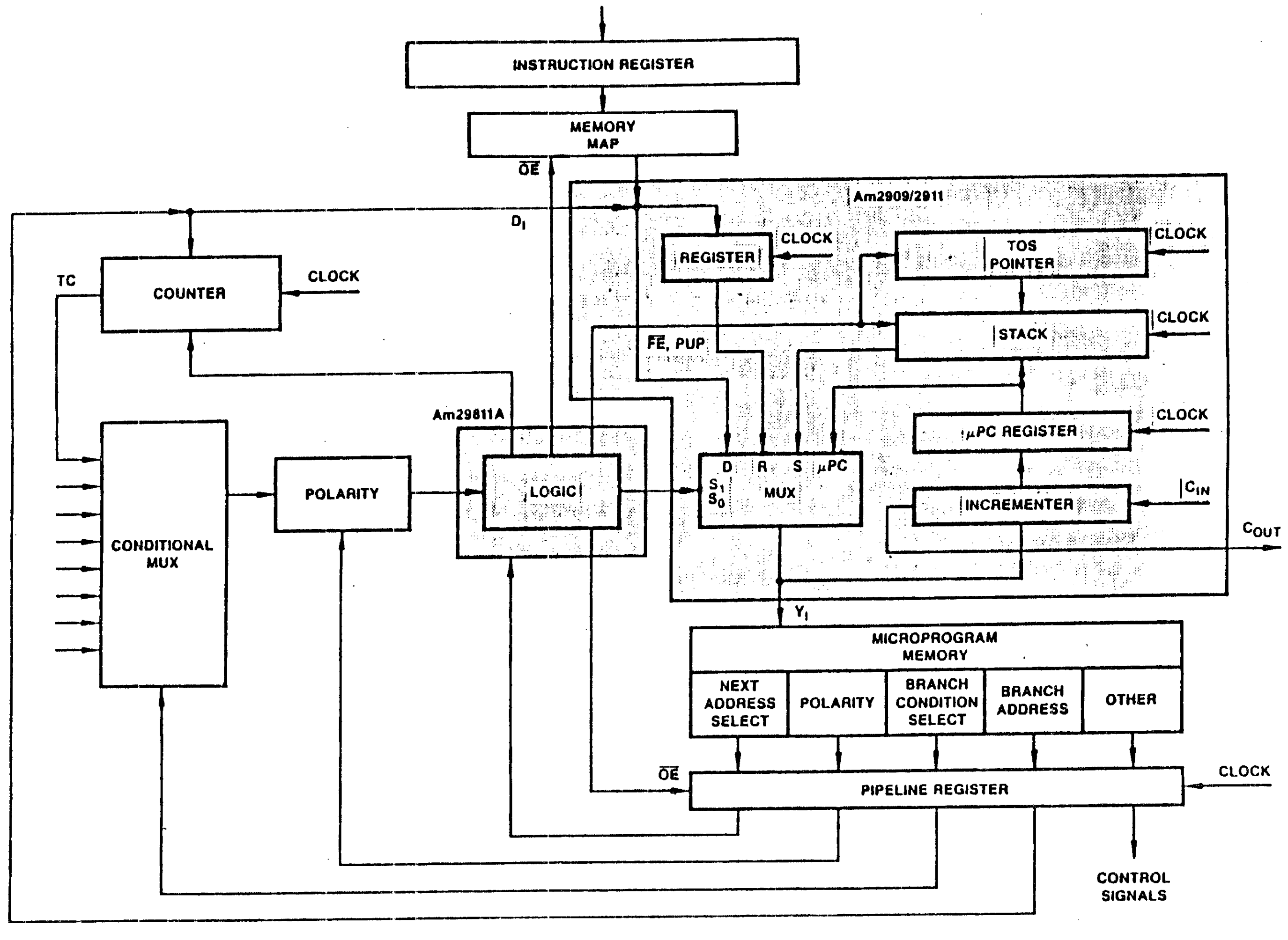
NEXT ADDRESS CONTROL

Am29811

MICROPROGRAMMED SEQUENCERS

Am2909 / Am2911

Am2909/2911


INSTRUCTION REGISTER
MEMORY MAP
OE
Am2909/2911
D_I
REGISTER
CLOCK
TOS POINTER
CLOCK
TC
COUNTER
CLOCK
STACK
CLOCK
FE, PUP
Am29811A
μPC REGISTER
CLOCK
D R S μPC
S1 S0 MUX
LOGIC
CONDITIONAL MUX
POLARITY
INCREMENTER
C_IN
C_OUT
Y_I
MICROPROGRAM MEMORY
NEXT ADDRESS SELECT
POLARITY
BRANCH CONDITION SELECT
BRANCH ADDRESS
OTHER
OE
PIPELINE REGISTER
CLOCK
CONTROL SIGNALS

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.

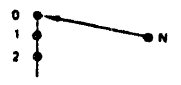
- Separate register (Ri) and direct (Di) inputs on Am2909A for flexibility.

- Am2909A and Am2911A speeds are comparable to Am2910. (See Data Book)

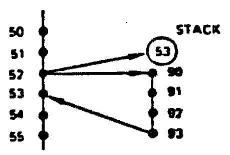
- Could replace Am29811A with ROM for customer instruction set.

Am29811

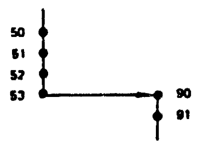
0 JUMP ZERO (JZ)



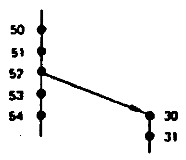
1 COND JSB PL (CJS)



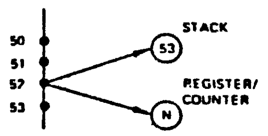
2 JUMP MAP (JMAP)



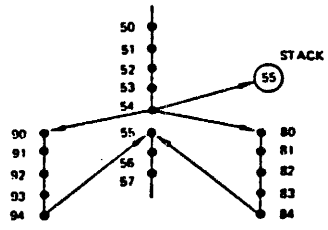
3 COND JUMP PL (CJP)



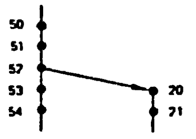
4 PUSH/COND LD CNTR (PUSH)



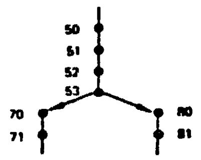
5 COND JSB R/PL (JSRP)



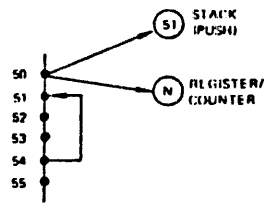
6 COND JUMP VECTOR (CJV)



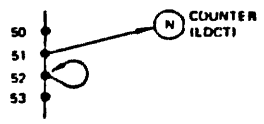
7 COND JUMP R/PL (JRP)



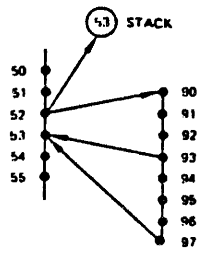
8 REPEAT LOOP, CNTR ≠ 0 (RFCT)



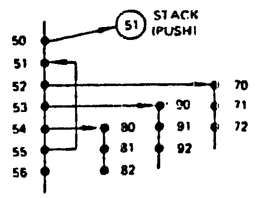
9 REPEAT PL, CNTR ≠ 0 (RPCT)



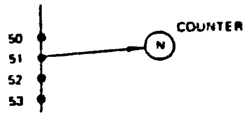
10 COND RETURN (CRTN)



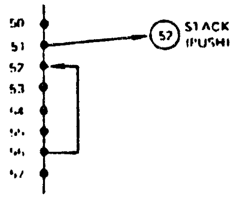
11 COND JUMP PL & POP (CJPP)



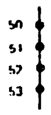
12 LD CNTR & CONTINUE (LDCT)



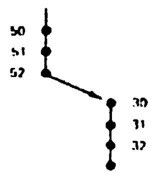
13 TEST END LOOP (LOOP)



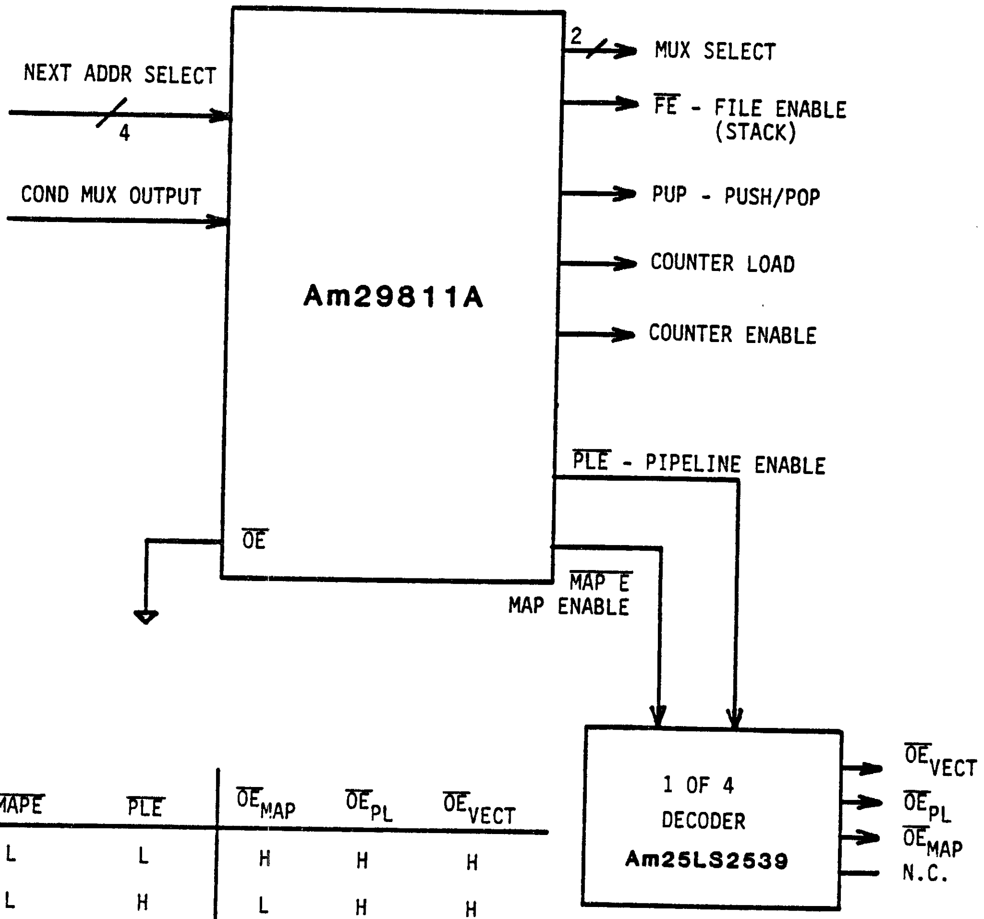
14 CONTINUE (CONT)



