COMPUTER ARCHITECTURE & ORGANIZATION

(6CS4-04)

Unit-2

Programming The Basic Computer

-Dr. Monica Lamba

Contents

- Introduction
- Machine Language
- Assembly Language
- Assembler
- Program loops
- Programming Arithmetic and logic operations
- Subroutines
- I-O Programming.
- Micro programmed Control: Control Memory
- Address sequencing
- Micro program Example, design of control Unit

Programming the Basic Computers

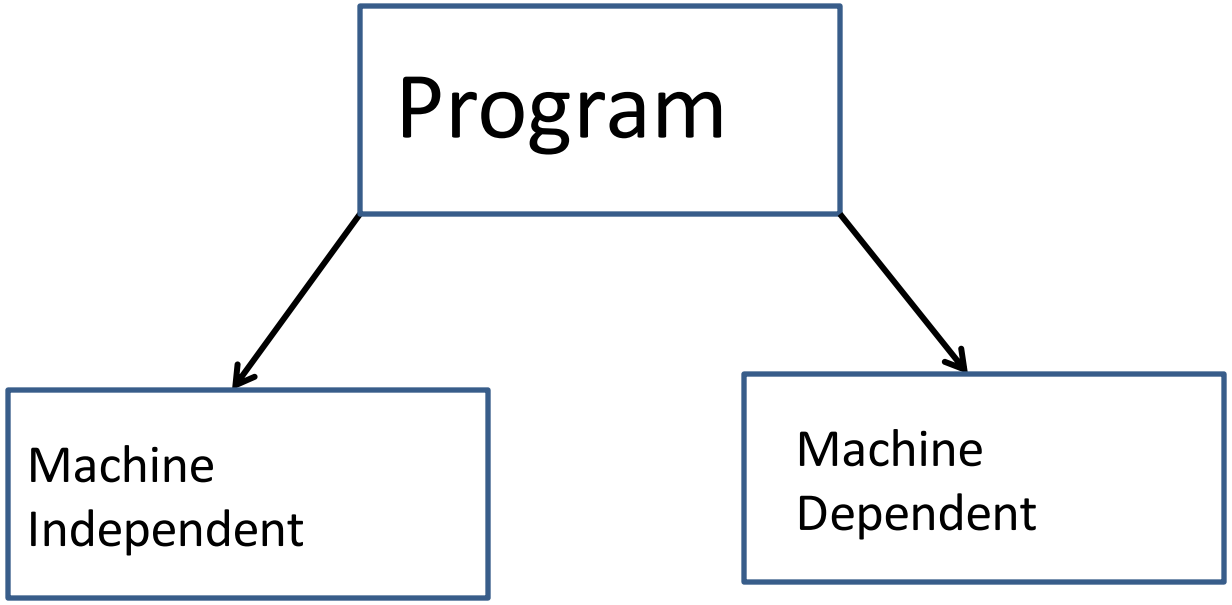
Programming the Basic Computer

Introduction

- A total computer system includes both *hardware* and *software*.
- Hardware consists of the physical components and all associated equipment.
- Software refers to the programs that are written for the computer.
- It is possible to be familiar with various aspects of computer software without being concerned with details of how the computer hardware operates. It is also possible to design parts of the hardware without a knowledge of its software capabilities.
- Engineers concerned with computer architecture should have a knowledge of both hardware and software because the two branches influence each other.

Programming the Basic Computer

- Writing a program for a computer consists of specifying, directly or indirectly, a sequence of machine instructions.
- Machine instructions inside the computer form a binary pattern which is difficult, if not impossible, for people to work with and understand.
- It is preferable to write programs with the more familiar symbols of the alphanumeric character set. As a consequence, there is a need for translating user-oriented symbolic programs into binary programs recognized by the hardware.


Program
Machine Independent
Machine Dependent

Instruction Set

- In basic Computer we have 25 instructions
- Out of which 7 are memory reference instructions(MRI)
- 18 are register reference instruction and input/output reference instructions.
- These 18 instructions are called non- memory reference instruction.

Computer Instructions

Symbol	Hexadecimal code	Description	
AND	0 or 8	AND M to AC	MRI
ADD	1 or 9	Add M to AC, carry to E	
LDA	2 or A	Load AC from M	
STA	3 or B	Store AC in M	
BUN	4 or C	Branch unconditionally to m	
BSA	5 or D	Save return address in m and branch to $m + 1$	
ISZ	6 or E	Increment M and skip if zero	
CLA	7800	Clear AC	RRI
CLE	7400	Clear E	
CMA	7200	Complement AC	
CME	7100	Complement E	
CIR	7080	Circulate right E and AC	
CIL	7040	Circulate left E and AC	
INC	7020	Increment AC	
SPA	7010	Skip if AC is positive	
SNA	7008	Skip if AC is negative	
SZA	7004	Skip if AC is zero	
SZE	7002	Skip if E is zero	
HLT	7001	Halt computer	
INP	F800	Input information and clear flag	I/O
OUT	F400	Output information and clear flag	
SKI	F200	Skip if input flag is on	
SKO	F100	Skip if output flag is on	
ION	F080	Turn interrupt on	
IOF	F040	Turn interrupt off	

Machine Language

- A program is a list of instructions or statements for directing the computer to perform a required data-processing task.
- The computer can execute programs only when they are represented internally in binary form.
  - for Programs written in any other language must be translated to the binary representation of instructions before they can be executed by the computer.
- Programs written for a computer may be in one of the following categories:
  1. Binary code.
  2. Octal or hexadecimal code.
  3. Symbolic code.
  4. High-level programming languages.

Machine Language

1. Binary code.
   This is a sequence of instructions and operands in binary that list the exact representation of instructions as they appear in computer memory.

Binary Program to Add Two Numbers

Location	Instruction code
0	0010 0000 0000 0100
1	0001 0000 0000 0101
10	0011 0000 0000 0110
11	0111 0000 0000 0001
100	0000 0000 0101 0011
101	1111 1111 1110 1001
110	0000 0000 0000 0000

- The first column gives the memory location (in binary) of each instruction or operand. The second column lists the binary content of these memory locations. (The *location* is the address of the memory word where the instruction is stored. It is important to differentiate it from the address part of the instruction itself.)

- The program can be stored in the indicated portion of memory, and then executed by the computer starting from address 0. The hardware of the computer will execute these instructions and perform the intended task. However, a person looking at this program will have a difficult time understanding what is to be achieved when this program is executed. Nevertheless, the computer hardware recognizes *only* this type of instruction code.

Machine Language

1. Binary code.
   This is a sequence of instructions and operands in binary that list the exact representation of instructions as they appear in computer memory.

Binary Program to Add Two Numbers

Location	Instruction code
0	0010 0000 0000 0100
1	0001 0000 0000 0101
10	0011 0000 0000 0110
11	0111 0000 0000 0001
100	0000 0000 0101 0011
101	1111 1111 1110 1001
110	0000 0000 0000 0000

2. Octal or hexadecimal code.

This is an equivalent translation of the binary code to octal or hexadecimal representation.

Binary Program to Add Two Numbers

Location	Instruction code
0	0010 0000 0000 0100
1	0001 0000 0000 0101
10	0011 0000 0000 0110
11	0111 0000 0000 0001
100	0000 0000 0101 0011
101	1111 1111 1110 1001
110	0000 0000 0000 0000

Hexadecimal Program to Add Two Numbers

Location	Instruction
000	2004
001	1005
002	3006
003	7001
004	0053
005	FFE9
006	0000

3. Symbolic code.

- The user employs symbols (letters, numerals, or special characters) for the operation part, the address part, and other parts of the instruction code. (Assembly Language)

- Each symbolic instruction can be translated into one binary coded instruction. This translation is done by a special program called an *assembler*.

Binary Program to Add Two Numbers

Location	Instruction code
0	0010 0000 0000 0100
1	0001 0000 0000 0101
10	0011 0000 0000 0110
11	0111 0000 0000 0001
100	0000 0000 0101 0011
101	1111 1111 1110 1001
110	0000 0000 0000 0000

Program with Symbolic Operation Codes

Location	Instruction	Comments
000	LDA 004	Load first operand into *AC*
001	ADD 005	Add second operand to *AC*
002	STA 006	Store sum in location 006
003	HLT	Halt computer
004	0053	First operand
005	FFE9	Second operand (negative)
006	0000	Store sum here

4. High-level programming languages. These are special languages developed to reflect the procedures used in the solution of a problem rather than be concerned with the computer hardware behavior.

- The program is written in a sequence of statements in a form that people prefer to think in when solving a problem.
- Each statement must be translated into a sequence of binary instructions before the program can be executed in a computer. The program that translates a high-level language program to binary is called a compiler.

Fortran Program to Add Two Numbers

```
INTEGER A, B, C
DATA A, 83      B, -23
C = A + B
END
```

Program with Symbolic Operation Codes

Location	Instruction	Comments
000	LDA 004	Load first operand into *AC*
001	ADD 005	Add second operand to *AC*
002	STA 006	Store sum in location 006
003	HLT	Halt computer
004	0053	First operand
005	FFE9	Second operand (negative)
006	0000	Store sum here

We can go one step further and replace each hexadecimal address by a symbolic address and each hexadecimal operand by a decimal operand.

This is convenient because one usually does not know exactly the numeric memory location of operands while writing a program.

Assembly Language Program to Add Two Numbers

```
        ORG 0       /Origin of program is location 0
        LDA A       /Load operand from location A
        ADD B       /Add operand from location B
        STA C       /Store sum in location C
        HLT         /Halt computer
A,      DEC 83      /Decimal operand
B,      DEC -23     /Decimal operand
C,      DEC 0       /Sum stored in location C
        END         /End of symbolic program
```

Assembly Language Program to Add Two Numbers

```
        ORG 0       /Origin of program is location 0
        LDA A       /Load operand from location A
        ADD B       /Add operand from location B
        STA C       /Store sum in location C
        HLT         /Halt computer
A,      DEC 83      /Decimal operand
B,      DEC -23     /Decimal operand
C,      DEC 0       /Sum stored in location C
        END         /End of symbolic program
```

Rules of the Language

- Each line of an assembly language program is arranged in three columns called fields. The fields specify the following information.

  1. The *label* field may be empty or it may specify a symbolic address.
  2. The *instruction* field specifies a machine instruction or a pseudoinstruction.
  3. The *comment* field may be empty or it may include a comment.

label opc comment

↑ ADD /.

Symbolic STA

address END

pseudoinstruction

- A pseudoinstruction is not a machine instruction but rather an instruction to the assembler giving information about some phase of the translation.

Symbol	Information for the Assembler
ORG N	Hexadecimal number N is the memory location for the instruction or operand listed in the following line
END	Denotes the end of symbolic program
DEC N	Signed decimal number N to be converted to binary
HEX N	Hexadecimal number N to be converted to binary

- The ORG (origin) pseudoinstruction informs the assembler that the instruction or operand in the following line is to be placed in a memory location specified by the number next to ORG.
- The END symbol is placed at the end of the program to inform the assembler that the program is terminated.
- The other two pseudoinstructions specify the radix of the operand and tell the assembler how to convert the listed number to a binary number.

Example

Assembly Language Program to Subtract Two Numbers

```
        ORG 100       /Origin of program is location 100
        LDA SUB       /Load subtrahend to AC
        CMA           /Complement AC
        INC           /Increment AC
        ADD MIN       /Add minuend to AC
        STA DIF       /Store difference
        HLT           /Halt computer
MIN,    DEC 83        /Minuend
SUB,    DEC -23       /Subtrahend
DIF,    HEX 0         /Difference stored here
        END           /End of symbolic program
```

6-7 Subroutines

Frequently, the same piece of code must be written over again in many different parts of a program. Instead of repeating the code every time it is needed, there is an obvious advantage if the common instructions are written only once. A set of common instructions that can be used in a program many times is called a *subroutine*. Each time that a subroutine is used in the main part of the program, a branch is executed to the beginning of the subroutine. After the subroutine has been executed, a branch is made back to the main program.

A subroutine consists of a self-contained sequence of instructions that carries out a given task. A branch can be made to the subroutine from any part of the main program. This poses the problem of how the subroutine knows which location to return to, since many different locations in the main program may make branches to the same subroutine. It is therefore necessary to store the return address somewhere in the computer for the subroutine to know where to return. Because branching to a subroutine and returning to the main program is such a common operation, all computers provide special instructions to facilitate subroutine entry and return.