15 JUMP PIPELINE (JP)

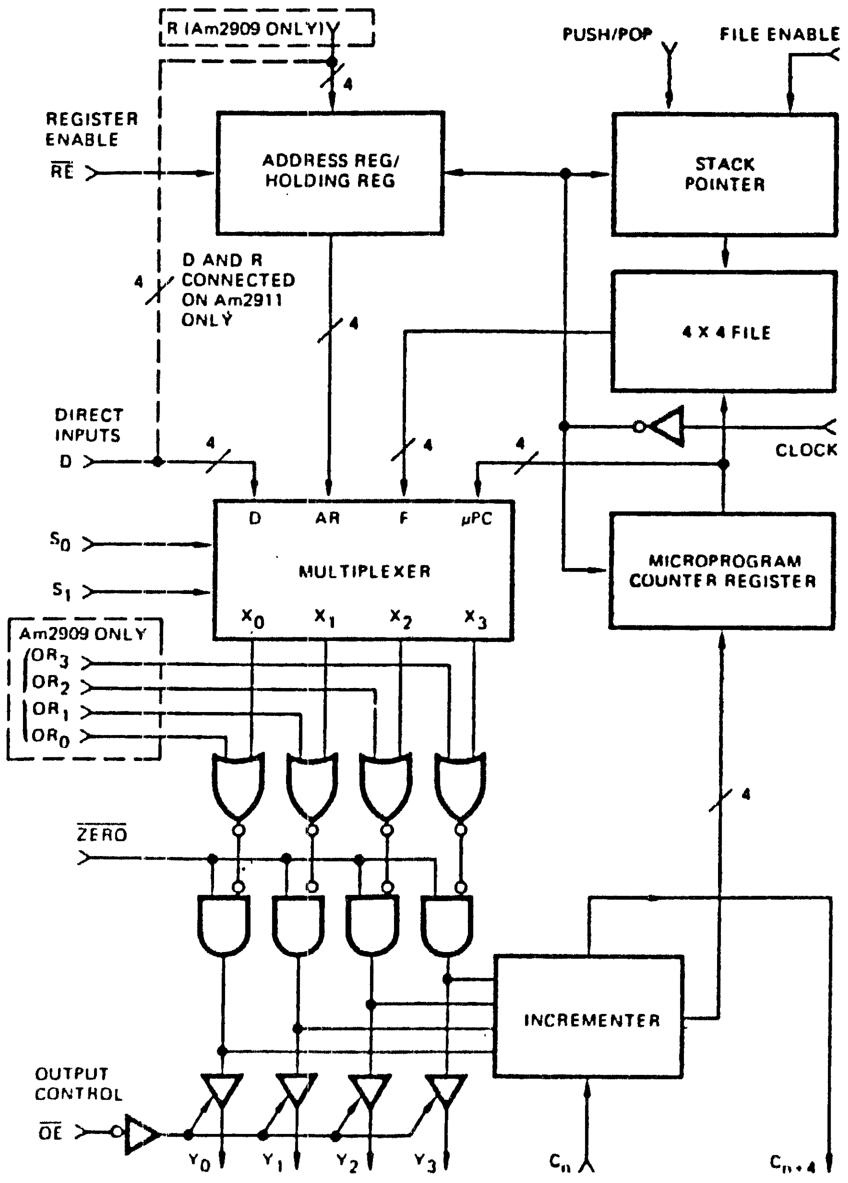


SUMMARY OF NEXT ADDRESS CONTROL LOGIC BLOCK



$\overline{MAPE}$	$\overline{PLE}$	$\overline{OE}_{MAP}$	$\overline{OE}_{PL}$	$\overline{OE}_{VECT}$
L	L	H	H	H
L	H	L	H	H
H	L	H	L	H
H	H	H	H	L

Microprogram Sequencer Block Diagram



2911	2909	2910	29112
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
$\overline{RE}$ = LOW loads reg	$\overline{RE}$ = LOW loads reg	$\overline{RLD}$ = 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

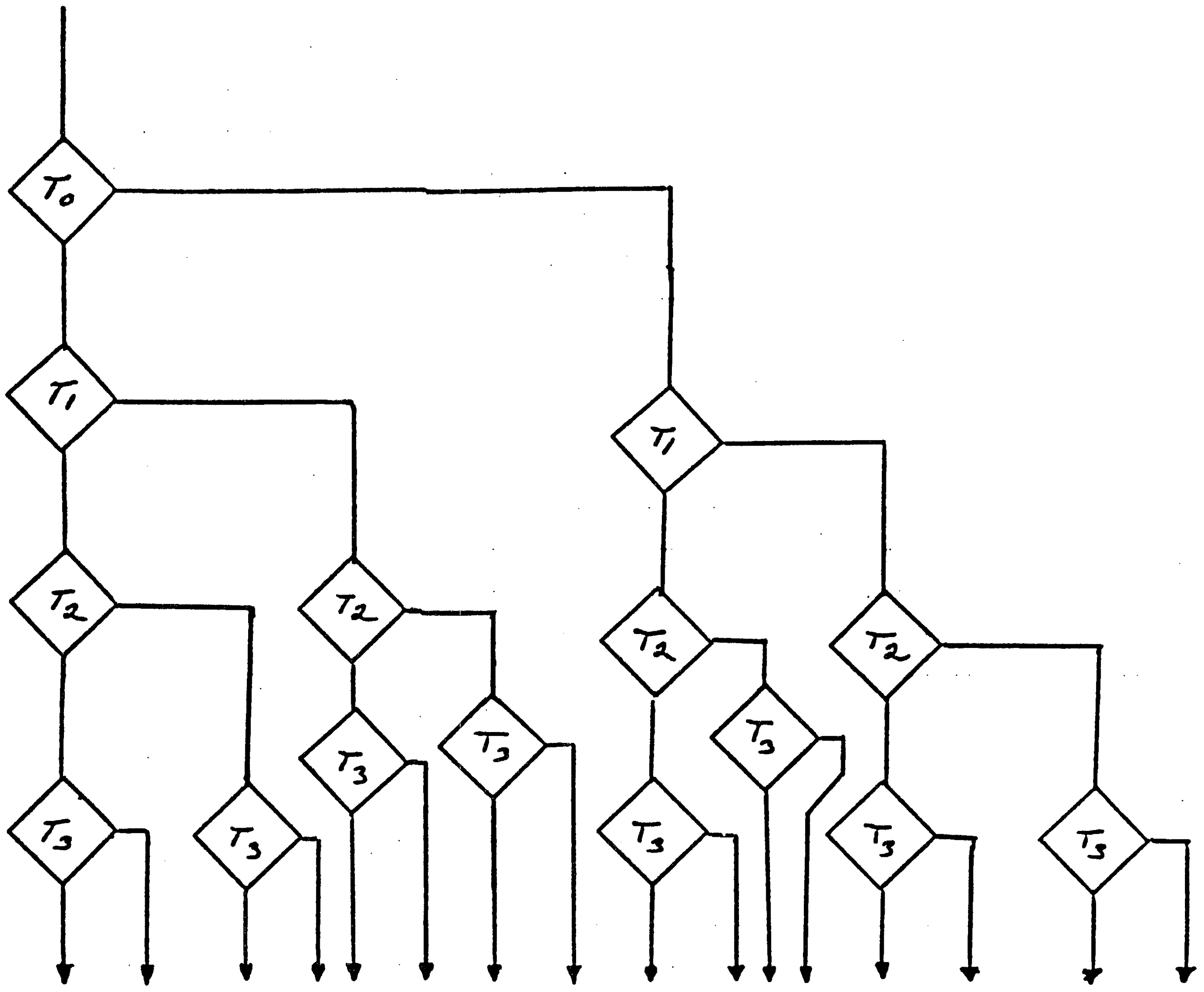
Am29803A

- 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

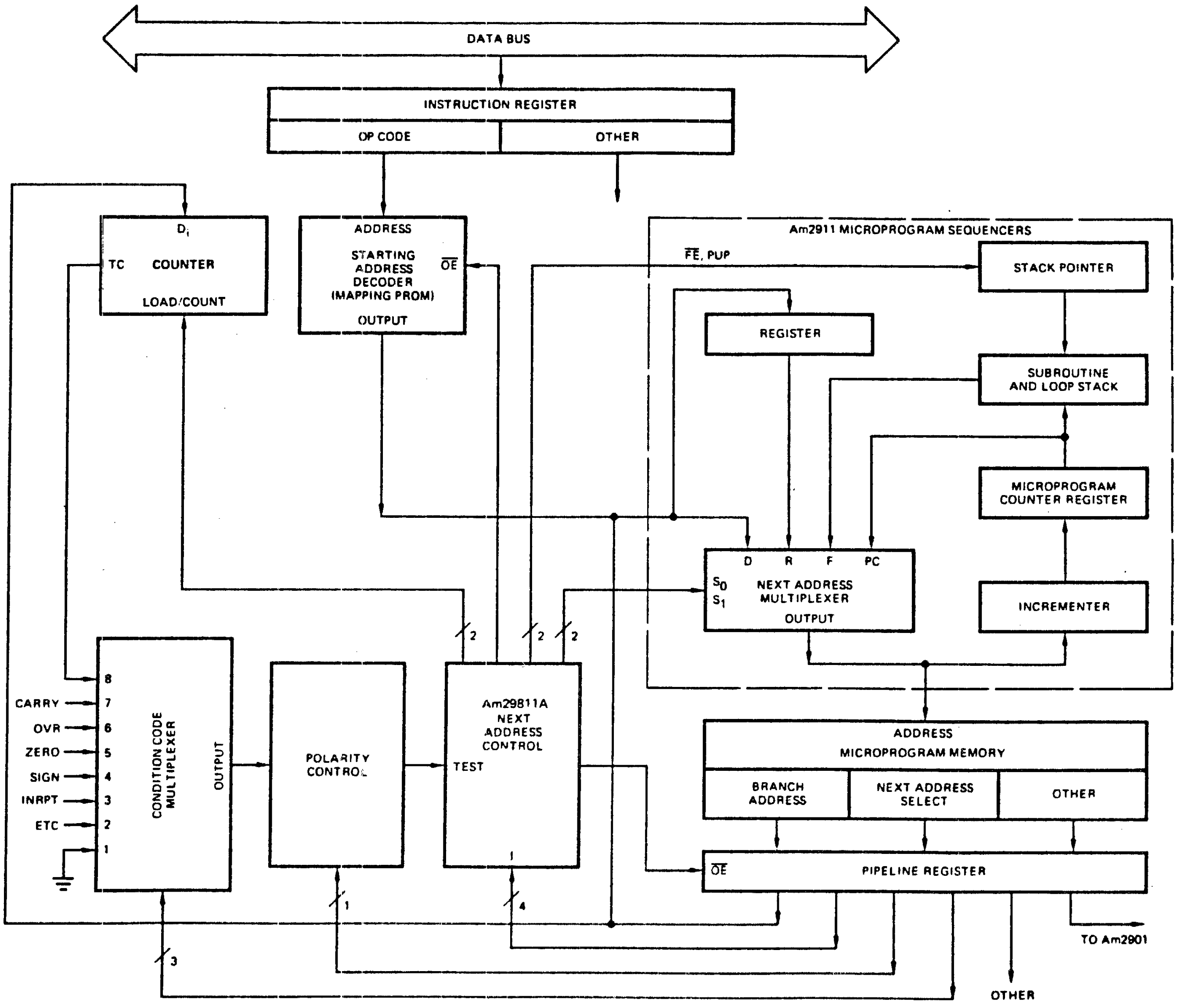
- Allows any combination of up to four tests (16-way branch) to be decoded in two microcycles.
- Faster than a series of conditional jumps and tests written in microcode.
- Easier for microprogramming.