In the basic computer, the link between the main program and a subroutine is the BSA instruction (branch and save return address). To explain how this instruction is used, let us write a subroutine that shifts the content of the accumulator four times to the left. Shifting a word four times is a useful operation for processing binary-coded decimal numbers or alphanumeric characters. Such an operation could have been included as a machine instruction in the computer. Since it is not included, a subroutine is formed to accomplish this task. The program of Table 6-16 starts by loading the value of X into the

TABLE 6-16 Program to Demonstrate the Use of Subroutines

Location			
		ORG 100	/Main program
100		LDA X	/Load X
101		BSA SH4	/Branch to subroutine
102		STA X	/Store shifted number
103		LDA Y	/Load Y
104		BSA SH4	/Branch to subroutine again
105		STA Y	/Store shifted number
106		HLT	
107	X,	HEX 1234	
108	Y,	HEX 4321	
			/Subroutine to shift left 4 times
109	SH4,	HEX 0	/Store return address here
10A		CIL	/Circulate left once
10B		CIL	
10C		CIL	
10D		CIL	/Circulate left fourth time
10E		AND MSK	/Set *AC*(13–16) to zero
10F		BUN SH4 I	/Return to main program
110	MSK,	HEX FFF0	/Mask operand
		END	

AC. The next instruction encountered is BSA SH4. The BSA instruction is in location 101. Subroutine SH4 must return to location 102 after it finishes its task. When the BSA instruction is executed, the control unit stores the return address 102 into the location defined by the symbolic address SH4 (which is 109). It also transfers the value of SH4 + 1 into the program counter. After this instruction is executed, memory location 109 contains the binary equivalent of hexadecimal 102 and the program counter contains the binary equivalent of hexadecimal 10A. This action has saved the return address and the subroutine is now executed starting from location 10A (since this is the content of *PC* in the next fetch cycle).

The computation in the subroutine circulates the content of *AC* four times to the left. In order to accomplish a logical shift operation, the four low-order bits must be set to zero. This is done by masking FFF0 with the content of *AC*. A mask operation is a logic AND operation that clears the bits of the *AC* where the mask operand is zero and leaves the bits of the *AC* unchanged where the mask operand bits are 1's.

The last instruction in the subroutine returns the computer to the main program. This is accomplished by the indirect branch instruction with an address symbol identical to the symbol used for the subroutine name. The address to which the computer branches is not SH4 but the value found in

location SH4 because this is an indirect address instruction. What is found in location SH4 is the return address 102 which was previously stored there by the BSA instruction. The computer returns to execute the instruction in location 102. The main program continues by storing the shifted number into location X. A new number is then loaded into the *AC* from location Y, and another branch is made to the subroutine. This time location SH4 will contain the return address 105 since this is now the location of the next instruction after BSA. The new operand is shifted and the subroutine returns to the main program at location 105.

From this example we see that the first memory location of each subroutine serves as a link between the main program and the subroutine. The procedure for branching to a subroutine and returning to the main program is referred to as a subroutine *linkage*. The BSA instruction performs an operation commonly called subroutine *call*. The last instruction of the subroutine performs an operation commonly called subroutine *return*.

The procedure used in the basic computer for subroutine linkage is commonly found in computers with only one processor register. Many computers have multiple processor registers and some of them are assigned the name *index registers*. In such computers, an index register is usually employed to implement the subroutine linkage. A branch-to-subroutine instruction stores the return address in an index register. A return-from-subroutine instruction is effected by branching to the address presently stored in the index register.

Subroutine Parameters and Data Linkage

When a subroutine is called, the main program must transfer the data it wishes the subroutine to work with. In the previous example, the data were transferred through the accumulator. The operand was loaded into the *AC* prior to the branch. The subroutine shifted the number and left it there to be accepted by the main program. In general, it is necessary for the subroutine to have access to data from the calling program and to return results to that program. The accumulator can be used for a single input parameter and a single output parameter. In computers with multiple processor registers, more parameters can be transferred this way. Another way to transfer data to a subroutine is through the memory. Data are often placed in memory locations following the call. They can also be placed in a block of storage. The first address of the block is then placed in the memory location following the call. In any case, the return address always gives the link information for transferring data between the main program and the subroutine.

As an illustration, consider a subroutine that performs the logic OR operation. Two operands must be transferred to the subroutine and the subroutine must return the result of the operation. The accumulator can be used

to transfer one operand and to receive the result. The other operand is inserted in the location following the BSA instruction. This is demonstrated in the program of Table 6-17. The first operand in location X is loaded into the *AC*. The second operand is stored in location 202 following the BSA instruction. After the branch, the first location in the subroutine holds the number 202. Note that in this case, 202 is not the return address but the address of the second operand. The subroutine starts performing the OR operation by complementing the first operand in the *AC* and storing it in a temporary location TMP. The second operand is loaded into the *AC* by an indirect instruction at location OR. Remember that location OR contains the number 202. When the instruction refers to it indirectly, the operand at location 202 is loaded into the *AC*. This operand is complemented and then ANDed with the operand stored in TMP. Complementing the result forms the OR operation.

The return from the subroutine must be manipulated so that the main program continues from location 203 where the next instruction is located. This is accomplished by incrementing location OR with the ISZ instruction. Now location OR holds the number 203 and an indirect BUN instruction causes a return to the proper place.

It is possible to have more than one operand following the BSA instruc-

TABLE 6-17 Program to Demonstrate Parameter Linkage

Location			
		ORG 200	
200		LDA X	/Load first operand into *AC*
201		BSA OR	/Branch to subroutine OR
202		HEX 3AF6	/Second operand stored here
203		STA Y	/Subroutine returns here
204		HLT	
205	X,	HEX 7B95	/First operand stored here
206	Y,	HEX 0	/Result stored here
207	OR,	HEX 0	/Subroutine OR
208		CMA	/Complement first operand
209		STA TMP	/Store in temporary location
20A		LDA OR I	/Load second operand
20B		CMA	/Complement second operand
20C		AND TMP	/AND complemented first operand
20D		CMA	/Complement again to get OR
20E		ISZ OR	/Increment return address
20F		BUN OR I	/Return to main program
210	TMP,	HEX 0	/Temporary storage
		END	

tion. The subroutine must increment the return address stored in its first location for each operand that it extracts from the calling program. Moreover, the calling program can reserve one or more locations for the subroutine to return results that are computed. The first location in the subroutine must be incremented for these locations as well, before the return. If there is a large amount of data to be transferred, the data can be placed in a block of storage and the address of the first item in the block is then used as the linking parameter.

A subroutine that moves a block of data starting at address 100 into a block starting with address 200 is listed in Table 6-18. The length of the block is 16 words. The first introduction is a branch to subroutine MVE. The first part of the subroutine transfers the three parameters 100, 200 and −16 from the main program and places them in its own storage location. The items are retrieved from their blocks by the use of two pointers. The counter ensures that only 16 items are moved. When the subroutine completes its operation, the data required is in the block starting from the location 200. The return to the main program is to the HLT instruction.

TABLE 6-18 Subroutine to Move a Block of Data

```
                        /Main program
        BSA MVE         /Branch to subroutine
        HEX 100         /First address of source data
        HEX 200         /First address of destination data
        DEC -16         /Number of items to move
        HLT
MVE,    HEX 0           /Subroutine MVE
        LDA MVE I       /Bring address of source
        STA PT1         /Store in first pointer
        ISZ MVE         /Increment return address
        LDA MVE I       /Bring address of destination
        STA PT2         /Store in second pointer
        ISZ MVE         /Increment return address
        LDA MVE I       /Bring number of items
        STA CTR         /Store in counter
        ISZ MVE         /Increment return address
LOP,    LDA PT1 I       /Load source item
        STA PT2 I       /Store in destination
        ISZ PT1         /Increment source pointer
        ISZ PT2         /Increment destination pointer
        ISZ CTR         /Increment counter
        BUN LOP         /Repeat 16 times
        BUN MVE I       /Return to main program
PT1,    —
PT2,    —
CTR,    —
```

6-8 Input–Output Programming

Users of the computer write programs with symbols that are defined by the programming language employed. The symbols are strings of characters and each character is assigned an 8-bit code so that it can be stored in computer memory. A binary-coded character enters the computer when an INP (input) instruction is executed. A binary-coded character is transferred to the output device when an OUT (output) instruction is executed. The output device detects the binary code and types the corresponding character.

Table 6-19(a) lists the instructions needed to input a character and store it in memory. The SKI instruction checks the input flag to see if a character is available for transfer. The next instruction is skipped if the input flag bit is 1. The INP instruction transfers the binary-coded character into *AC*(0–7). The character is then printed by means of the OUT instruction. A terminal unit that communicates directly with a computer does not print the character when a key is depressed. To type it, it is necessary to send an OUT instruction for the printer. In this way, the user is ensured that the correct transfer has occurred. If the SKI instruction finds the flag bit at 0, the next instruction in sequence is executed. This instruction is a branch to return and check the flag bit again. Because the input device is much slower than the computer, the two instructions in the loop will be executed many times before a character is transferred into the accumulator.

Table 6-19(b) lists the instructions needed to print a character initially stored in memory. The character is first loaded into the *AC*. The output flag is then checked. If it is 0, the computer remains in a two-instruction loop checking the flag bit. When the flag changes to 1, the character is transferred from the accumulator to the printer.

TABLE 6-19 Programs to Input and Output One Character

(a) Input a character:

```
CIF,    SKI       /Check input flag
        BUN CIF   /Flag=0, branch to check again
        INP       /Flag=1, input character
        OUT       /Print character
        STA CHR   /Store character
        HLT
CHR,    —         /Store character here
```

(b) Output one character:

```
        LDA CHR   /Load character into AC
COF,    SKO       /Check output flag
        BUN COF   /Flag=0, branch to check again
        OUT       /Flag=1, output character
        HLT
CHR,    HEX 0057  /Character is "W"
```

Character Manipulation

A computer is not just a calculator but also a symbol manipulator. The binary-coded characters that represent symbols can be manipulated by computer instructions to achieve various data-processing tasks. One such task may be to pack two characters in one word. This is convenient because each character occupies 8 bits and a memory word contains 16 bits. The program in Table 6-20 lists a subroutine named IN2 that inputs two characters and packs them into one 16-bit word. The packed word remains in the accumulator. Note that subroutine SH4 (Table 6-16) is called twice to shift the accumulator left eight times.

In the discussion of the assembler it was assumed that the symbolic program is stored in a section of memory which is sometimes called a *buffer*. The symbolic program being typed enters through the input device and is stored in consecutive memory locations in the buffer. The program listed in Table 6-21 can be used to input a symbolic program from the keyboard, pack two characters in one word, and store them in the buffer. The first address of the buffer is 500. The first double character is stored in location 500 and all characters are stored in sequential locations. The program uses a pointer for keeping track of the current empty location in the buffer. No counter is used in the program, so characters will be read as long as they are available or until the buffer reaches location 0 (after location FFFF). In a practical situation it may be necessary to limit the size of the buffer and a counter may be used for this purpose. Note that subroutine IN2 of Table 6-20 is called to input and pack the two characters.

In discussing the second pass of the assembler in Sec. 6-4 it was mentioned that one of the most common operations of an assembler is table lookup. This is an operation that searches a table to find out if it contains a given symbol. The search may be done by comparing the given symbol with each of the symbols stored in the table. The search terminates when a match occurs

TABLE 6-20 Subroutine to Input and Pack Two Characters

```
IN2,    —           /Subroutine entry
FST,    SKI
        BUN FST
        INP         /Input first character
        OUT
        BSA SH4     /Shift left four times
        BSA SH4     /Shift left four more times
SCD,    SKI
        BUN SCD
        INP         /Input second character
        OUT
        BUN IN2 I   /Return
```

TABLE 6-21 Program to Store Input Characters in a Buffer

```
        LDA ADS     /Load first address of buffer
        STA PTR     /Initialize pointer
LOP,    BSA IN2     /Go to subroutine IN2 (Table 6-20)
        STA PTR I   /Store double character word in buffer
        ISZ PTR     /Increment pointer
        BUN LOP     /Branch to input more characters
        HLT
ADS,    HEX 500     /First address of buffer
PTR,    HEX 0       /Location for pointer
```

or if none of the symbols match. When a match occurs, the assembler retrieves the equivalent binary value. A program for comparing two words is listed in Table 6-22. The comparison is accomplished by forming the 2's complement of a word (as if it were a number) and arithmetically adding it to the second word. If the result is zero, the two words are equal and a match occurs. If the result is not zero, the words are not the same. This program can serve as a subroutine in a table-lookup program.