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
					X	X	X	H	L	L	L	H
Test T_1	L	L	H	L	X	X	L	X	L	L	L	L
					X	X	H	X	L	L	L	H
Test T_0 & T_1	L	L	H	H	X	X	L	L	L	L	L	L
					X	X	L	H	L	L	L	H
					X	X	H	L	L	L	H	L
					X	X	H	H	L	L	H	H
Test T_2	L	H	L	L	X	L	X	X	L	L	L	L
					X	H	X	X	L	L	L	H
Test T_0 & T_2	L	H	L	H	X	L	X	L	L	L	L	L
					X	L	X	H	L	L	L	H
					X	H	X	L	L	L	H	L
					X	H	X	H	L	L	H	H
Test T_1 & T_2	L	H	H	L	X	L	L	X	L	L	L	L
					X	L	H	X	L	L	L	H
					X	H	L	X	L	L	H	L
					X	H	H	X	L	L	H	H
Test T_0, T_1 & T_2	L	H	H	H	X	L	L	L	L	L	L	L
					X	L	L	H	L	L	L	H
					X	L	H	L	L	L	H	L
					X	L	H	H	L	L	H	H
					X	H	L	L	L	H	L	L
					X	H	L	H	L	H	L	H
					X	H	H	L	L	H	H	L
					X	H	H	H	L	H	H	H
Test T_3	H	L	L	L	L	X	X	X	L	L	L	L
					H	X	X	X	L	L	L	H
Test T_0 & T_3	H	L	L	H	L	X	X	L	L	L	L	L
					L	X	X	H	L	L	L	H
					H	X	X	L	L	L	H	L
					H	X	X	H	L	L	H	H
Test T_1 & T_3	H	L	H	L	L	X	L	X	L	L	L	L
					L	X	H	X	L	L	L	H
					H	X	L	X	L	L	H	L
					H	X	H	X	L	L	H	H
Test T_0, T_1 & T_3	H	L	H	H	L	X	L	L	L	L	L	L
					L	X	L	H	L	L	L	H
					L	X	H	L	L	L	H	L
					L	X	H	H	L	L	H	H
					H	X	L	L	L	H	L	L
					H	X	L	H	L	H	L	H
					H	X	H	L	L	H	H	L
					H	X	H	H	L	H	H	H
Test T_2 & T_3	H	H	L	L	L	L	X	X	L	L	L	L
					L	H	X	X	L	L	L	H
					H	L	X	X	L	L	H	L
					H	H	X	X	L	L	H	H
Test T_0, T_2 & T_3	H	H	L	H	L	L	X	L	L	L	L	L
					L	L	X	H	L	L	L	H
					L	H	X	L	L	L	H	L
					L	H	X	H	L	L	H	H
					H	L	X	L	L	H	L	L
					H	L	X	H	L	H	L	H
					H	H	X	L	L	H	H	L
					H	H	X	H	L	H	H	H
Test T_1, T_2 & T_3	H	H	H	L	L	L	L	X	L	L	L	L
					L	L	H	X	L	L	L	H
					L	H	L	X	L	L	H	L
					L	H	H	X	L	L	H	H
					H	L	L	X	L	H	L	L
					H	L	H	X	L	H	L	H
					H	H	L	X	L	H	H	L
					H	H	H	X	L	H	H	H
Test T_0, T_1, T_2 & T_3	H	H	H	H	L	L	L	L	L	L	L	L
					L	L	L	H	L	L	L	H
					L	L	H	L	L	L	H	L
					L	L	H	H	L	L	H	H
					L	H	L	L	L	H	L	L
					L	H	L	H	L	H	L	H
					L	H	H	L	L	H	H	L
					L	H	H	H	L	H	H	H
					H	L	L	L	H	L	L	L
					H	L	L	H	H	L	L	H
					H	L	H	L	H	L	H	L
					H	L	H	H	H	L	H	H
					H	H	L	L	H	H	L	L
					H	H	L	H	H	H	L	H
					H	H	H	L	H	H	H	L
					H	H	H	H	H	H	H	H

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

A Typical CCU using the Am2909, Am2911, Am29803A and Am29811A



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, T11, 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 (T2TØ = ØØ) GO TO 10