Program Interrupt

The running time of input and output programs is made up primarily of the time spent by the computer in waiting for the external device to set its flag. The waiting loop that checks the flag keeps the computer occupied with a task that wastes a large amount of time. This waiting time can be eliminated if the interrupt facility is used to notify the computer when a flag is set. The advantage of using the interrupt is that the information transfer is initiated upon request from the external device. In the meantime, the computer can be busy performing other useful tasks. Obviously, if no other program resides in memory, there is nothing for the computer to do, so it might as well check for

TABLE 6-22 Program to Compare Two Words

```
        LDA WD1     /Load first word
        CMA
        INC         /Form 2's complement
        ADD WD2     /Add second word
        SZA         /Skip if AC is zero
        BUN UEQ     /Branch to "unequal" routine
        BUN EQL     /Branch to "equal" routine
WD1,    —
WD2,    —
```

the flags. The interrupt facility is useful in a multiprogram environment when two or more programs reside in memory at the same time.

Only one program can be executed at any given time even though two or more programs may reside in memory. The program currently being executed is referred to as the running program. The other programs are usually waiting for input or output data. The function of the interrupt facility is to take care of the data transfer of one (or more) program while another program is currently being executed. The running program must include an ION instruction to turn the interrupt on. If the interrupt facility is not used, the program must include an IOF instruction to turn it off. (The *start* switch of the computer should also turn the interrupt off.)

The interrupt facility allows the running program to proceed until the input or output device sets its ready flag. Whenever a flag is set to 1, the computer completes the execution of the instruction in progress and then acknowledges the interrupt. The result of this action is that the return address is stored in location 0. The instruction in location 1 is then performed; this initiates a service routine for the input or output transfer. The service routine can be stored anywhere in memory provided a branch to the start of the routine is stored in location 1. The service routine must have instructions to perform the following tasks:

1. Save contents of processor registers.
2. Check which flag is set.
3. Service the device whose flag is set.
4. Restore contents of processor registers.
5. Turn the interrupt facility on.
6. Return to the running program.

The contents of processor registers before the interrupt and after the return to the running program must be the same; otherwise, the running program may be in error. Since the service routine may use these registers, it is necessary to save their contents at the beginning of the routine and restore them at the end. The sequence by which the flags are checked dictates the priority assigned to each device. Even though two or more flags may be set at the same time, the devices nevertheless are serviced one at a time. The device with higher priority is serviced first followed by the one with lower priority.

The occurrence of an interrupt disables the facility from further interrupts. The service routine must turn the interrupt on before the return to the running program. This will enable further interrupts while the computer is executing the running program. The interrupt facility should not be turned on until after the return address is inserted into the program counter.

An example of a program that services an interrupt is listed in Table 6-23.

TABLE 6-23 Program to Service an Interrupt

Location			
0	ZRO,	—	/Return address stored here
1		BUN SRV	/Branch to service routine
100		CLA	/Portion of running program
101		ION	/Turn on interrupt facility
102		LDA X	
103		ADD Y	/Interrupt occurs here
104		STA Z	/Program returns here after interrupt
•		•	
•		•	
•			/Interrupt service routine
200	SRV,	STA SAC	/Store content of *AC*
		CIR	/Move *E* into *AC*(1)
		STA SE	/Store content of *E*
		SKI	/Check input flag
		BUN NXT	/Flag is off, check next flag
		INP	/Flag is on, input character
		OUT	/Print character
		STA PT1 I	/Store it in input buffer
		ISZ PT1	/Increment input pointer
	NXT,	SKO	/Check output flag
		BUN EXT	/Flag is off, exit
		LDA PT2 I	/Load character from output buffer
		OUT	/Output character
		ISZ PT2	/Increment output pointer
	EXT,	LDA SE	/Restore value of *AC*(1)
		CIL	/Shift it to *E*
		LDA SAC	/Restore content of *AC*
		ION	/Turn interrupt on
		BUN ZRO I	/Return to running program
	SAC,	—	/*AC* is stored here
	SE,	—	/*E* is stored here
	PT1,	—	/Pointer of input buffer
	PT2,	—	/Pointer of output buffer

Chap. 7 Microprogrammed Control *(Control Unit)*

7-1 Control Memory

Control Unit

- Initiate sequences of microoperations
  - » Control signal *(that specify microoperations)* in a bus-organized system by the groups of bits that select the paths in multiplexers, decoders, and arithmetic logic units
- Two major types of Control Unit
  - » Hardwired Control :
    - The control logic is implemented with gates, F/Fs, decoders, and other digital circuits
    - + Fast operation, - Wiring change(if the design has to be modified) is difficult.
  - » Microprogrammed Control :
    - The control information is stored in a control memory, and the control memory is programmed to initiate the required sequence of microoperations for an instruction
    - + Any required change can be done by updating the microprogram in control memory, - Slow operation

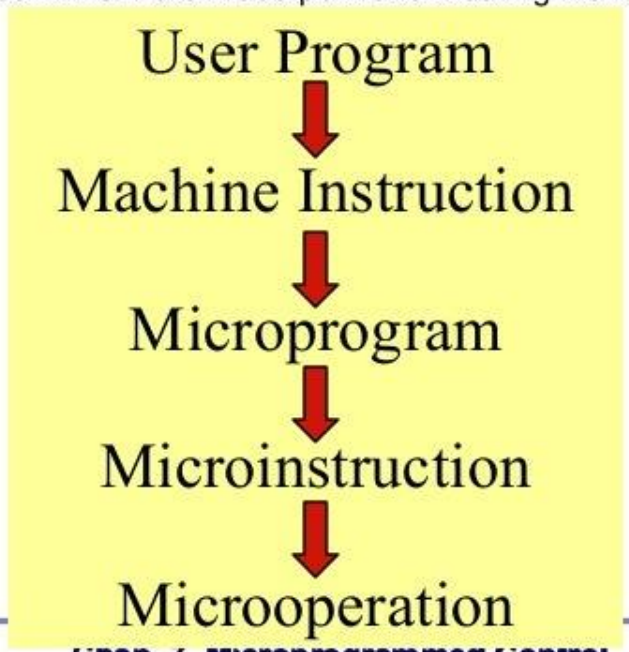
Control Word

- The control variables at any given time can be represented by a string of 1's and 0's is called control world

Microprogrammed Control Unit

- A control unit whose binary control variables are stored in memory (*control memory*).

- Microinstruction : (*Control Word in Control Memory)*
  - The instruction store in control memory is called microinstruction (specifies one or more microoperations)
- Microprogram
  - Microprogram is a sequence of microinstruction just like as program is a sequence of program. It is two type as follow:
    - » Dynamic microprogramming : (*Control Memory* = RAM)
      - RAM can be used for writing (*to change a writable control memory*)
      - Microprogram is loaded initially from an auxiliary memory such as a magnetic disk
    - » Static microprogramming : (*Control Memory* = ROM)
      - Control words in ROM are made permanent during the hardware production.



Micro programmed Control Organization

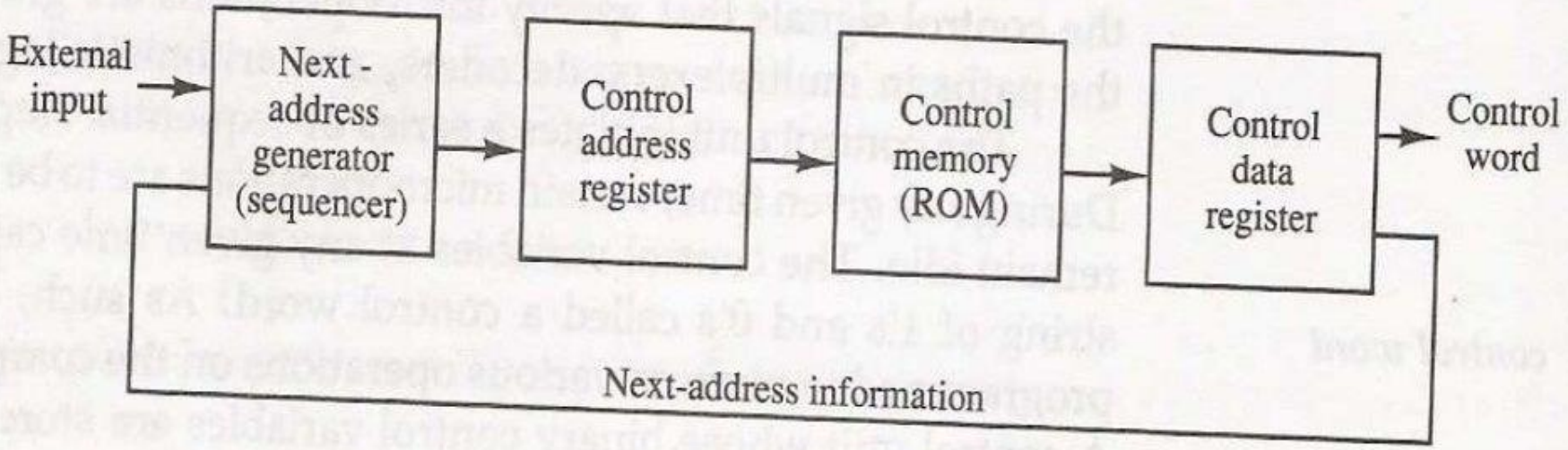
- Microprogrammed control Organization :(***Fig. 7-1)***
  - 1) Control Memory
    - » Computer Memory *employs a micro programmed control unit which have two separate memory*
      - Main Memory : for storing user program (*Machine instruction/data*)
      - Control Memory : for storing microprogram (*Microinstruction*)
  - 2) Control Address Register
    - » Specify the address of the microinstruction3) Sequencer (= *Next Address Generator*)
    - » Determine the address sequence that is read from control memory
    - » Next address of the next microinstruction can be specified several way depending on the sequencer input : *p. 217, [1, 2, 3, and 4]*

Block Diagram of Microprogarmmed Control Memory

- 4) Control Data Register (= *Pipeline Register*)
  - » Hold the microinstruction read from control memory
  - » Allows the execution of the microoperations specified by the control word ***simultaneously*** with the generation of the next microinstruction

- Example(RISC Architecture Concept)
  - RISC(Reduced Instruction Set Computer) system use hardwired control rather than microprogrammed control

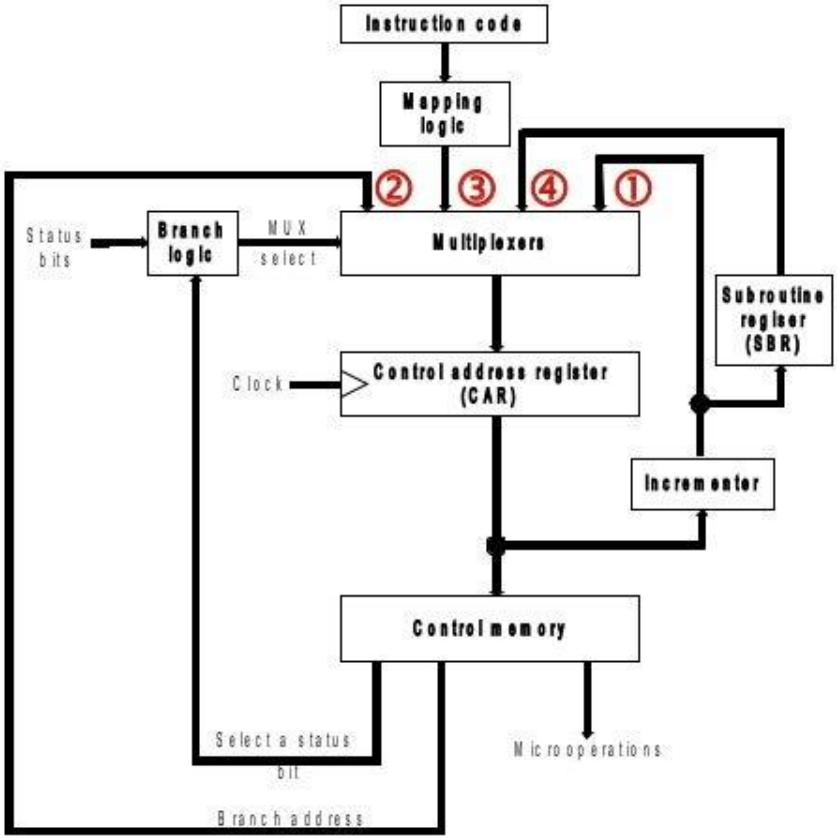
Figure 7-1 Microprogrammed control organization.



Address Sequencing

- 7-2 Address Sequencing
  - Address Sequencing = Sequencer : Next Address Generator
    - Selection of address for control memory
  - Routine → *Subroutine : program used by other ROUTINES*
    - Microinstruction are stored in control memory *in groups With each group specify a routine.*
    - *each computer instruction has it's own micro program routine in control memory to generate microinstructions to execute an instruction.*
  - Mapping : mapping of
    - Instruction Code → into Address in control memory*where routine is located* is called mapping process
  - **Process of Address Sequencing** :
    - 1) Incrementing of the control address register
    - 2) Unconditional branch or conditional branch, depending on status bit conditions
    - 3) Mapping process (*bits of the instruction address for control memory*)
    - 4) A facility for subroutine call and return

- Selection of address for control memory : ***Fig. 7-2***
  - Multiplexer
    - ① CAR Increment
    - ② JMP/CALL
    - ③ Mapping
    - ④ Subroutine Return
  - CAR : Control Address Register
    - » CAR receive the address from 4 different paths
      1) Incrementer
      2) Branch address from control memory
      3) Mapping Logic
      4) SBR : Subroutine Register
  - SBR : Subroutine Register
    - » Return Address can not be stored in ROM
    - » Return Address for a subroutine is stored in SBR

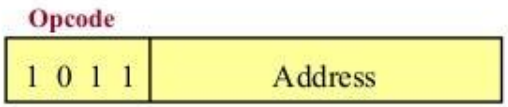


Conditional Branching

- Status Bits
  - » Control the conditional branch decisions generated in the ***Branch Logic***
- Branch Logic
  - » Test the specified condition and Branch to the indicated address if the condition is met ; otherwise, the control address register is just incremented.

Mapping of Instruction :

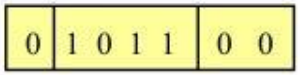
- Computer Instruction



Mapping bits

0 x x x x 0 0

Microinstruction Address



- 4 bit Opcode = specify up to 16 distinct instruction
- Mapping Process : Converts *the 4-bit Opcode* to *a 7-bit control memory address*
  - » 1) Place a “0” in the most significant bit of the address
  - » 2) Transfer 4-bit Operation code bits
  - » 3) Clear the two least significant bits of the CAR (*Microinstruction*)
- Mapping Function : Implemented by *Mapping ROM* or *PLD*
- Control Memory Size : 128 words (= 2^7)

- Subroutine
  - Subroutines are programs that are used by other routines
    - » Subroutine can be called from any point within the main body of the microprogram
  - Microinstructions can be saved by subroutines that use common section of microcode

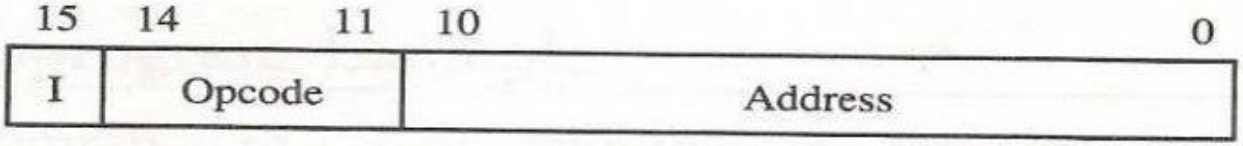
    Memory Reference ,Operands Effective Address

    Subroutine must have a provision for

    - » storing the return address during a subroutine call
    - » restoring the address during a subroutine return
      - Last-In First Out(LIFO) Register Stack

Instruction Format

- Instruction Format :
  - I : 1 bit for indirect addressing
  - Opcode : 4 bit operation code
  - Address : 11 bit address for system memory

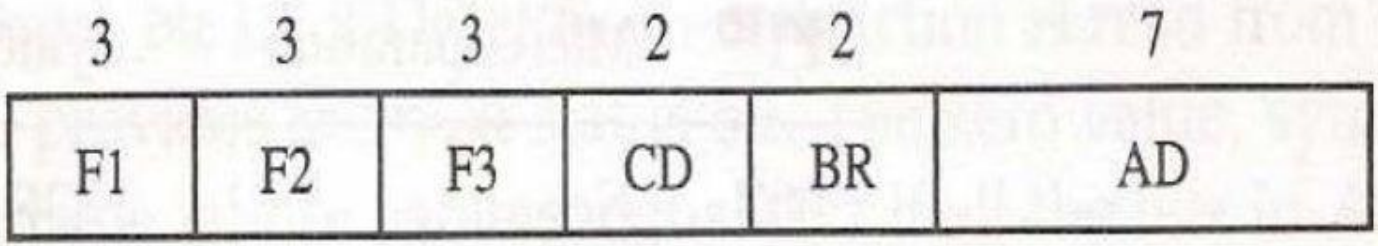


(a) Instruction format

Microinstruction Format :

The microinstruction format for the control memory is shown in Fig. 7-6. The 20 bits of the microinstruction are divided into four functional parts. The three fields F1, F2, and F3 specify microoperations for the computer. The CD field selects status bit conditions. The BR field specifies the type of branch to be used. The AD field contains a branch address. The address field is seven bits wide, since the control memory has $128 = 2^7$ words.

Microinstruction formats



F1, F2, F3: Microoperation fields

CD: Condition for branching

BR: Branch field

AD: Address field

- 2 bit Condition Fields : CD
  - » 00 : Unconditional branch, **U**
  - » 01 : Indirect address bit, **I** = DR(15)
  - » 10 : Sign bit of AC, **S** = AC(15)
  - » 11 : Zero value in AC, **Z** = AC = 0
- 2 bit Branch Fields : BR
  - » 00 : **JMP**
    - Condition = 0 : 1 $CAR \leftarrow CAR + 1$
    - Condition = 1 : 2 $CAR \leftarrow AD$
  - » 01 : **CALL**
    - Condition = 0 : 1 $CAR \leftarrow CAR + 1$
    - Condition = 1 : 2 $CAR \leftarrow AD, SBR \leftarrow CAR + 1$
  - » 10 : **RET** 3 $CAR \leftarrow SBR$
  - » 11 : **MAP** 4 $CAR(2-5) \leftarrow DR(11-14), CAR(0, 1, 6) \leftarrow 0$

Save Return Address

Restore Return Address

- 7 bit Address Fields : AD
  - » 128 word : 128 X 20 bit

Micro-instruction Types

- Micro instruction are two type:
  1) *vertical* micro-programming
  2) *horizontal* micro-programming

vertical micro-programming

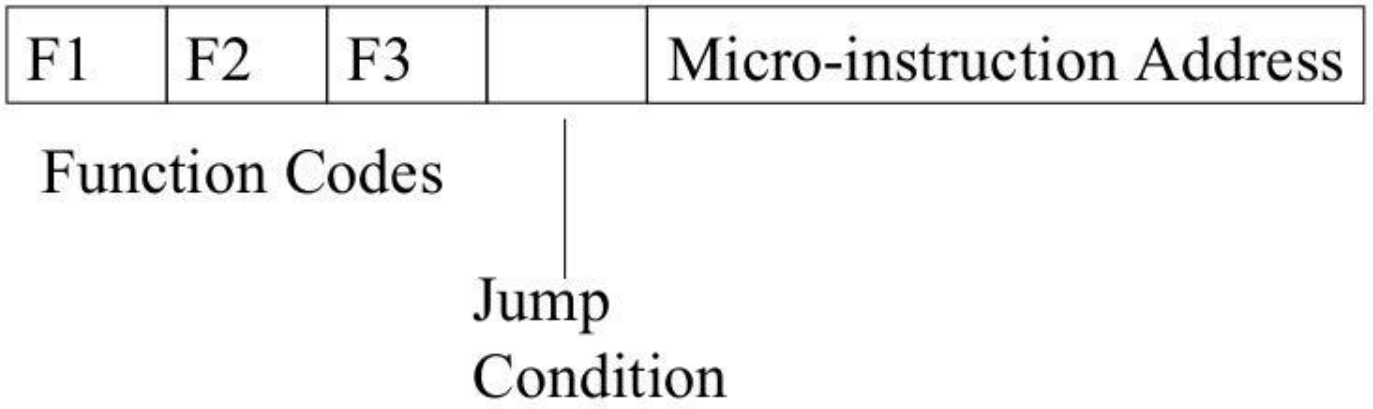
Each micro-instruction specifies single (or few) micro-operations to be performed

horizontal micro-programming

Each micro-instruction specifies many different micro-operations to be performed in parallel

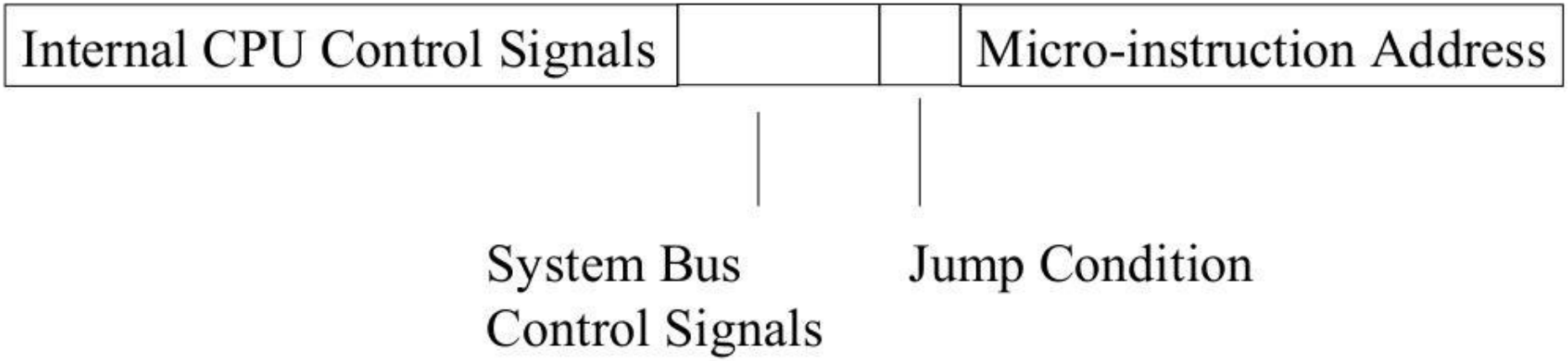
Vertical Micro-programming

- Width is narrow
- n control signals encoded into $\log_2 n$ bits
- Limited ability to express parallelism
- Considerable encoding of control information requires external memory word decoder to identify the exact control line being manipulated



Vertical Micro-programming diag

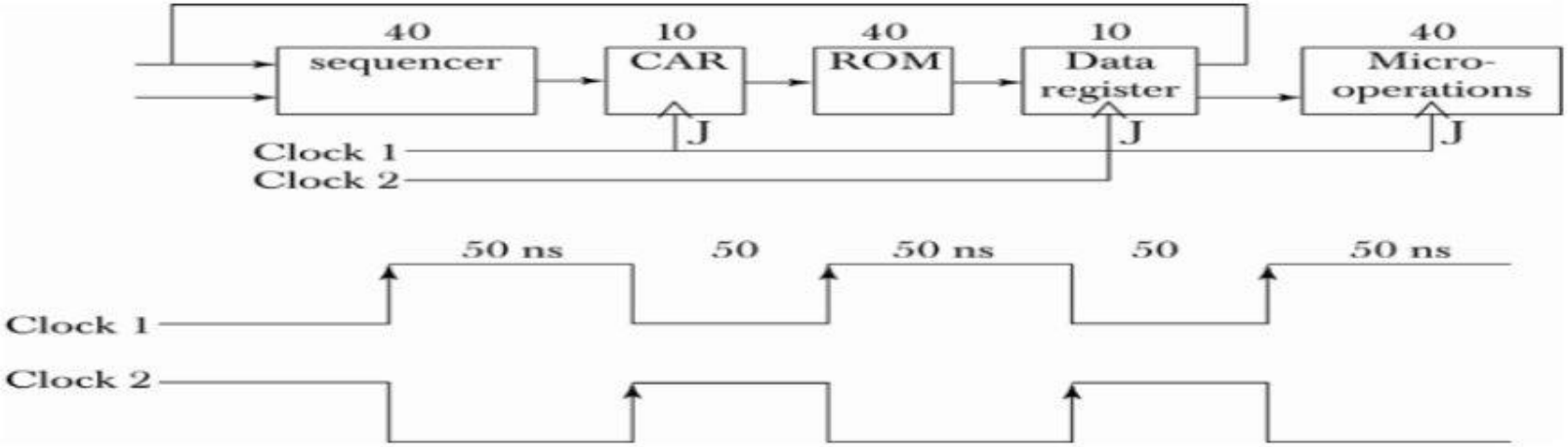
Horizontal Micro-programmed



Wide memory word
High degree of parallel operations possible
Little encoding of control information

QUESTION

The microprogrammed control organization shown in Fig. 7-1 has the following propagation delay times. 40 ns to generate the next address, 10 ns to transfer the address into the control address register, 40 ns to access the control memory ROM, 10 ns to transfer the microinstruction into the control data register, and 40 ns to perform the required microoperations specified by the control word. What is the maximum clock frequency that the control can use? What would the clock frequency be if the control data register is not used?



frequency of each clock $= \frac{1}{100\times10^{-9}} = \frac{1000}{100}\times10^{6} = 10\,\text{MHz}$.