        IF (T2TØ = Ø1) GO TO 200

        IF (T2TØ = 1Ø) GO TO 30

        IF (T2TØ = 11) GO TO 40
```

Am29803 SOLUTION

1.

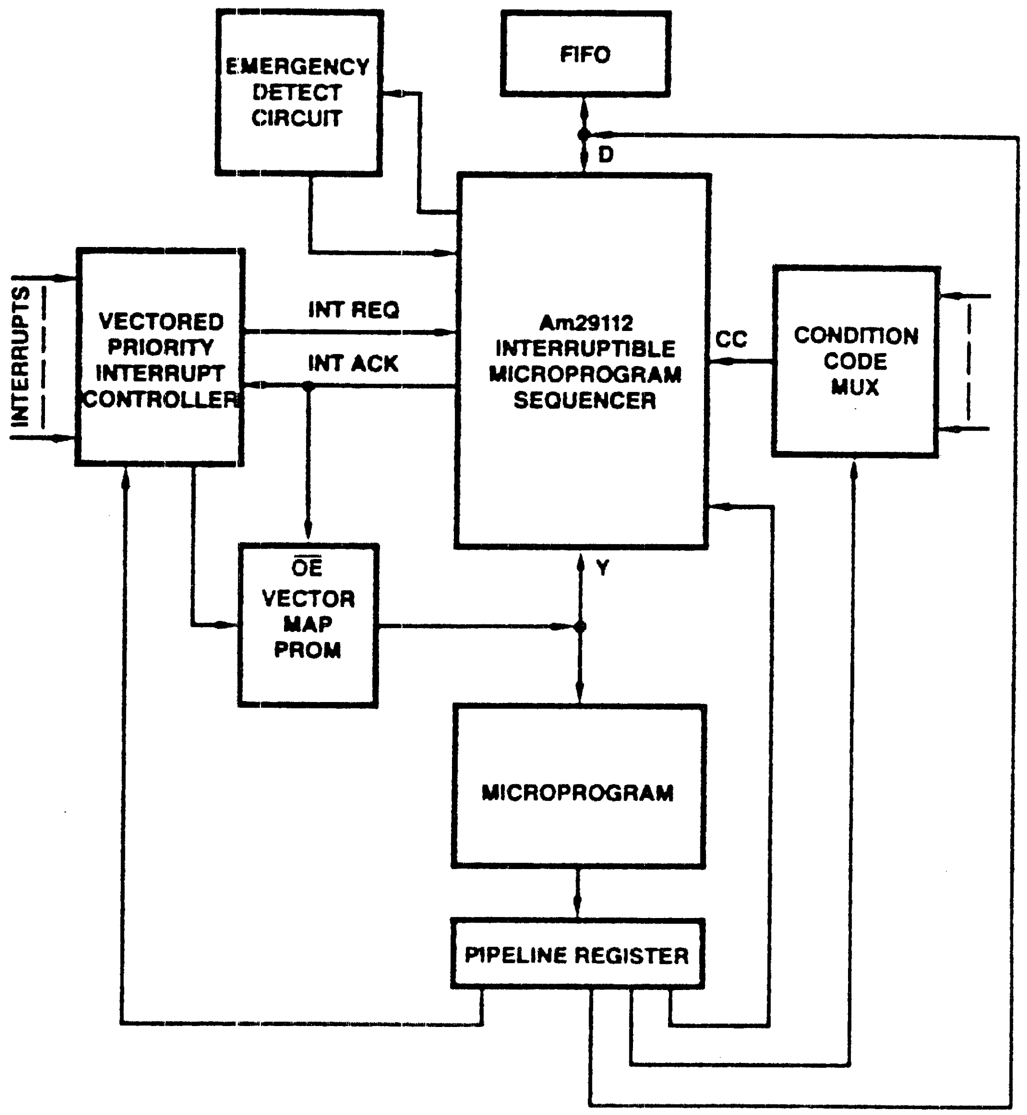
LABEL/ ADDR	29811 INSTR	MUX SEL	BR ADDR	29803 INSTR
i	CJP	A	i + 2	NO TEST
i + 1	JP	#	350*	T_3T_1
i + 2	JP	#	360*	$T_2\ T_Ø$
j	CJP	B	j + 2	NO TEST
j + 1	JP	#	370	$T_2T_1T_Ø$
j + 2	JP	#	380	$T_3T_2T_1$
350	JP	#	20	NO TEST
351	JP	#	200	NO TEST
352	JP	#	10	NO TEST
353	JP	#	20	NO TEST
360	JP	#	10	NO TEST
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
.				
.				
.				

* address must have Ø as final HEX digit if LSS 2909 attached to 29803

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.

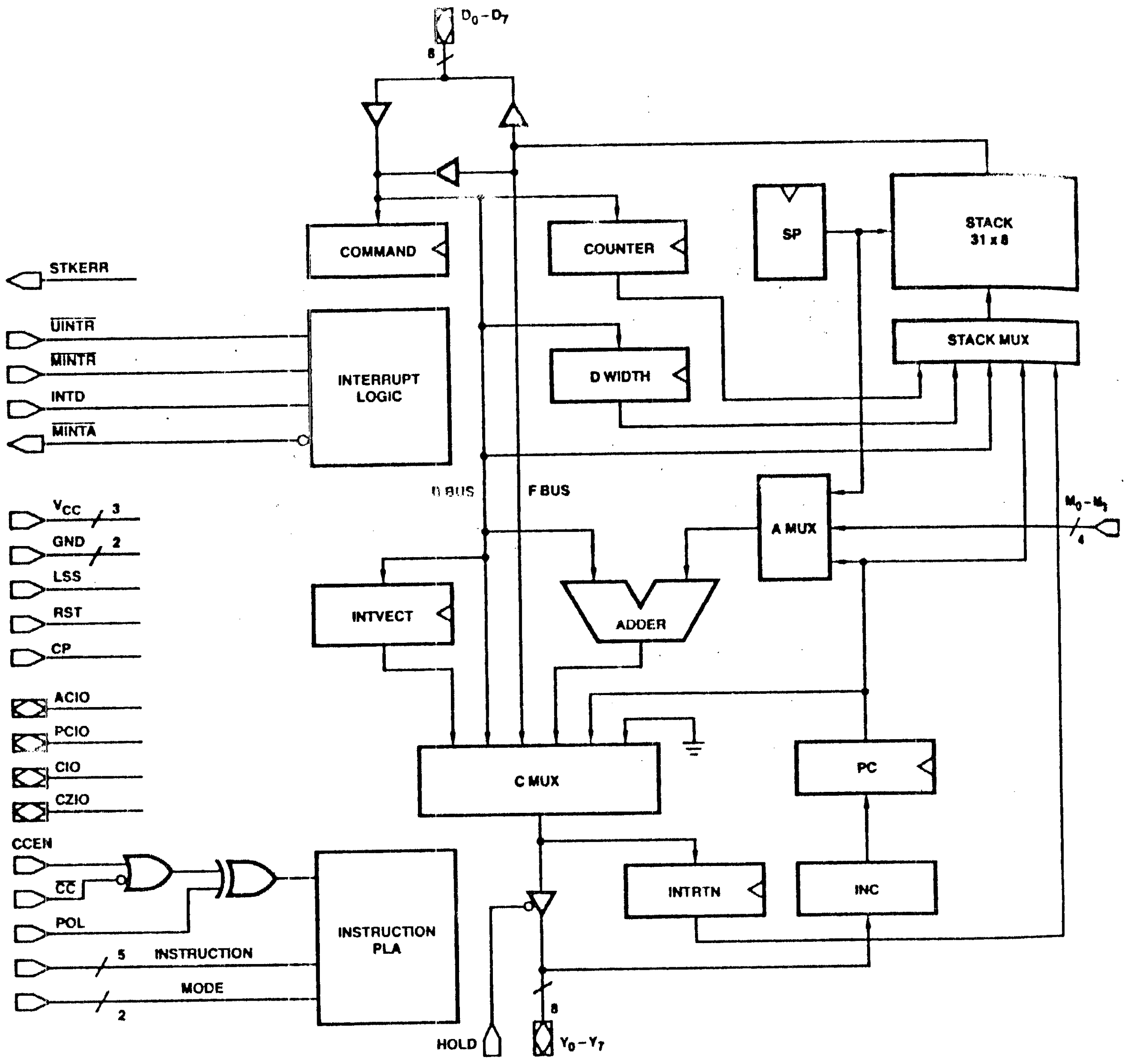
DISTINCTIVE CHARACTERISTICS:

- 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 (Ø,Ø), extended (Ø1) and forced continue (1Ø and 11). In the normal mode the entire instruction set of the Am29112 applies.

MODE CONTROLS

I_{65}	Mode	Description
ØØ	Normal	For cascaded Am29112s, two independent 8-bit counters
Ø1	Extended	For cascaded Am29112s, one 16-bit counter
1Ø	Forced	The Am29112 executes a continue instruction regardless of instruction,
11	Continue	condition code, and multiway inputs.

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

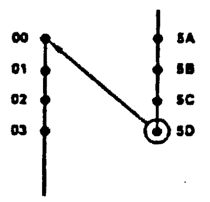
*These instructions are identical in the extended mode.
**These too.
***These too.

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.

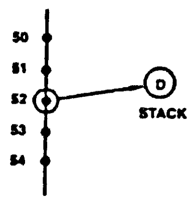
CC	CCEN	POL	Condition
X	1	0	PASS
X	1	1	FAIL
I	0	P	COND

0 Jump Zero (JZ.U)



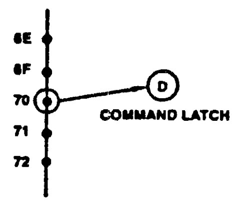
UNCONDITIONAL

1 Push D (PUSHD.P)



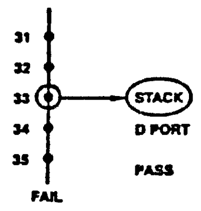
FORCED PASS

1 Load Command Latch from D (LDCMD.F)



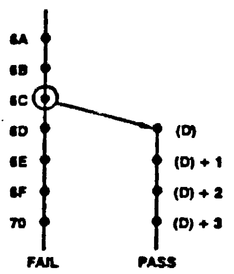
FORCED FAIL

2 Pop and Unconditional Stackout to D (POP.C)



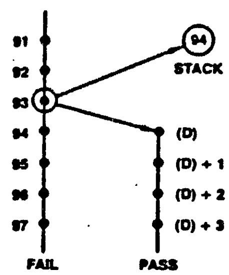
CONDITIONAL