If the data register is removed, we can use a single phase clock with a frequency of $\frac{1}{90\times10^{-9}} = 11.1\,\text{MHz}$.

The system shown in Fig. 7-2 uses a control memory of 1024 words of 32 bits each. The microinstruction has three fields as shown in the diagram. The microoperations field has 16 bits.

a. How many bits are there in the branch address field and the select field?

b. If there are 16 status bits in the system, how many bits of the branch logic are used to select a status bit?

c. How many bits are left to select an input for the multiplexers?

Control memory = $2^{10} \times 32$

(a)

6	10	16	=	32 bits
Select	Address	Micro operations		

(b) 4 bits

(c) 2 bits

The control memory in Fig. 7-2 has 4096 words of 24 bits each.

a. How many bits are there in the control address register?
b. How many bits are there in each of the four inputs shown going into the multiplexers?
c. What are the number of inputs in each multiplexer and how many multiplexers are needed?

Control memory = $2^{12} \times 24$

(a) 12 bits

(b) 12 bits

(c) 12 multiplexers, each of size 4-to-1 line.

Control Unit

- Introduction
- Types
- Comparison
- Control Memory
- Address Sequencing
- Micro instruction format and Description

Introduction

- CU is the engine that runs the entire computer with the help of the control signals.
- It perform the correct sequencing of the correct signals.
- It controls everything with a few control signals that points within processor and a few control signals to the system bus.

Introduction

- All the micro-operation are controlled by CU by performing two basic tasks:
  - Sequencing: It causes the processor to step through the series of micro-operation in proper sequence, based on program being executed.
  - Execution: It causes each micro-operation to be performed.

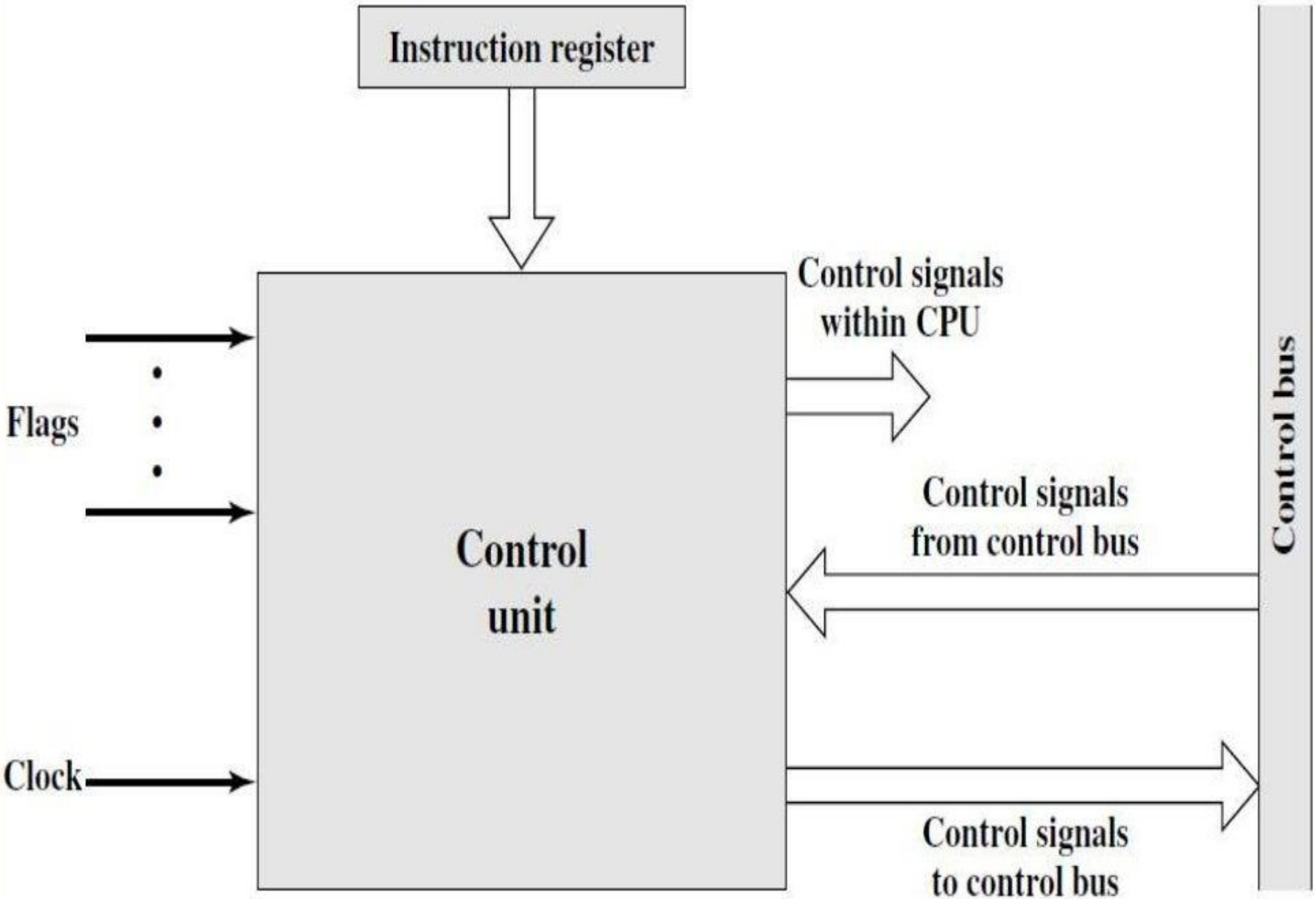


Figure Block Diagram of the Control Unit

Control Signal Sources

- Clock
  - It helps to synchronize the operation. It causes one micro-operation to be performed for each clock pulse
- Instruction Register
  - Op-code for current instruction
  - Determines which micro-instructions are performed
- Flags
  - State of CPU
  - Results of previous operations

Control Signal Sources

- From Control Bus
  - Interrupts / Bus Requests
  - Acknowledgements

Control Signal Outputs

- Within Processor
  - Cause data movement
  - Activate specific functions
- Via Main Bus
  - To memory
  - To I/O modules

Types

- There are two design approach for CU:
  - Hardwired approach
  - Micro-programming approach

Hardwired Approach

- The control signals are generated by the help of the hardware.
- It can be designed as the clock sequential circuit.
- It is implemented with logic gates, flip-flops, decoders, multiplexers and other logic buildings blocks.

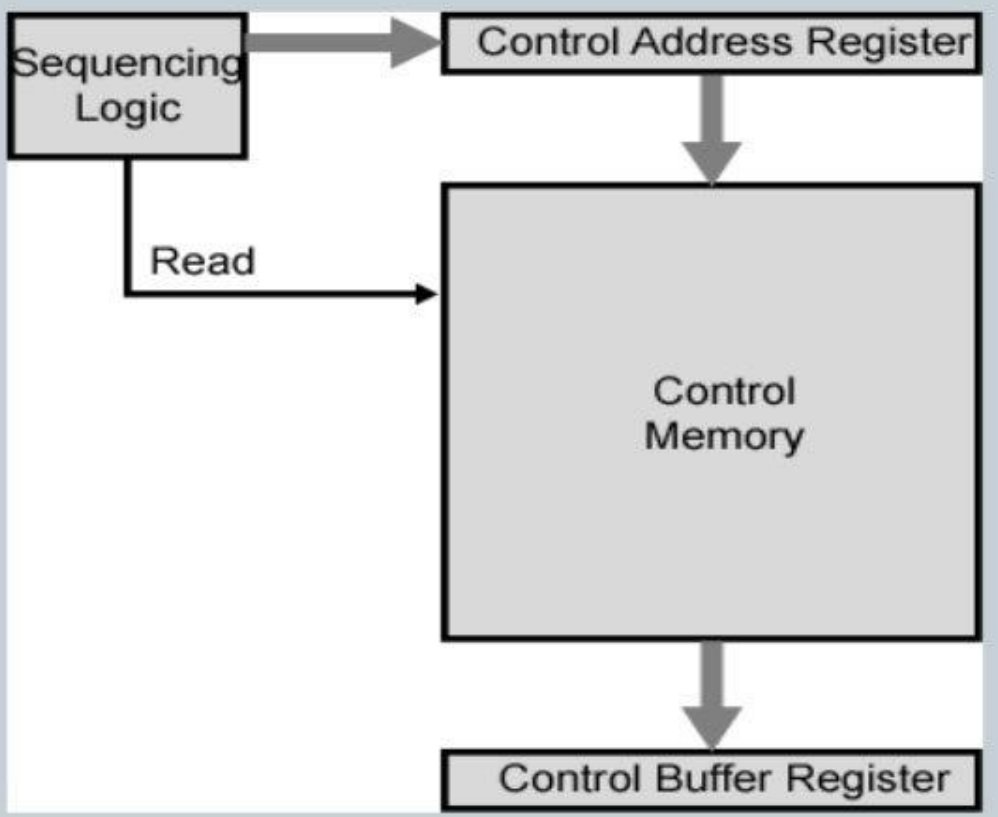
Micro programmed Approach

- All controls that can be activated simultaneously are grouped together to form the control words.
- These words are stored in the control memory.
- The control words are fetched from the control memory and are routed to various functional units to enable appropriate processing hardware.

Comparison

Attributes	Hardwired Control	Microprogramming Control
Speed	Fast	Slow
Cost of Implementation	More	Cheaper
Flexibility	Difficult to modify	Flexible
Ability to handle complex instruction	Difficult	Easier
Decoding	Complex	Easy
Application	RISC	CISC
Instruction Set Size	Small	Large
Control Memory	Absent	Present

Micro programmed Control Unit



Control Unit Function

- Sequence login unit issues read command
- Word specified in control address register is read into control buffer register
- Control buffer register contents generates control signals and next address information
- Sequence login loads new address into control buffer register based on next address information from control buffer register and ALU flags

Next Address Decision

- Depending on ALU flags and control buffer register
  - Get next instruction
    - Add 1 to control address register
  - Jump to new routine based on jump microinstruction
    - Load address field of control buffer register into control address register
  - Jump to machine instruction routine
    - Load control address register based on opcode in IR