3 Jump D (CJD. C)



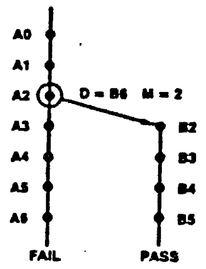
CONDITIONAL

4 Jump Subroutine D (CJSD.C)



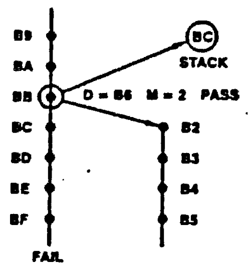
CONDITIONAL

5 Jump Multiway D (CJMW.C)



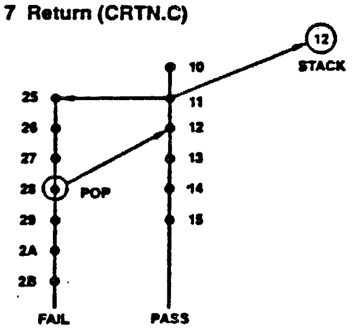
CONDITIONAL

6 Jump Subroutine Multiway D (CJSMW.C)



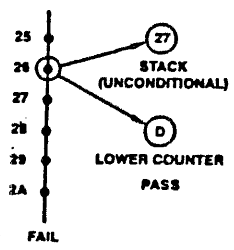
CONDITIONAL

7 Return (CRTN.C)



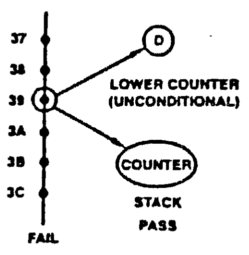
CONDITIONAL

8 Push PC and Conditional Load Lower Counter (PUSHPL.C)



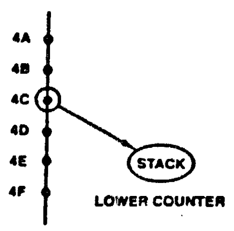
CONDITIONAL

9 Load Lower Counter and Conditional Push Counter (LDLC. C)



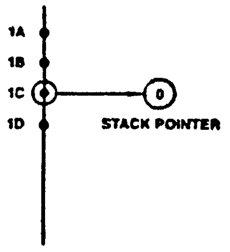
CONDITIONAL

10 Pop to Lower Counter (POPLC.P)



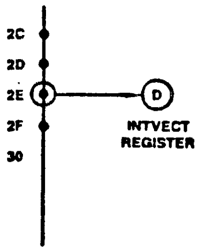
FORCED PASS

11 Reset Stack Pointer (RSTSP.P)



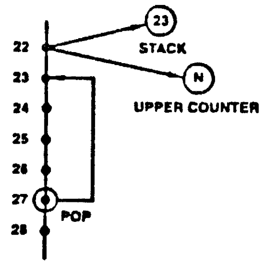
FORCED PASS

11 Load Unmaskable Interrupt Vector (LDINTV.F)



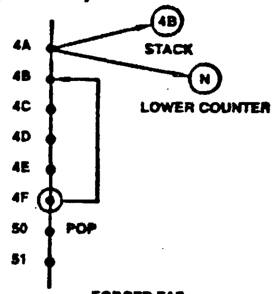
FORCED FAIL

12 Repeat Loop, Upper Counter (RFCTU.P)



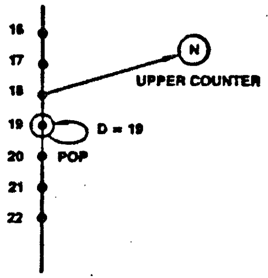
FORCED PASS

12 Repeat Loop, Lower Counter (RFCTL.F)



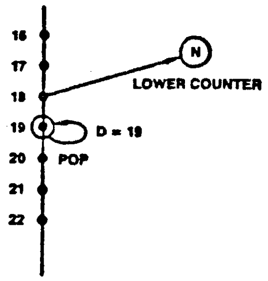
FORCED FAIL

13 Repeat Pipeline, Upper Counter (RPCTL.P)



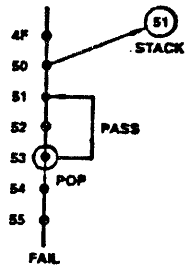
FORCED PASS

13 Repeat Pipeline, Lower Counter (RPCTL.F)



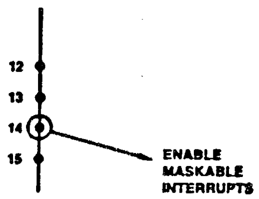
FORCED FAIL

14 Test End Loop (LOOP.C)



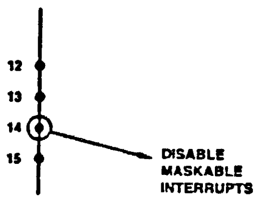
CONDITIONAL

15 Enable Interrupts (ENINT.P)



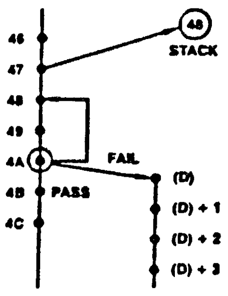
FORCED PASS

15 Disable Interrupts (DISINT.F)



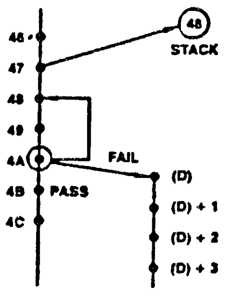
FORCED FAIL

16 Three-Way Branch, Lower Counter (TWBL.C)



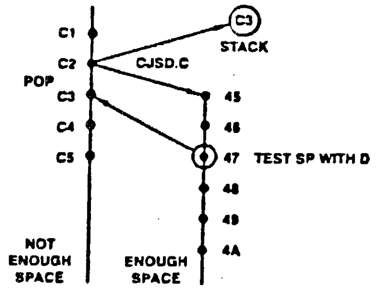
CONDITIONAL

17 Three-Way Branch, Upper Counter (TWBU.C)



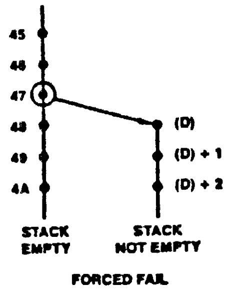
CONDITIONAL

18 Test SP with D (TSTSP.P)

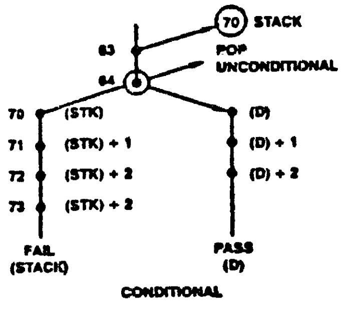


FORCED PASS

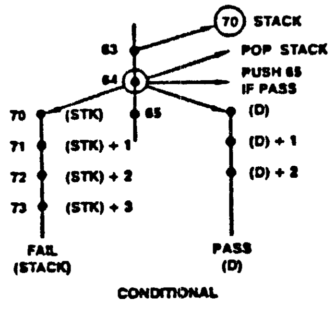
18 Jump D If Stack Not Empty (TSTMT.F)



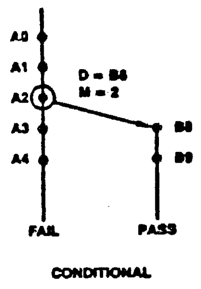
19 Conditional Jump D/Stack and Pop (CJDF.C)



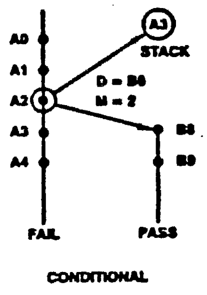
20 Conditional Jump Subroutine D/Stack and Pop (CJSDF.C)



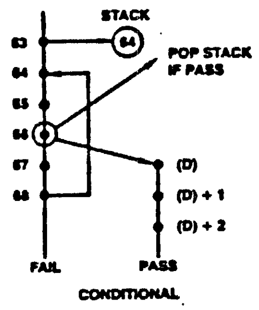
21 Conditional Jump Multiway Relative D (CJMWR.C)



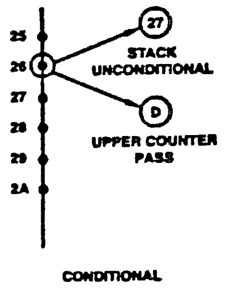
22 Conditional Jump Subroutine Multiway Relative D (CJSMWR.C)



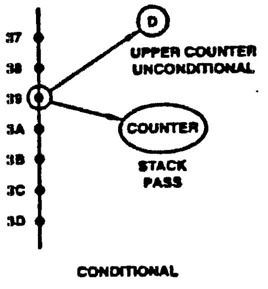
23 Conditional Jump Pipeline and Pop (CJPP)



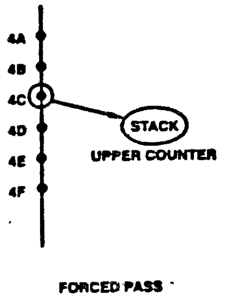
24 Push PC and Conditional Load Upper Counter (PUSHPU.C)



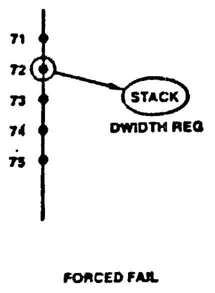
25 Load Upper Counter and Conditional Push Counter (LDUC.C)



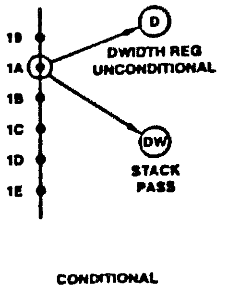
26 Pop to Upper Counter (POPUC.P)



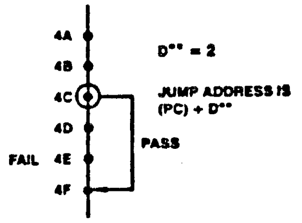
26 Pop to Displacement Width (POPDW.F)



27 Load Displacement Width and Conditional Push DW (LDDW.C)



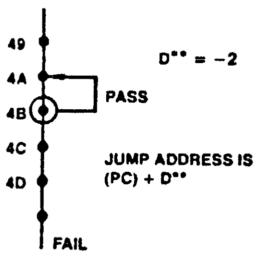
28 Conditional Jump D PC Relative (CJR.C)



D** is displacement (see 1).

CONDITIONAL

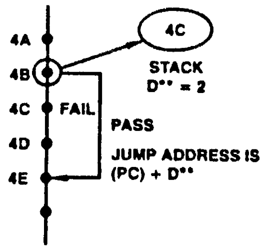
29 Conditional Jump D PC Relative Negative (CJRN.C)



$D^{**} = -2$, should be two's complement (see 2).

CONDITIONAL

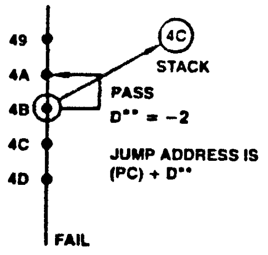
30 Conditional Jump Subroutine D PC Relative (CJSR.C)



D^{**} is displacement (see 1).

CONDITIONAL

31 Conditional Jump Subroutine D PC Relative Negative (CJSRN.C)



$D^{**} = -2$, should be two's complement (see 2).

CONDITIONAL

Notes: 1. The number of bits of D used as displacement is stored in DWIDTH register. The remaining high order bits are zero-extended.
2. The number of bits of D used as displacement is stored in DWIDTH register. The remaining high order bits are one-extended.

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 EXCERCISE