Control Memory Organization

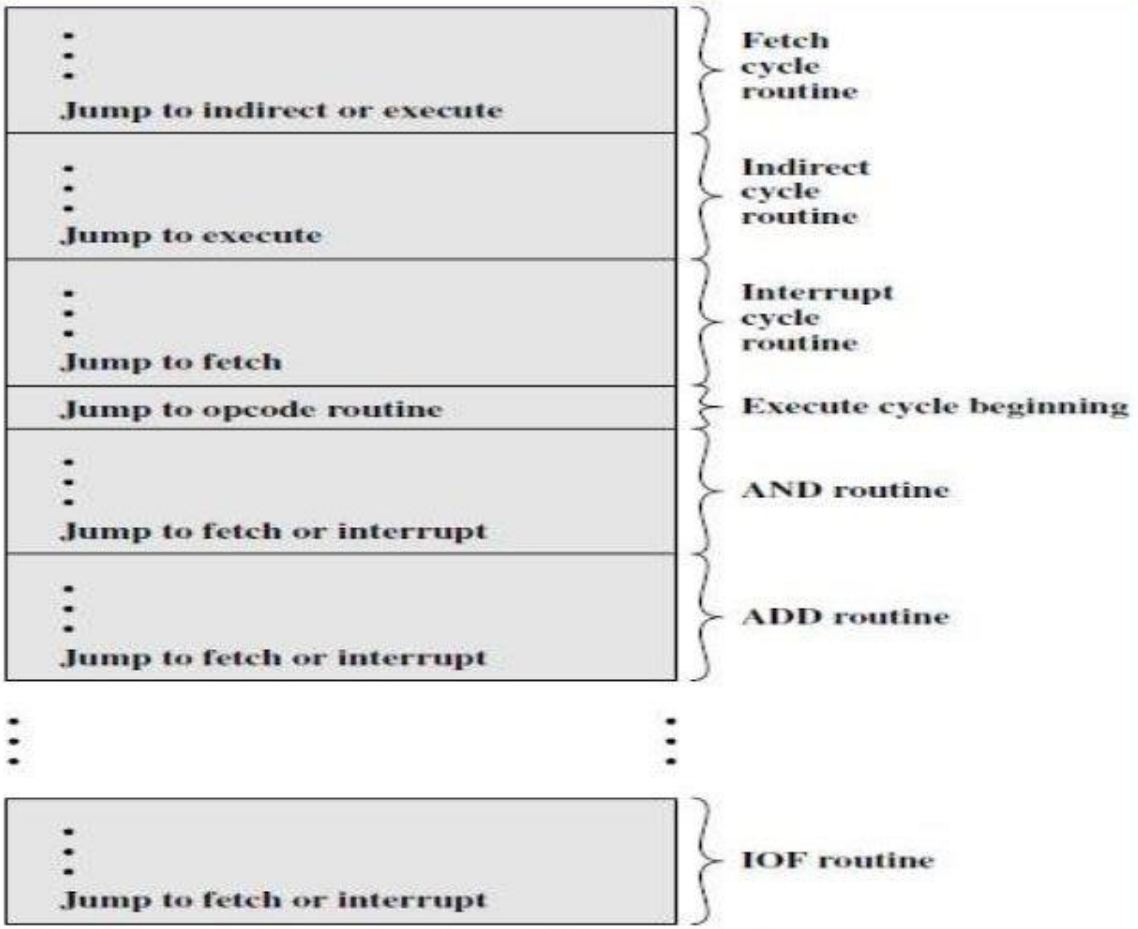
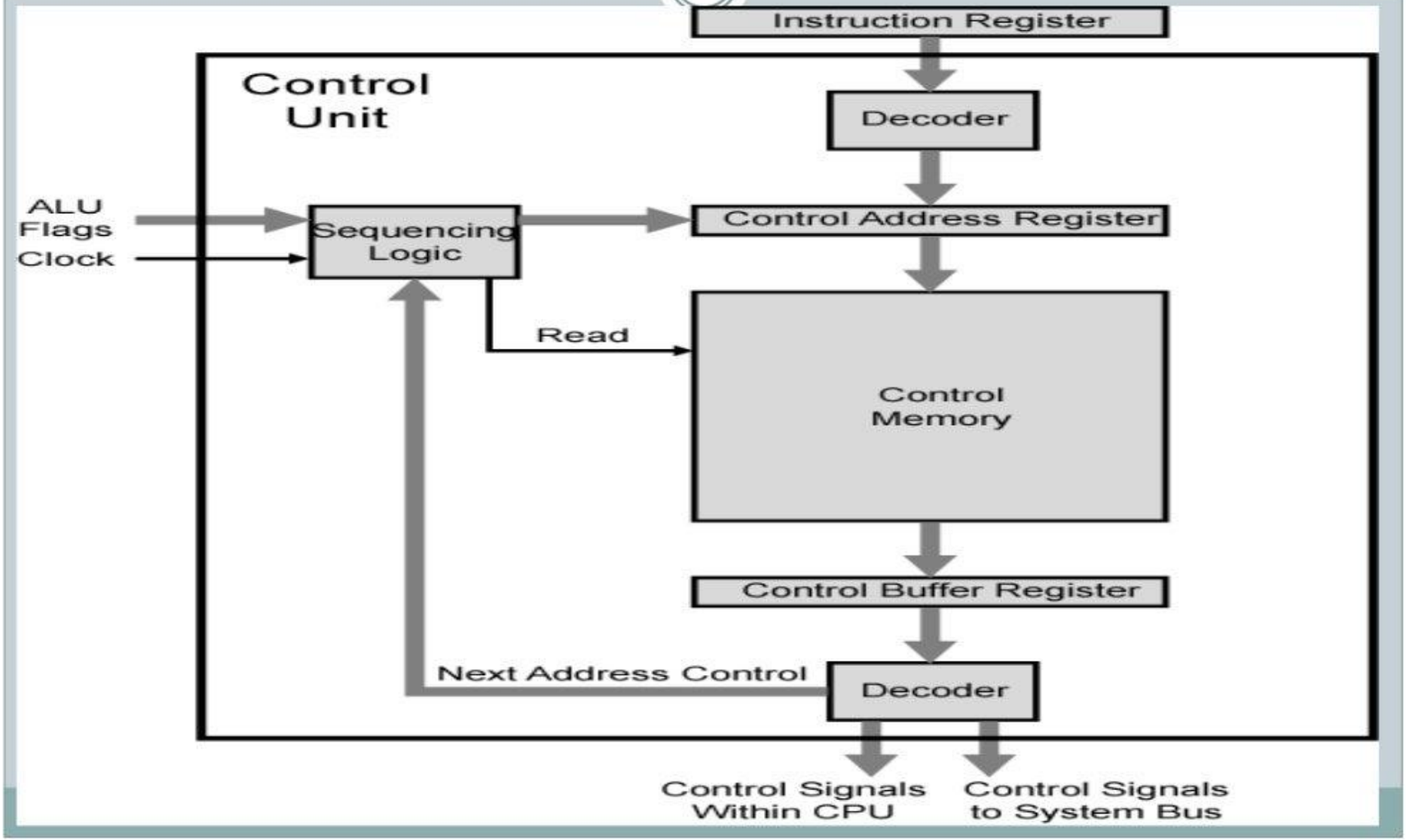
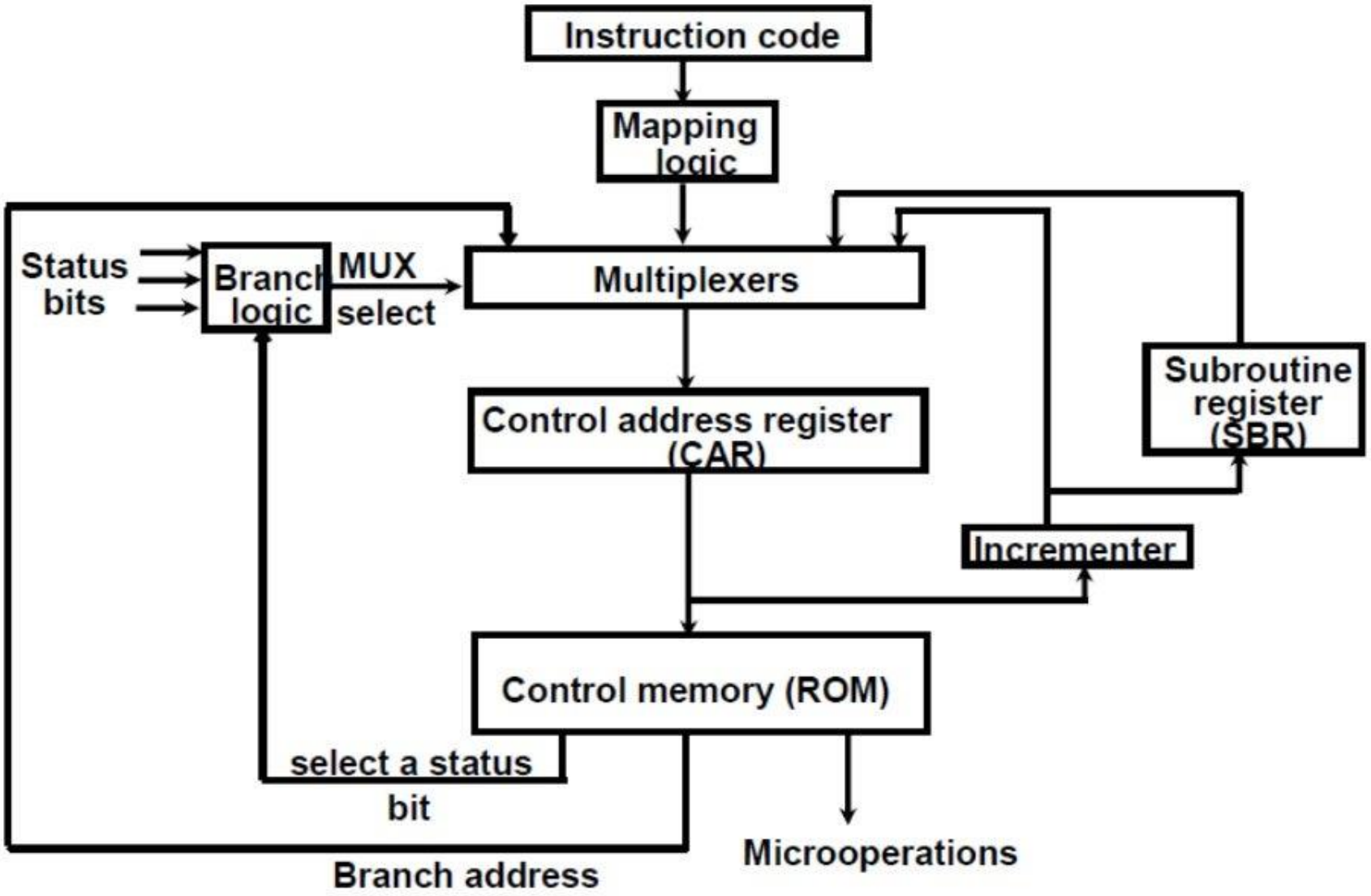