- 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 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.

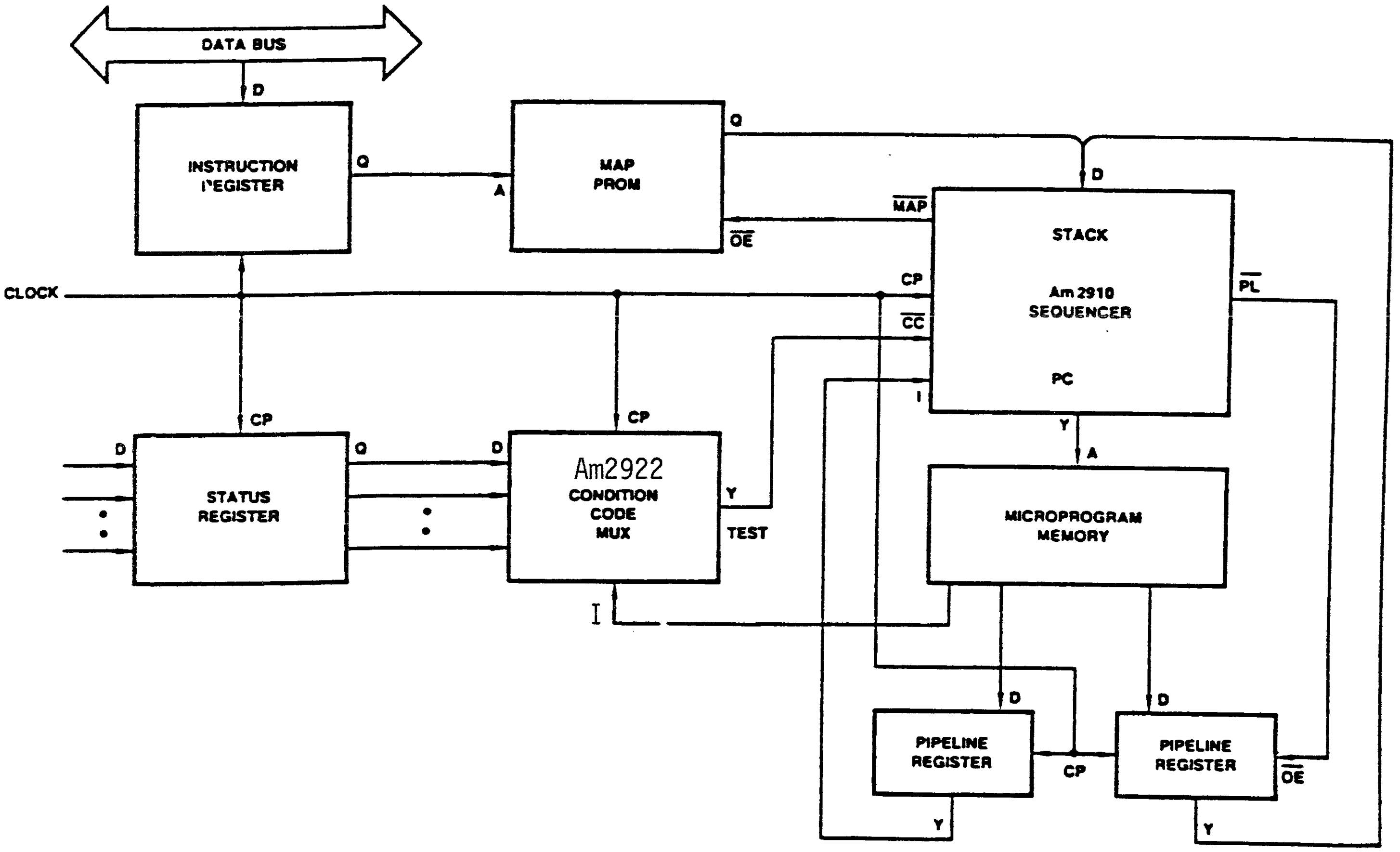
- Although the ALU is not shown, a similar timing analysis must be conducted for its paths for a complete design.

- 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.


DATA BUS
D
INSTRUCTION REGISTER
Q
A
MAP PROM
Q
D
MAP
OE
STACK
Am 2910 SEQUENCER
PC
CP
CC
I
PL
CLOCK
CP
D
STATUS REGISTER
Q
D
CP
Am2922 CONDITION CODE MUX
Y
TEST
I
Y
A
MICROPROGRAM MEMORY
D
D
PIPELINE REGISTER
CP
PIPELINE REGISTER
OE
Y
Y

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°C, V_{CC} = 4.75 to 5.25V, C_L = 50pF)

A. Set-up and Hold Times

Input	t_s	t_h
D → R	24	6
D_i → PC	58	4
I_0-I_3	**75**	0
$\overline{CC}$	**63**	0
$\overline{CCEN}$	**63**	0
CI	46	5
$\overline{RLD}$	36	6

B. Combinational Delays

Input	Y	$\overline{PL}$, $\overline{VECT}$, $\overline{MAP}$	$\overline{Full}$
D_0-D_{11}	20	-	-
I_0-I_3	**50**	51	-
$\overline{CC}$	**30**	-	-
$\overline{CCEN}$	**30**	-	-
CP (Note 2) I = 8, 9, 15	**75**	-	60
	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	
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°C, V_{CC} = 4.5 to 5.5V, C_L = 50pF)

A. Set-up and Hold Times

Input	t_s	t_h
D_i → R	28	6
D_i → PC	62	4
I_0-I_3	**81**	0
$\overline{CC}$	**65**	0
$\overline{CCEN}$	**63**	0
CI	58	5
$\overline{RLD}$	42	6

B. Combinational Delays

Input	Y	$\overline{PL}$, $\overline{VECT}$, $\overline{MAP}$	$\overline{Full}$
D_0-D_{11}	25	-	-
I_0-I_3	**54**	58	-
$\overline{CC}$	**35**	-	-
$\overline{CCEN}$	**37**	-	-
CP (Note 2) I = 8, 9, 15	**77**	-	67
	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	
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 = 5.0pF.

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

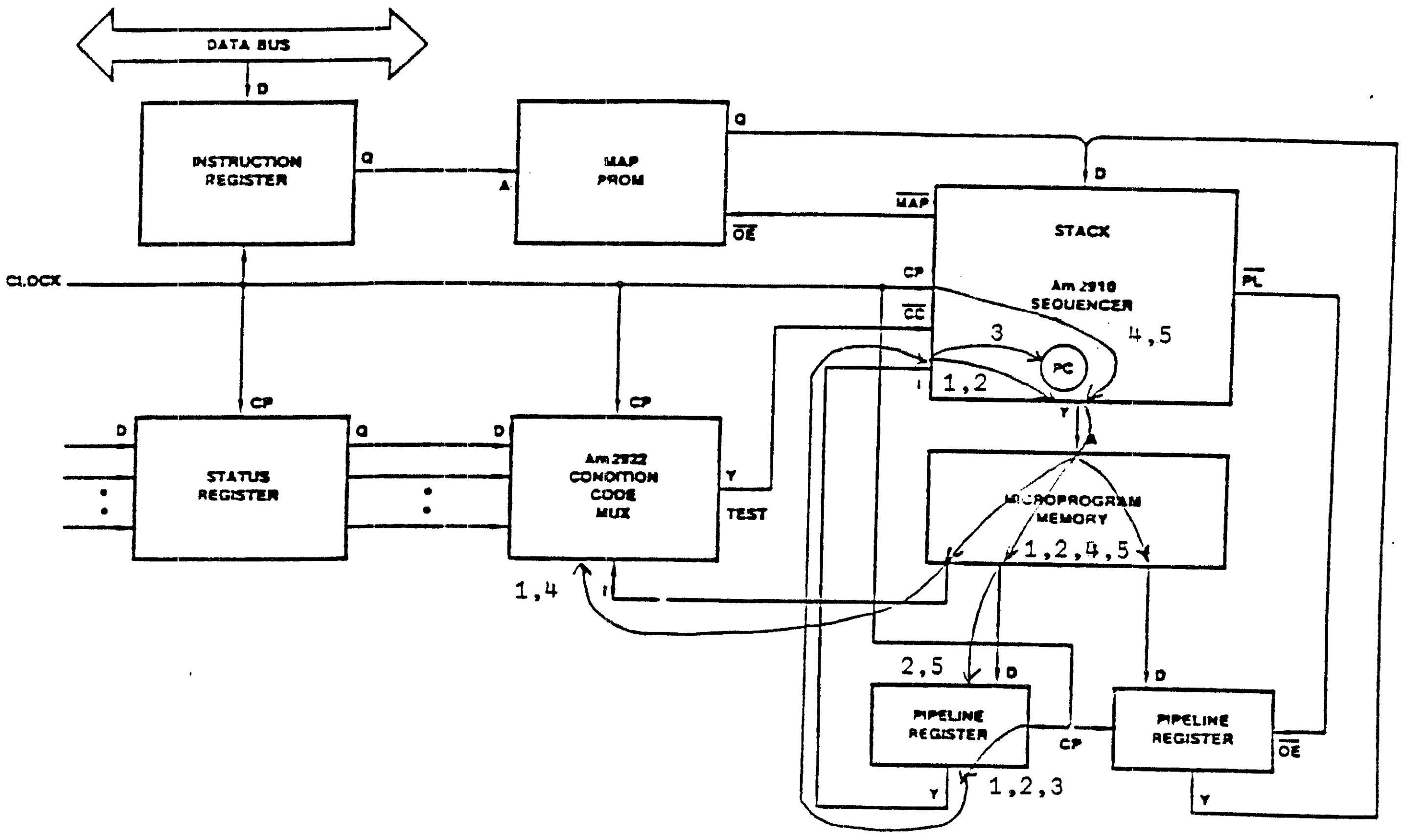
- On **PERT chart**, assign worst-case times to each block.

- Add up times along each path.

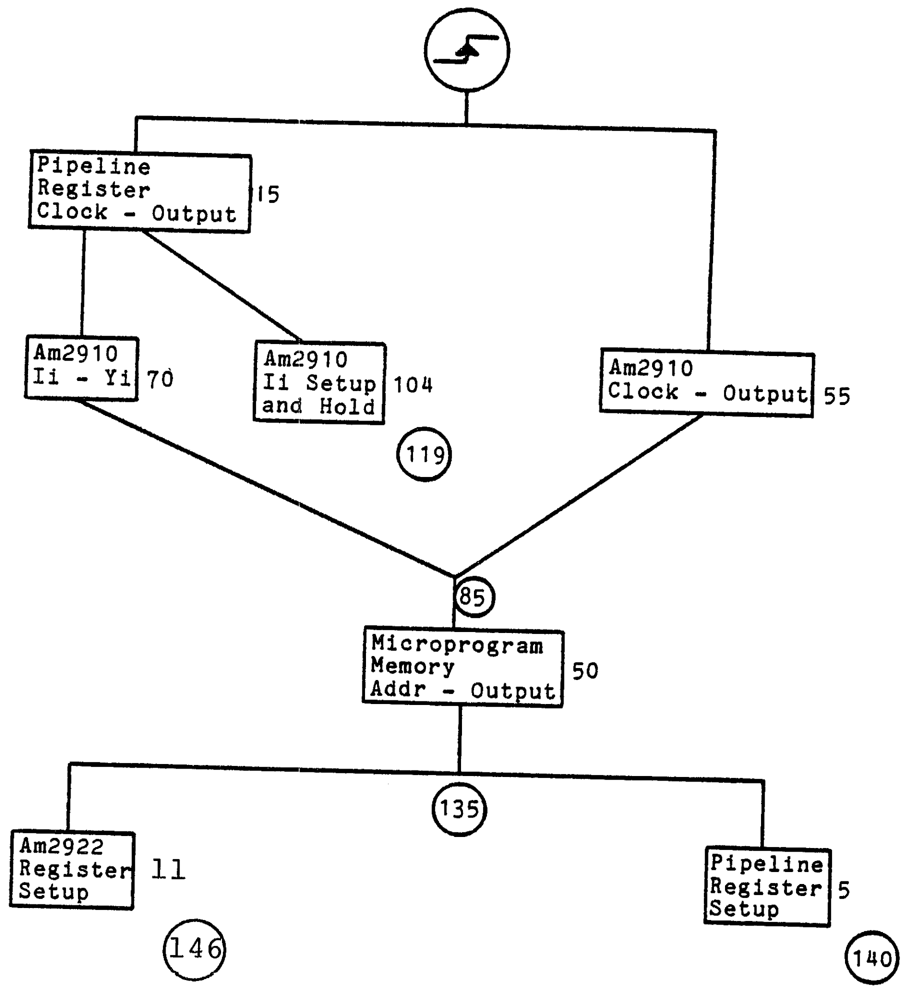
- For PERT chart, converging paths must all be satisfied, hence use maximum time at that point (e.g. address input to micromemory).

- Maximum path defines minimum clock cycle possible.

CONTINUE



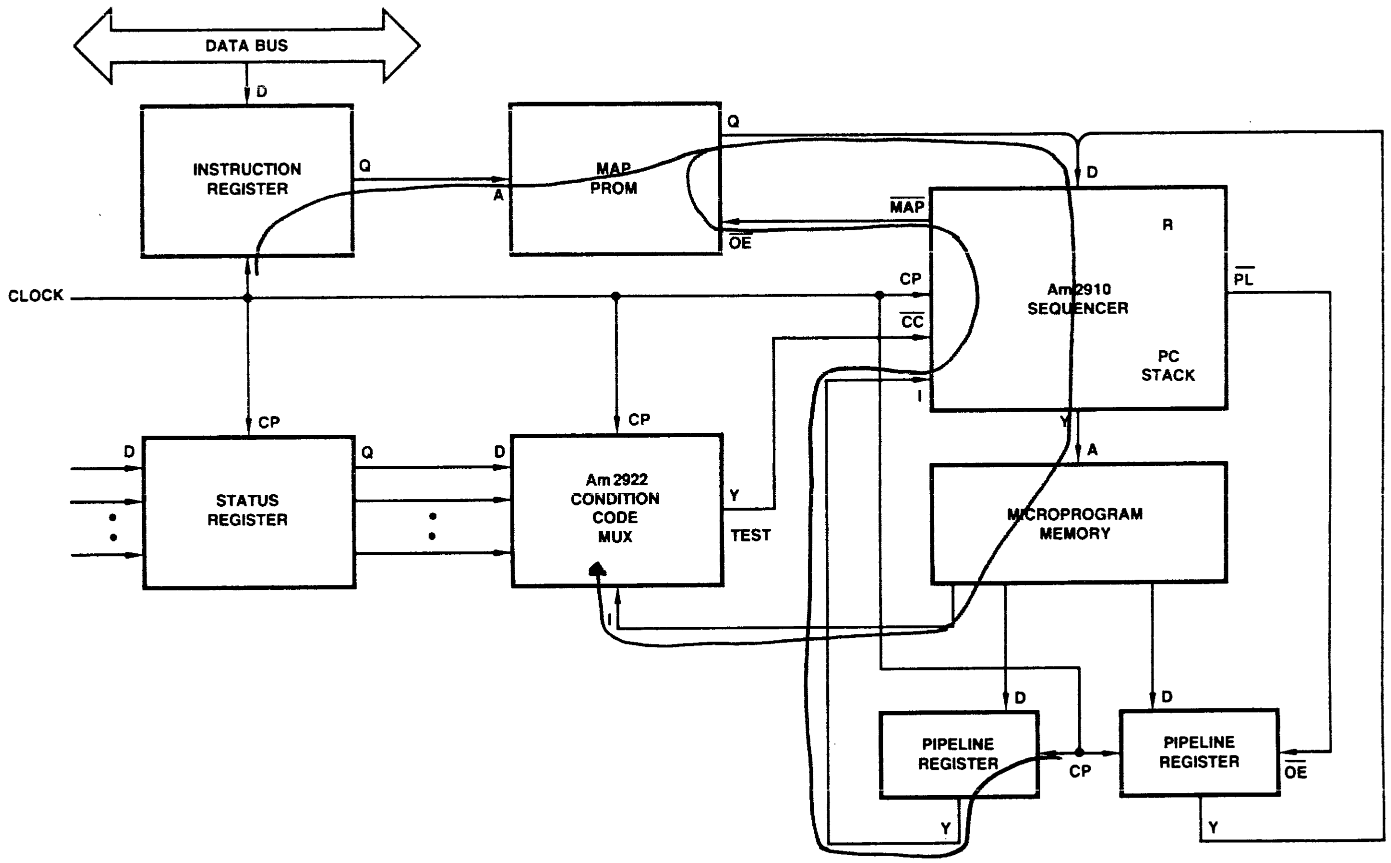
DEVICE	DEVICE PATH	PATH 1	PATH 2	PATH3	PATH 4	PATH 5
PIPELINE	CP → Y	15	15	15	---	---
2910	I → Y	70	70	---	---	---
2910	I → PC SETUP	---	---	104	---	---
2910	CP → Y	---	---	---	55	55
MEMORY	ADDR OUT	50	50	---	50	50
2922	SET-UP	11	---	---	11	---
PIPELINE	SET-UP	---	5	---	---	5
TOTAL ns		146	140	119	116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

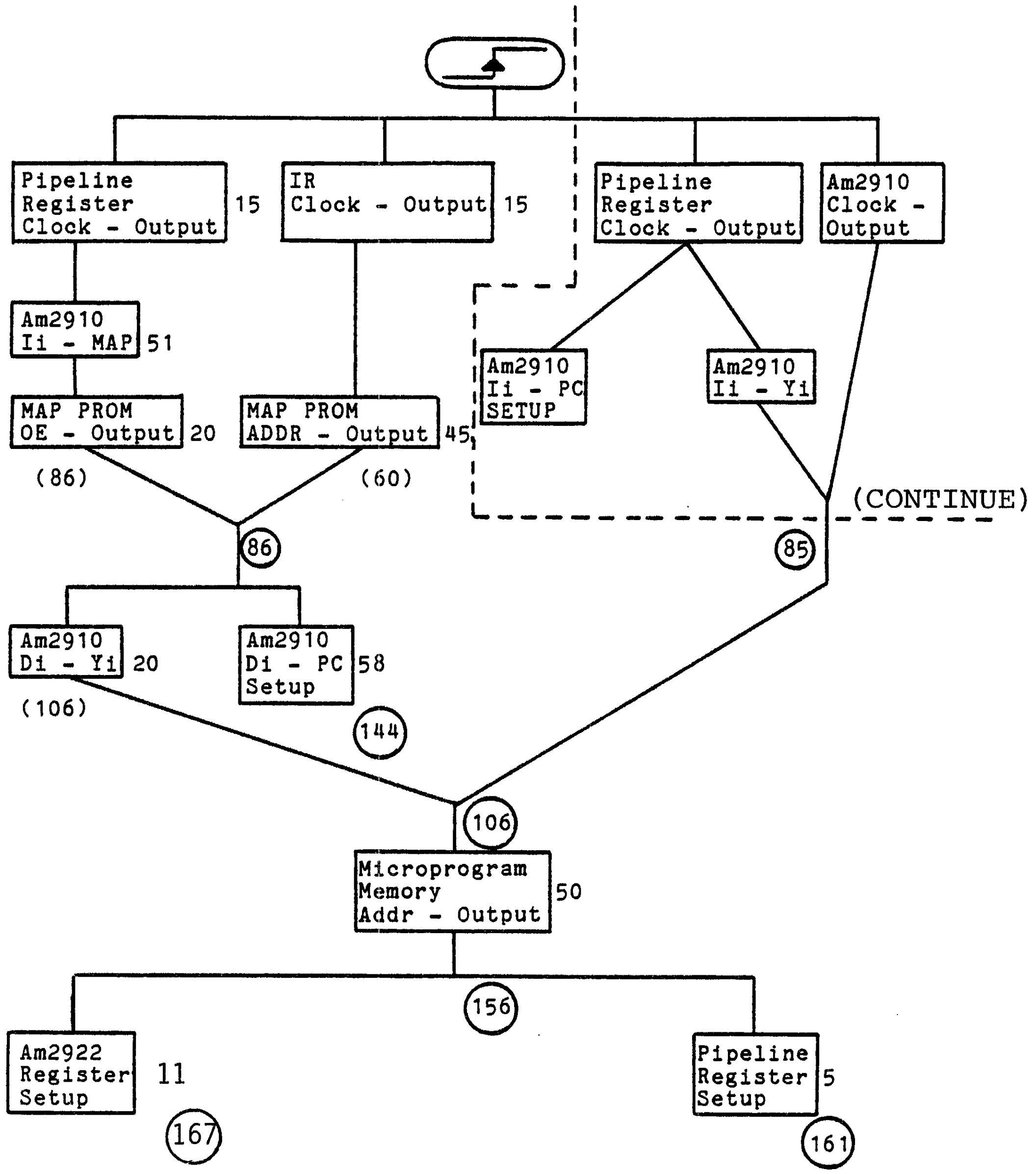


CONTINUE INSTRUCTION

PERT CHART

JUMP MAP

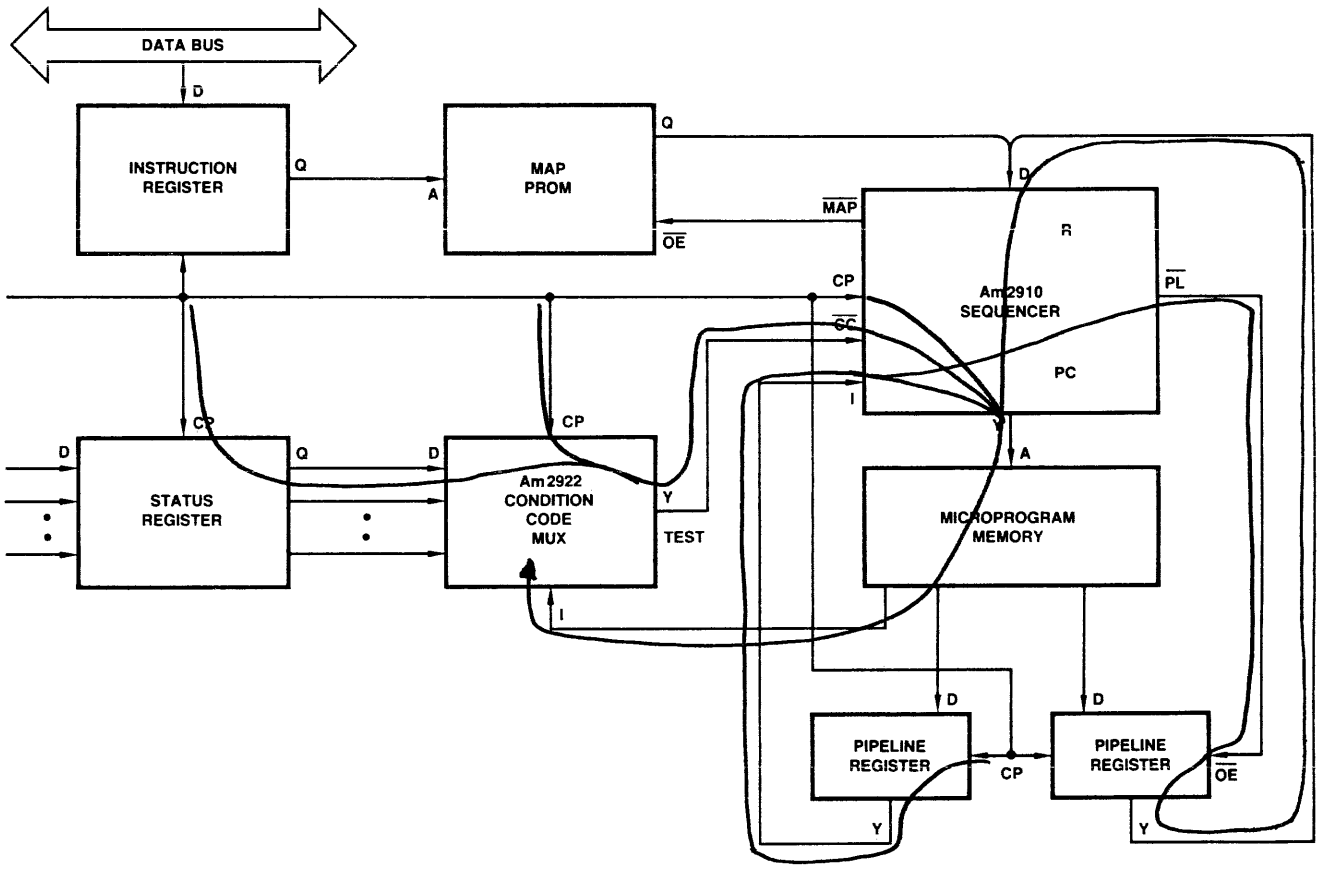

DATA BUS
D
INSTRUCTION REGISTER
Q
A
MAP PROM
Q
D
MAP
OE
CLOCK
CP
CC
I
Am2910 SEQUENCER
R
PC STACK
PL
CP
CP
D
Q
STATUS REGISTER
D
Am2922 CONDITION CODE MUX
Y
TEST
I
Y
A
MICROPROGRAM MEMORY
D
D
PIPELINE REGISTER
CP
PIPELINE REGISTER
OE
Y
Y

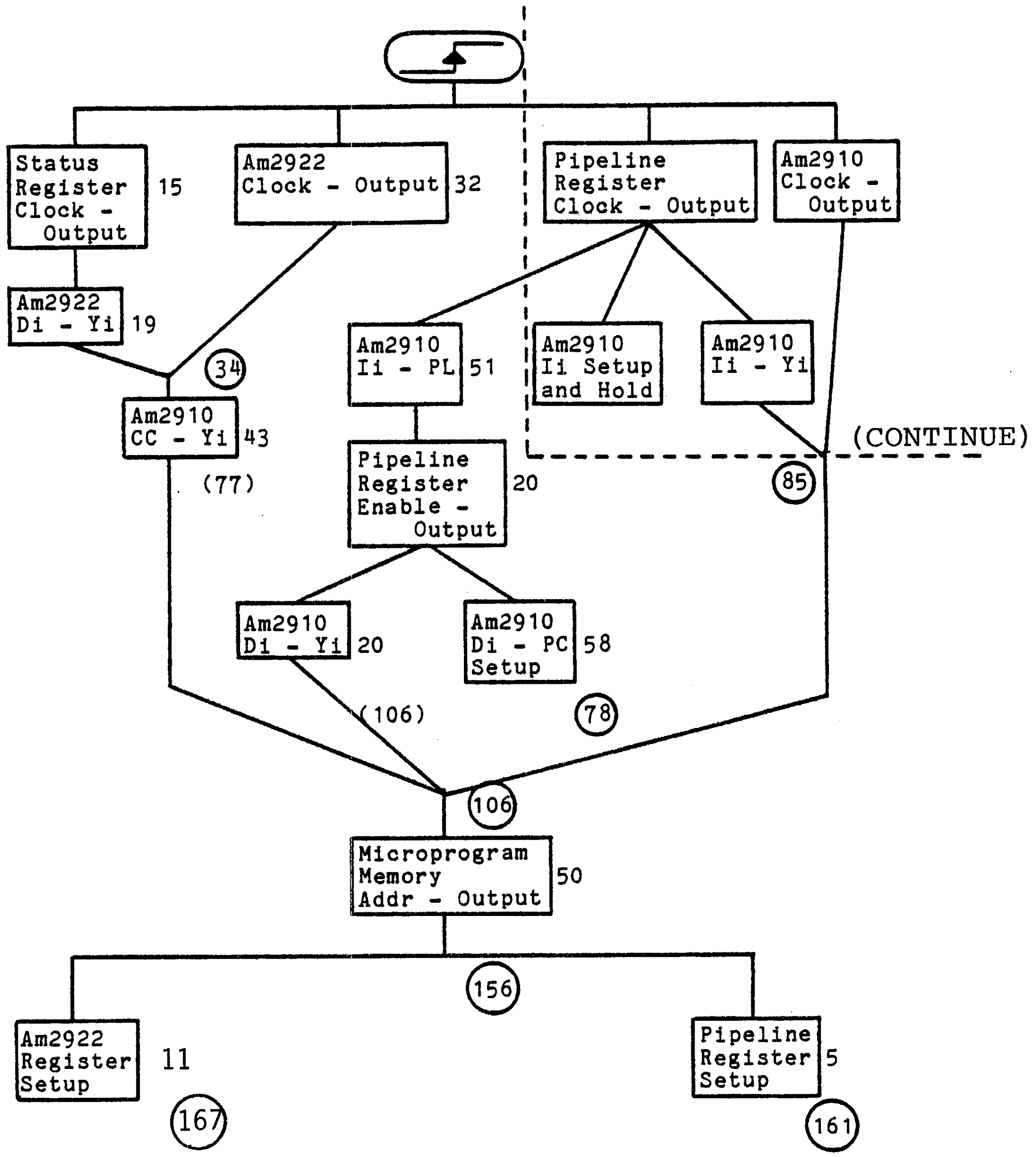


JUMP MAP

PERT CHART

CONDITIONAL JUMP - TAKEN





CONDITIONAL JUMP - TAKEN

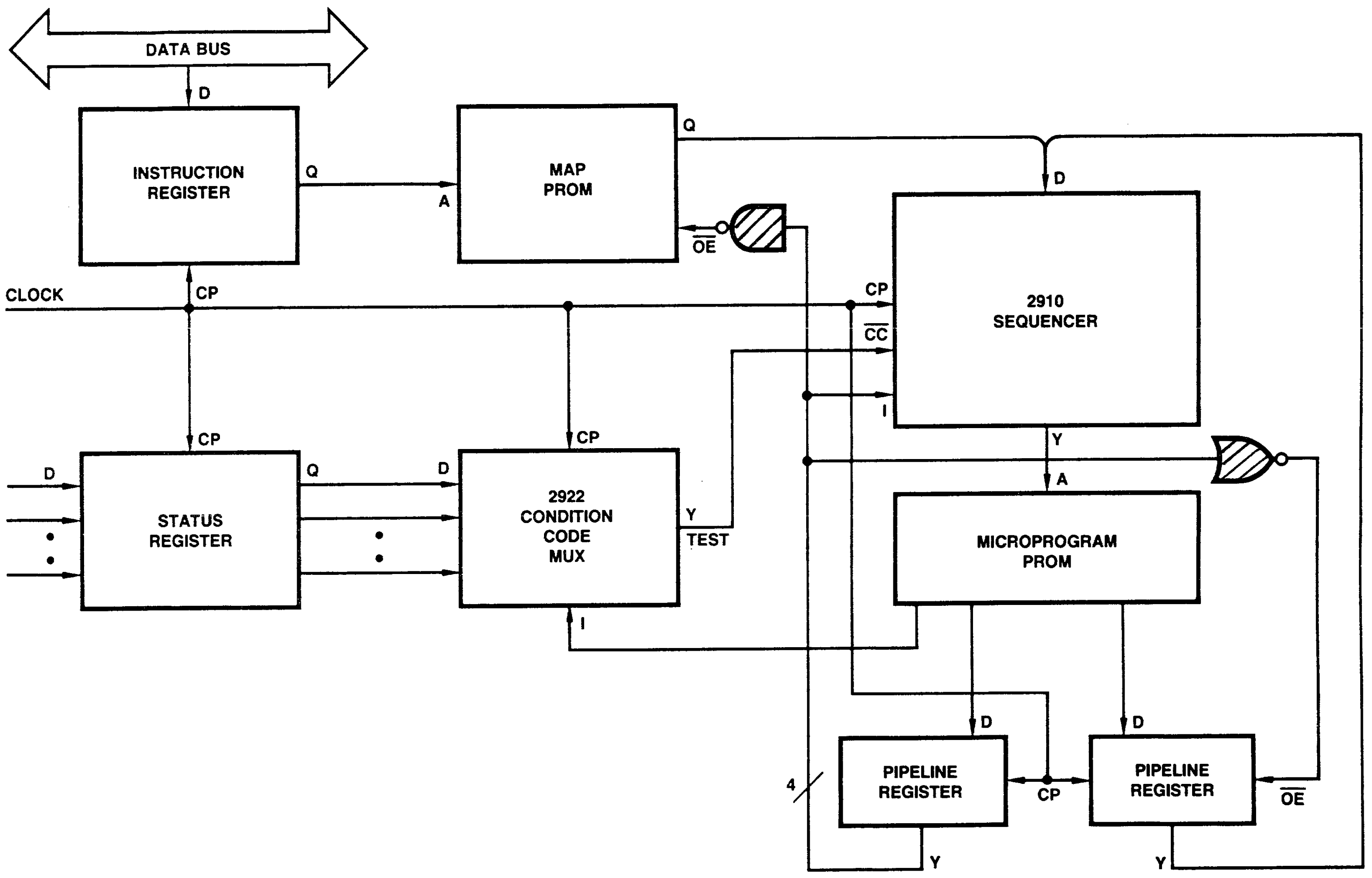
PERT CHART

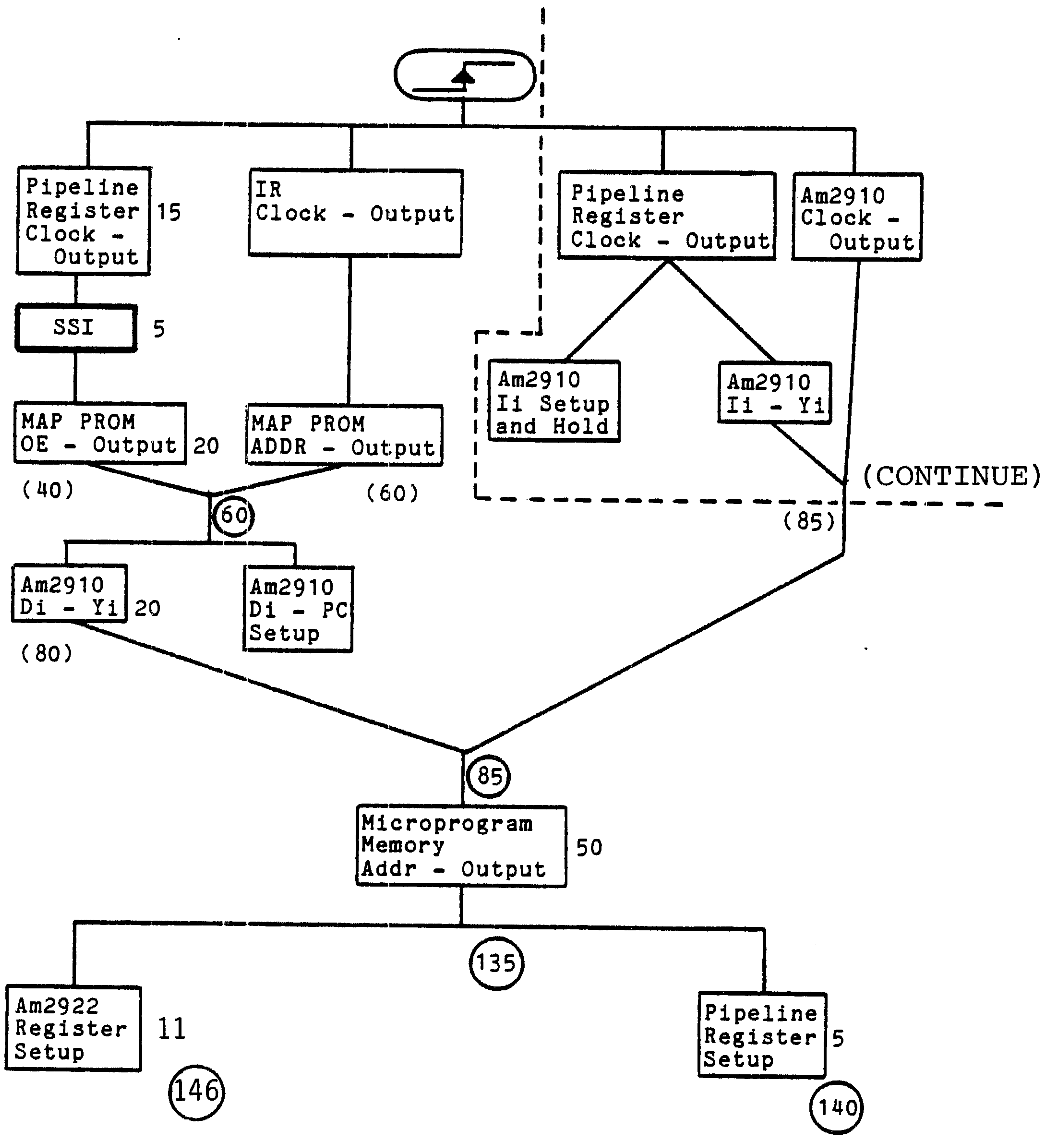
SPEEDING UP THE MICROCYCLE

- 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.

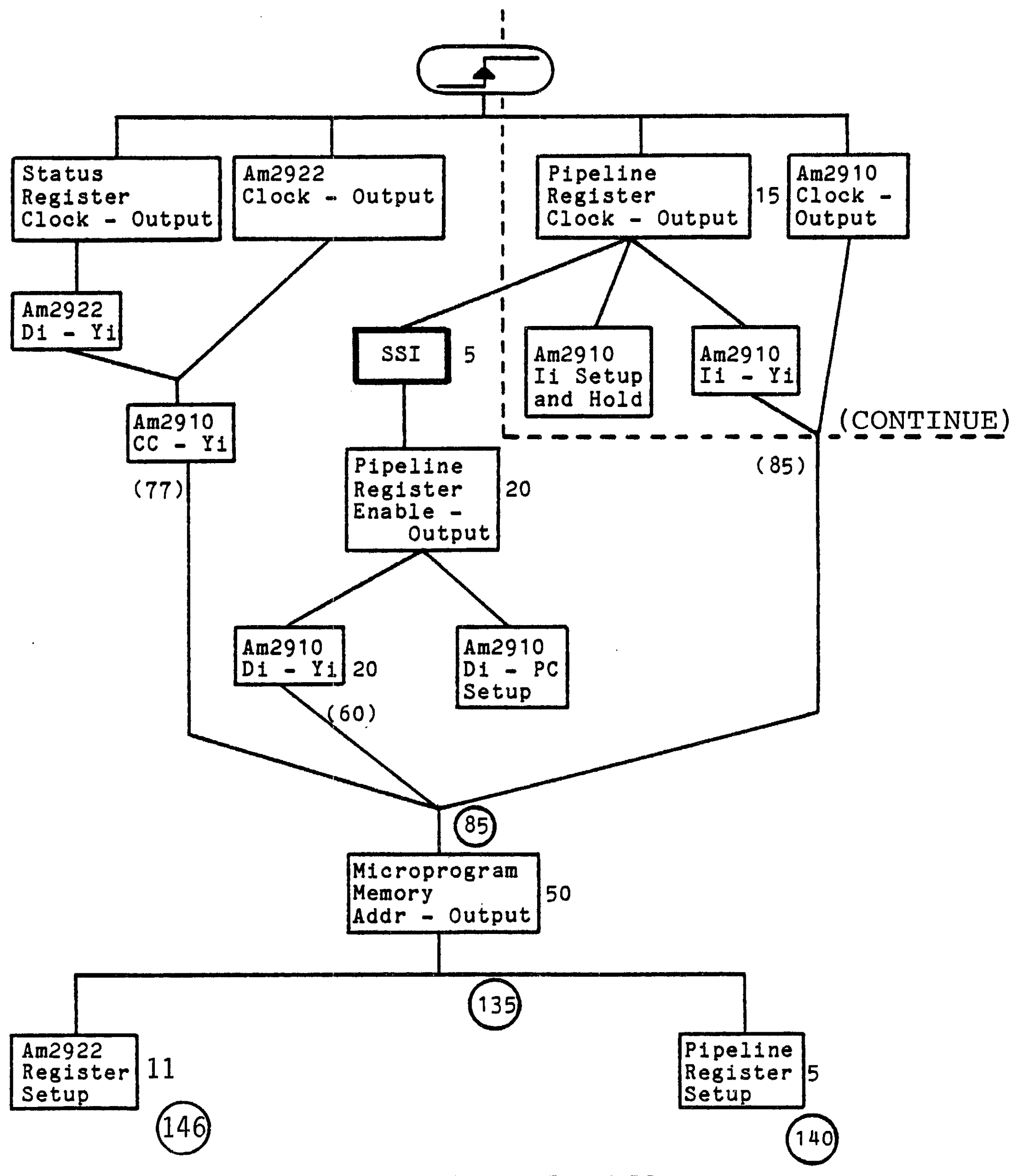
- Although the SSI delay is small, it too could be eliminated by driving the map and pipeline enables directly from the microword.


DATA BUS
D
INSTRUCTION REGISTER
Q
A
MAP PROM
Q
D
OE
CLOCK
CP
CP
2910 SEQUENCER
CC
I
Y
A
CP
STATUS REGISTER
D
Q
D
CP
2922 CONDITION CODE MUX
Y
TEST
I
MICROPROGRAM PROM
D
D
4
PIPELINE REGISTER
CP
PIPELINE REGISTER
OE
Y
Y



JUMP MAP - IMPROVED ARCHITECTURE

PERT CHART



JUMP TAKEN - IMPROVED ARCHITECTURE

PERT CHART

COFFEE MACHINE SOLUTION

(See ED2900A Excercise and Laboratory Manual)