Figure Organization of Control Memory

Functioning of Micro programmed Control Unit

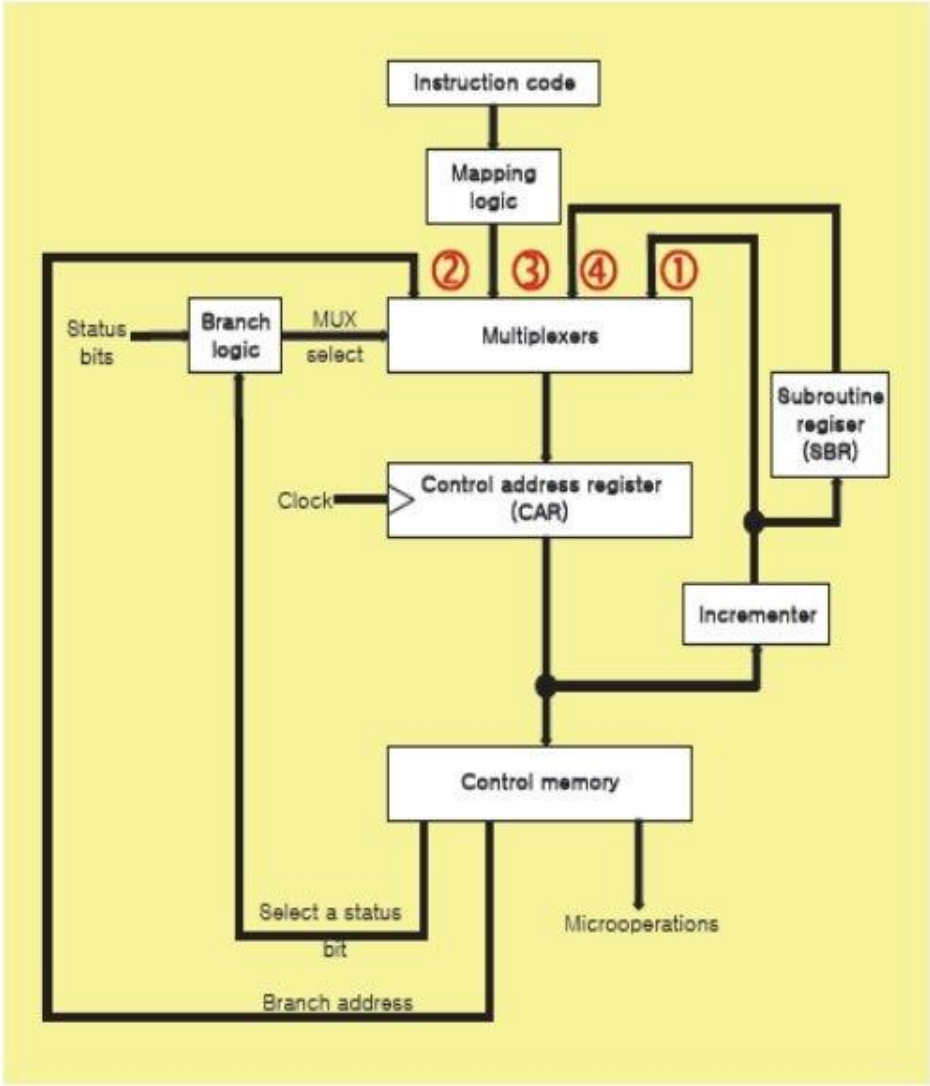


Selection of address for control memory

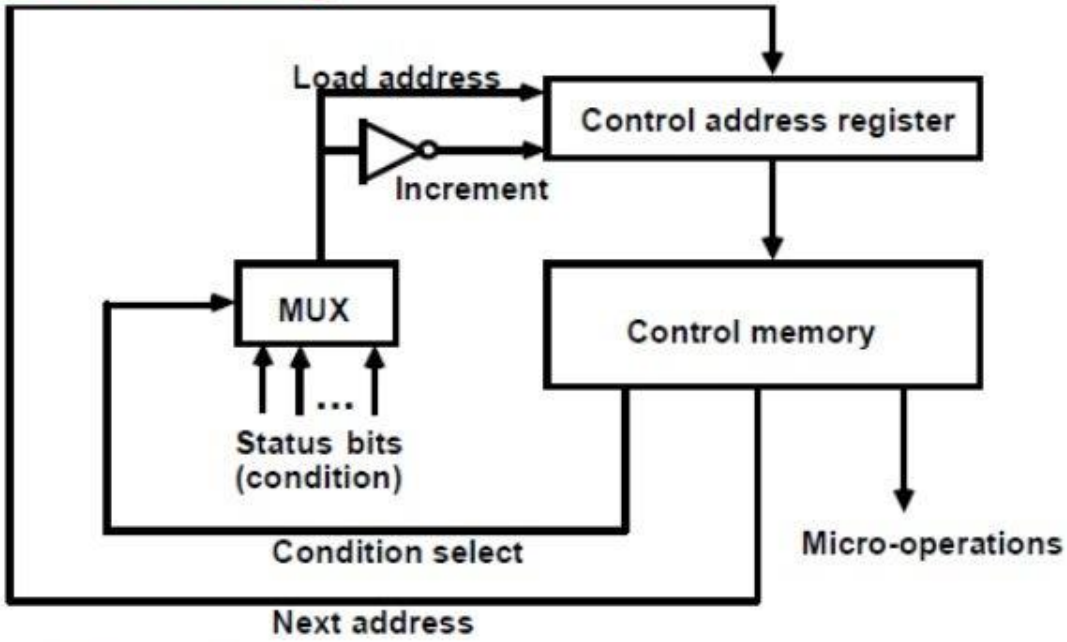


Selection of address for control memory : *Fig. 7-2*

- Multiplexer
  - ① CAR Increment
  - ② JMP/CALL
  - ③ Mapping
  - ④ Subroutine Return
- CAR : Control Address Register
  - » CAR receive the address from 4 different paths
    1) Incrementer
    2) Branch address from control memory
    3) Mapping Logic
    4) SBR : Subroutine Register
- SBR : Subroutine Register
  - » Return Address can not be stored in ROM
  - » Return Address for a subroutine is stored in SBR



Conditional Branching



Conditional Branch

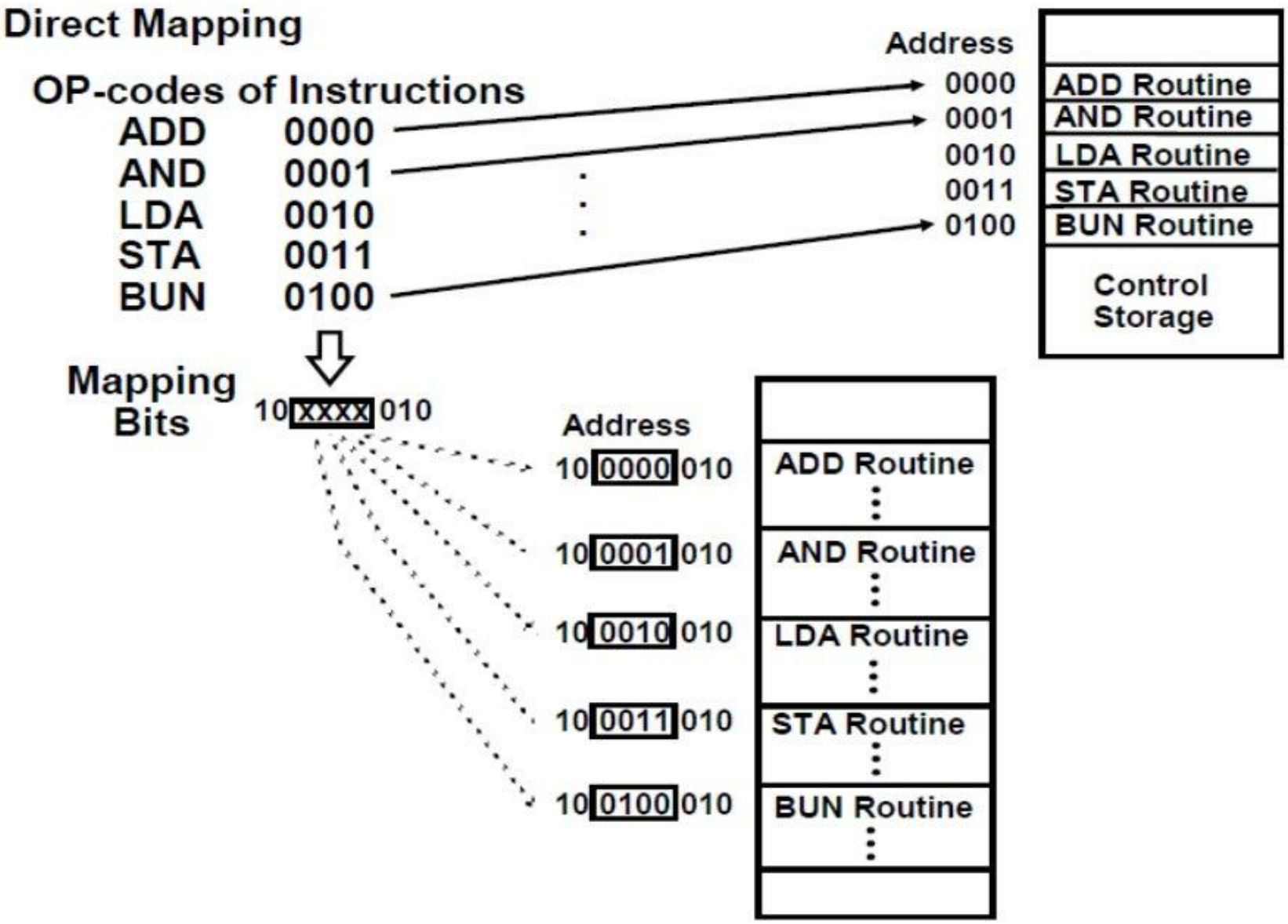
If *Condition* is true, then *Branch* (address from the next address field of the current microinstruction) else *Fall Through*

Conditions to Test: O(overflow), N(negative), Z(zero), C(carry), etc.

Unconditional Branch

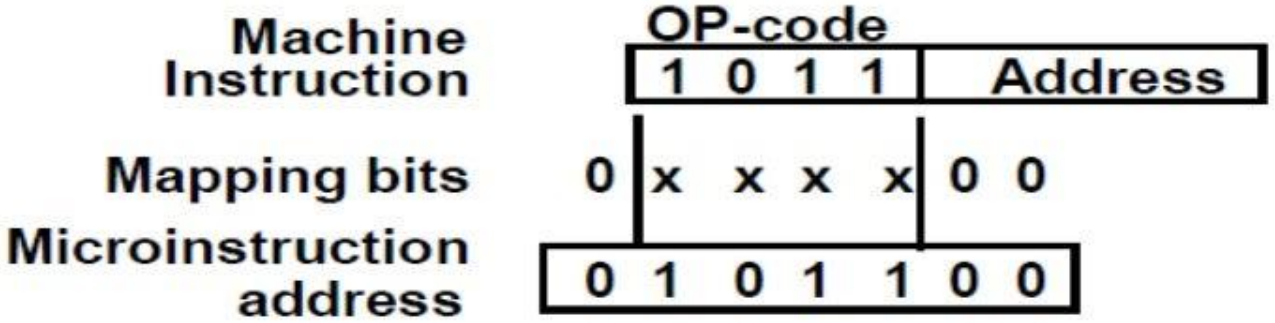
Fixing the value of one status bit at the input of the multiplexer to 1

Mapping of Instructions

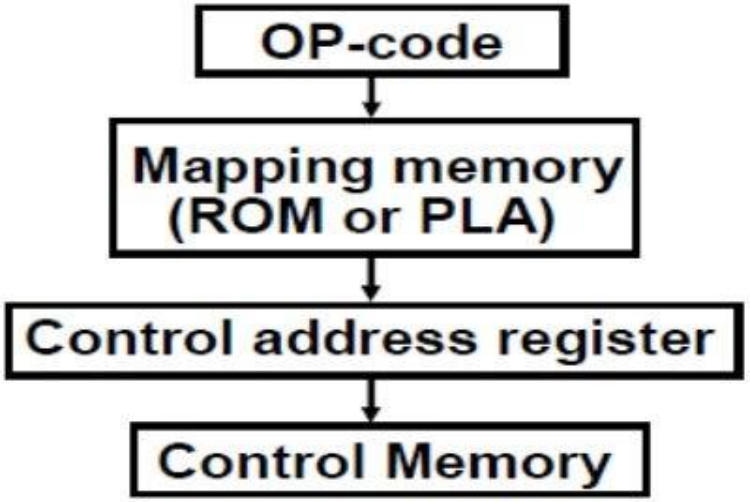


Mapping of Instructions to Microroutines

Mapping from the OP-code of an instruction to the address of the Microinstruction which is the starting microinstruction of its execution microprogram



Mapping function implemented by ROM or PLA

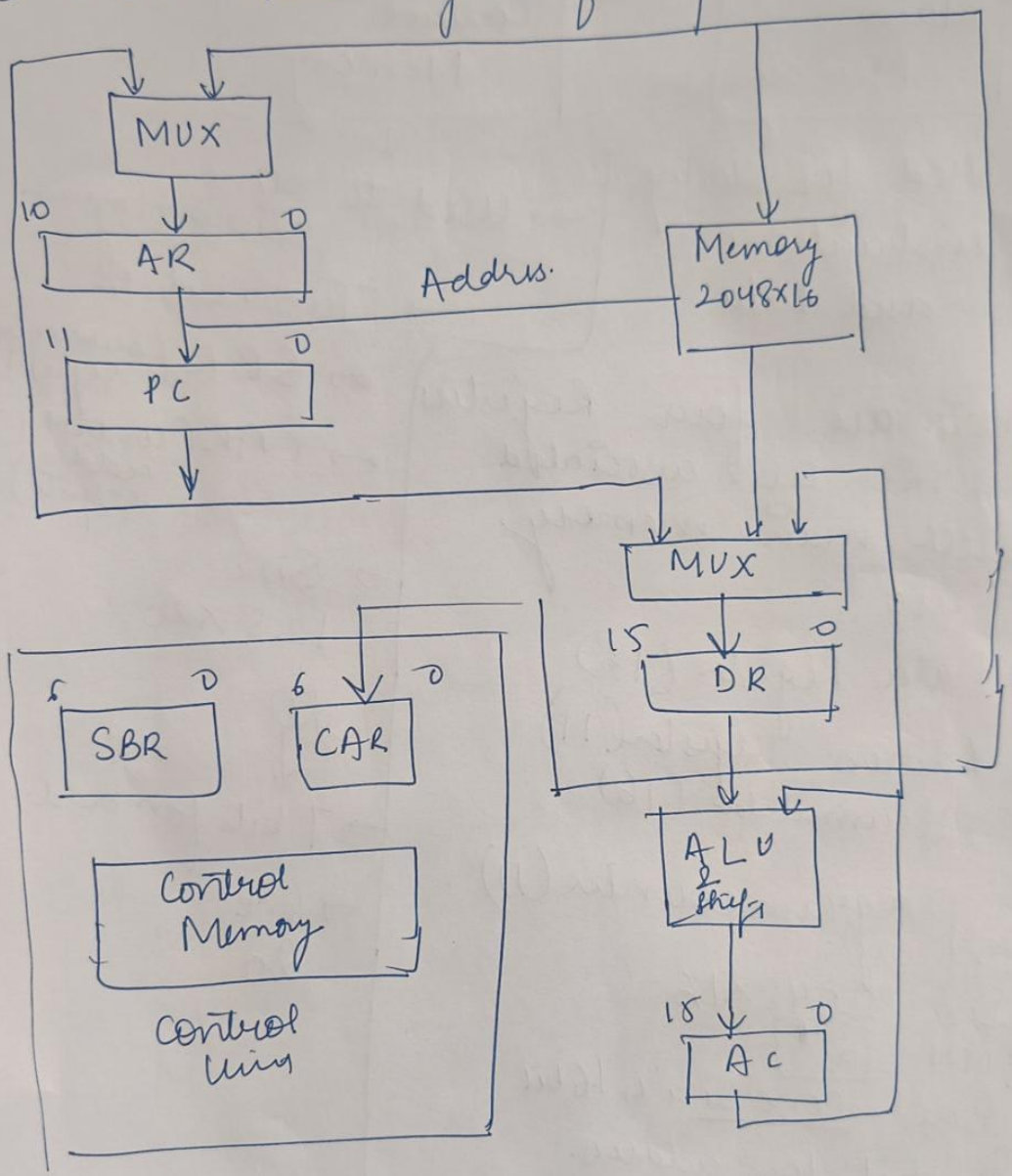


Microprogram Example

1. Computer Configuration
    - Instruction Format
    - Microinstruction Format
    - Microoperation
    - Conditional Field
    - Branch Field
    - Symbolic Microinstruction
    - Fetch Routine
    - Symbolic Microprogram
    - Execution of program
    - Binary Microprogram

Microprogram Example.

① Basic Block diagram of computer



Computer Memory

Main Memory

→ Used for storing instructions and data

Control Memory

→ Used to store microprogram

→ Two registers
- SBR (Subroutine register)
- CAR (control address register)

→ Size 128 × 20

2^7

→ 7 bits for add.

→ 6 are 20

→ There are four Registers which are associated with main memory are

(i) Data Register (16)
(ii) Address register (11)
(iii) Accumulator (16)
(iv) Program counter (11)

→ Size of MM 2048×16 $[2^{11}]$

Each core size is 16 bit

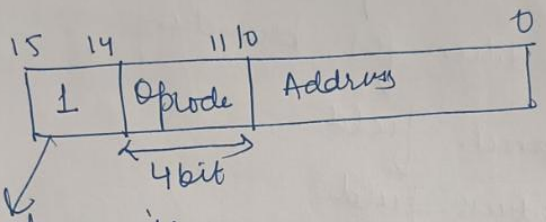
→ 11 bit for address.

→ 4 for op-code.

Multiplexer

- To transfer the information among registers
- → Generally Buses are used.
- → Data register Receive the information

Machine Instruction format



Indirect address mode bit

ie 2^4 operations = 16

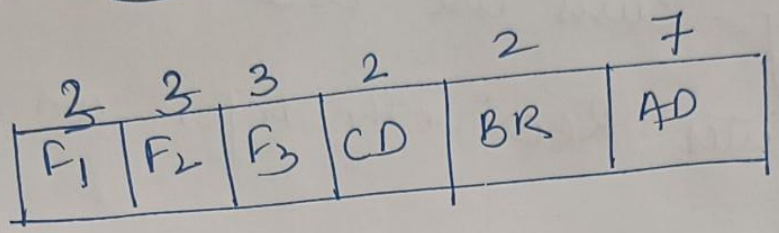
Out of 16 operation here we have demostrated 4 exg

Symbol	Opcode	Description.
ADD	0000	AC ← AC + M[EA] → Operand
BRANCH	0001	If AC < 0 then PC ← EA else M[EA] ← AC
STORE	0010	AC ← M[EA]
EXCHANGE	0011	M[EA] ← AC

Microinstruction format
(Association with control memory)

128 × 20

2^7 bits



F_1, F_2, F_3 :- Microoperation field

CD : Condition for branching

BR : Branch field

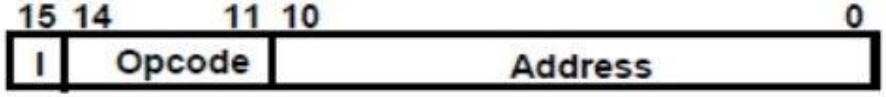
AD : Address field

Symbols & Binary code for Microinstruction fields.

Point of page 235

Machine Instruction Format

Machine instruction format

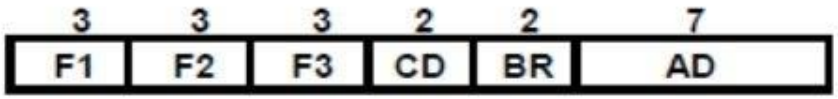


Sample machine instructions

Symbol	OP-code	Description
ADD	0000	AC ← AC + M[EA]
BRANCH	0001	if (AC < 0) then (PC ← EA)
STORE	0010	M[EA] ← AC
EXCHANGE	0011	AC ← M[EA], M[EA] ← AC

EA is the effective address

Microinstruction Format



F1, F2, F3: Microoperation fields
CD: Condition for branching
BR: Branch field
AD: Address field

F1	Microoperation	Symbol
000	None	NOP
001	AC ← AC + DR	ADD
010	AC ← 0	CLRAC
011	AC ← AC + 1	INCAC
100	AC ← DR	DRTAC
101	AR ← DR(0-10)	DRTAR
110	AR ← PC	PCTAR
111	M[AR] ← DR	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	AC ← AC - DR	SUB
010	AC ← AC ∨ DR	OR
011	AC ← AC ∧ DR	AND
100	DR ← M[AR]	READ
101	DR ← AC	ACTDR
110	DR ← DR + 1	INCDR
111	DR(0-10) ← PC	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	AC ← AC ⊕ DR	XOR
010	AC ← AC'	COM
011	AC ← shl AC	SHL
100	AC ← shr AC	SHR
101	PC ← PC + 1	INCPC
110	PC ← AR	ARTPC
111	Reserved	

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	DR(15)	I	Indirect address bit
10	AC(15)	S	Sign bit of AC
11	AC = 0	Z	Zero value in AC

BR	Symbol	Function
00	JMP	CAR ← AD if condition = 1
		CAR ← CAR + 1 if condition = 0
01	CALL	CAR ← AD, SBR ← CAR + 1 if condition = 1
		CAR ← CAR + 1 if condition = 0
10	RET	CAR ← SBR (Return from subroutine)
11	MAP	CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0

SYMBOLIC MICROINSTRUCTIONS

- Symbols are used in microinstructions as in assembly language
- A symbolic microprogram can be translated into its binary equivalent by a microprogram assembler.

Sample Format

five fields: label; micro-ops; CD; BR; AD

Label: may be empty or may specify a symbolic address terminated with a colon

Micro-ops: consists of one, two, or three symbols separated by commas

CD: one of {U, I, S, Z}, where
- U: Unconditional Branch
- I: Indirect address bit
- S: Sign of AC
- Z: Zero value in AC

BR: one of {JMP, CALL, RET, MAP}

AD: one of {Symbolic address, NEXT, empty}

DESIGN OF CONTROL UNIT

Micro-Operations

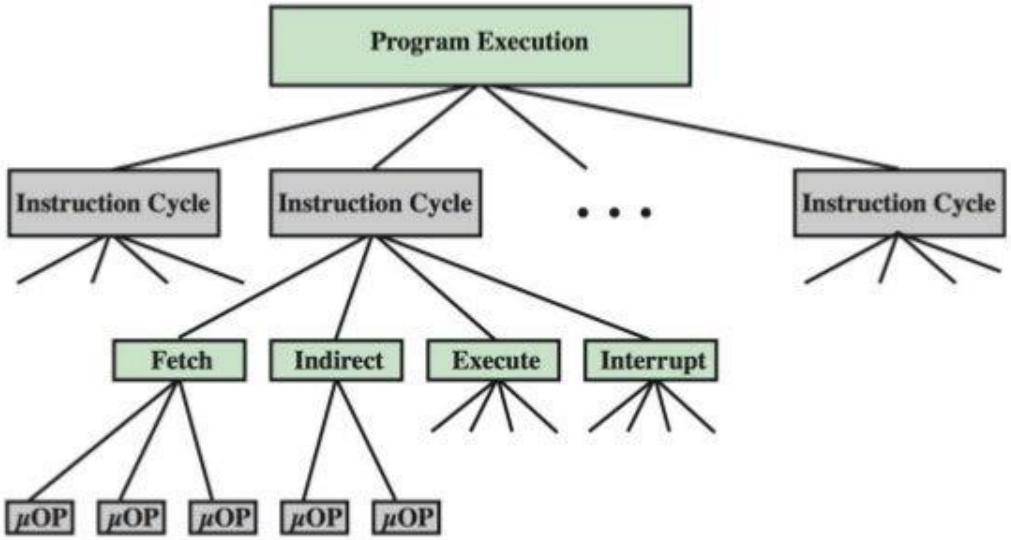


Figure 19.1 Constituent Elements of a Program Execution

Fetch Cycle

- At the beginning of the fetch cycle, the address of the next instruction to be executed is in the program counter (PC)
- The first step is to move that address to the memory address register (MAR) because this is the only register connected to the address lines of the system bus.
- The second step is to bring in the instruction. The desired address (in the MAR) is placed on the address bus, the control unit issues a READ command on the control bus, and the result appears on the data bus and is copied into the memory buffer register (MBR).
- We need to increment the PC by the instruction length to get ready for the next instruction.

- The third step is to move the contents of the MBR to the instruction register (IR).

- t1 : MAR ← (PC) …(Contents of PC)
- t2 : MBR ← Memory
- PC ← (PC) + I …(I is instruction length)
- t3 : IR ← (MBR)

Indirect Cycle

- If the instruction specifies an indirect address, then an indirect cycle must precede the execute cycle.
- t1 : MAR ← (IR(Address)) … (Content of IR (which is an Indirect address))
- t2 : MBR ← Memory
- t3 : IR(Address) ← (MBR(Address))

- The address field of the instruction is transferred to the MAR. This is then used to fetch the address of the operand.
- Finally, the address field of the IR is updated from the MBR, so that it now contains a direct rather than an indirect address.

Interrupt Cycle

- At the completion of the execute cycle, a test is made to determine whether any enabled interrupts have occurred. If so the interrupt cycle occurs.

- t1 : MBR ← (PC)
- t2 : MAR ← Save_Address
- PC ← Routine_Address
- t3 : Memory ← (MBR)

- Save the PC contents in memory and update PC with Routine Address
- It may take one or more additional micro-operations to obtain the Save_Address and the Routine_Address before they can be transferred

Execute Cycle

- Because of the variety opcodes, there are a number of different sequences of micro-operations that can occur.
- The control unit examines the opcode and generates a sequence of micro-operations based on the value of the opcode.
- This is referred to as instruction decoding.

- t1 : MAR ← (IR(address))
- t2 : MBR ← Memory
- t3 : R1 ← (R1) + (MBR)

Instruction Cycle

- A 2-bit register called the instruction cycle code (ICC) designates the state of the processor in terms of which portion of the cycle it is in:
- 00: Fetch
- 01: Indirect
- 10: Execute
- 11: Interrupt
- At the end of each of the four cycles, the ICC is set appropriately.
- The indirect cycle is always followed by the execute cycle.
- The interrupt cycle is always followed by the fetch cycle.
- For both the fetch and execute cycles, the next cycle depends on the state of the system.

Wilkie's Microprogrammed CU

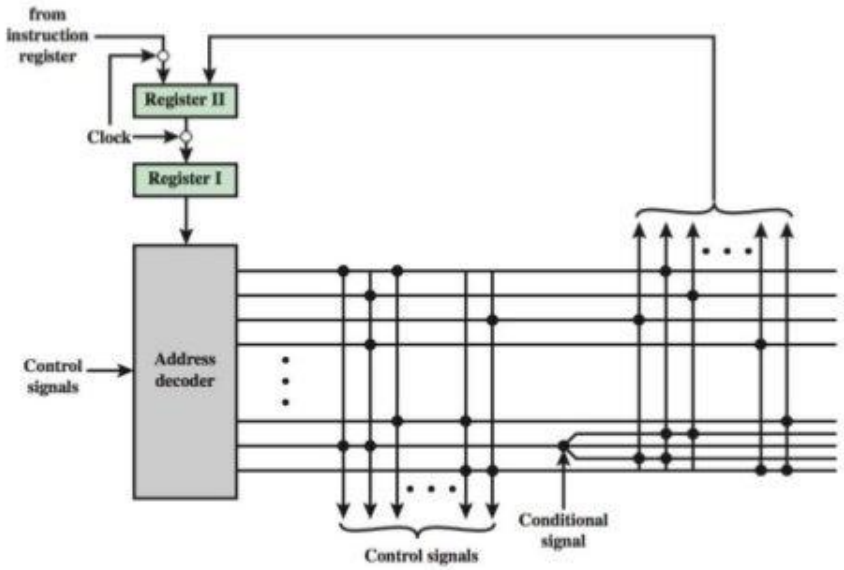


Figure 20.5 Wilkes's Microprogrammed Control Unit

- During a machine cycle, one row of the matrix is activated with a pulse. This generates signals at those points where a diode is present (indicated by a dot in the diagram).
- Each row of the matrix is one microinstruction, and the layout of the matrix is the control memory.
- At the beginning of the cycle, the address of the row to be pulsed is contained in Register I.
- The decoder, when activated by a clock pulse, activates one row of the matrix.
- Depending on the control signals, either the opcode in the instruction register or the second part of the pulsed row is passed into Register II during the cycle.

- Register II is then gated to Register I by a clock pulse.
- Alternating clock pulses are used to activate a row of the matrix and to transfer from Register II to Register I.
- The two-register arrangement is needed because the decoder is simply a combinatorial circuit; with only one register, the output would become the input during a cycle, causing an unstable condition.

Thank You