IAS Computer

Instructions

The IAS computer was designed in the 1940's and built in the early 1950's by John von Neumann at the Princeton Institute for Advanced Studies. It can arguably be called the father of all modern computers in that it was one of the first computers which stored both the program and the data in the computer's memory and it was used as a prototype for many of the computers developed in the 1950's, including the IBM family of computers. The IAS has an accumulator register, `AC`, and an arithmetic register, `AR`, to store the results of operations. Each register has 40 bits. There is a memory (RAM) called `Selectron` that has up to 4096 (that is, 2^{12}) 40-bit memory locations—words. There are also several other registers that are used internally by the computer but are invisible to the machine-language programmer.

inst name	opcode	description
S(x)->Ac+	1	copy the number in Selectron location x into AC
S(x)->Ac-	2	same as #1 but copy the negative of the number
S(x)->AcM	3	same as #1 but copy the absolute value
S(x)->Ac-M	4	same as #1 but subtract the absolute value
S(x)->Ah+	5	add the number in Selectron location x into AC
S(x)->Ah-	6	subtract the number in Selectron location x from AC
S(x)->AhM	7	same as #5, but add the absolute value
S(x)->Ah-M	8	same as #7, but subtract the absolute value
S(x)->R	9	copy the number in Selectron location x into AR
R->A	10	copy the number in AR to AC
S(x)*R->A	11	Multiply the number in Selectron location x by the number in AR. Place the left half of the result in AC and the right half in AR.
A/S(x)->R	12	Divide the number in AC by the number in Selectron location x. Place the quotient in AR and the remainder in AC.
Cu->S(x)	13	Continue execution at the left-hand instruction of the pair at Selectron location x
Cu`->S(x)	14	Continue execution at the right-hand instruction of the pair at Selectron location x
Cc->S(x)	15	If the number in AC is >= 0, continue as in #13. Otherwise, continue normally.
Cc`->S(x)	16	If the number in AC is >= 0, continue as in #14. Otherwise, continue normally.
At->S(x)	17	Copy the number in AC to Selectron location x
Ap->S(x)	18	Replace the right-hand 12 bits of the left-hand instruction at Selectron location x by the right-hand 12 bits of AC
Ap`->S(x)	19	Same as above, but modifies right-hand instruction
L	20	Shift the number in AC to the left 1 bit (new bit on the right is 0)
R	21	Shift the number in AC to the right 1 bit (leftmost bit is copied)
halt	0	Halt the program (see paragraph 6.8.5 of IAS report)

AC: Accumulator
AR: Arithmetic Register

Prepared by:- Anwar Bari

The IAS machine language instructions are 20 bits long and have the following form:*op-code memory-address*, where *op-code* is an 8-bit operation code and *memory-address* is a 12-bit memory address. The table above lists the complete IAS instruction set. There are 22 machine instructions for the IAS computer, each associated with an 8-bit op-code, as shown in the table below. Note that the opcode is given in decimal but, of course, is stored in the computer in binary. Note also that not all instructions require a memory address, in which case the last 12 bits of the instruction are ignored.

For example, the machine language statement `0000001 000000000001110` says to get the data in Selectron memory location 14 and load it into the AC.

Since each addressable cell in memory (each word) contains 40 bits, two instructions fit in each word of memory. If the word is used to store data instead of instructions, then all 40 bits are used to store one data value.

Data Path

Below is the diagram of the CPU data paths for the IAS computer. This comes from Stallings 2010; the register labeled MQ is the register we have been calling AR (arithmetic register).

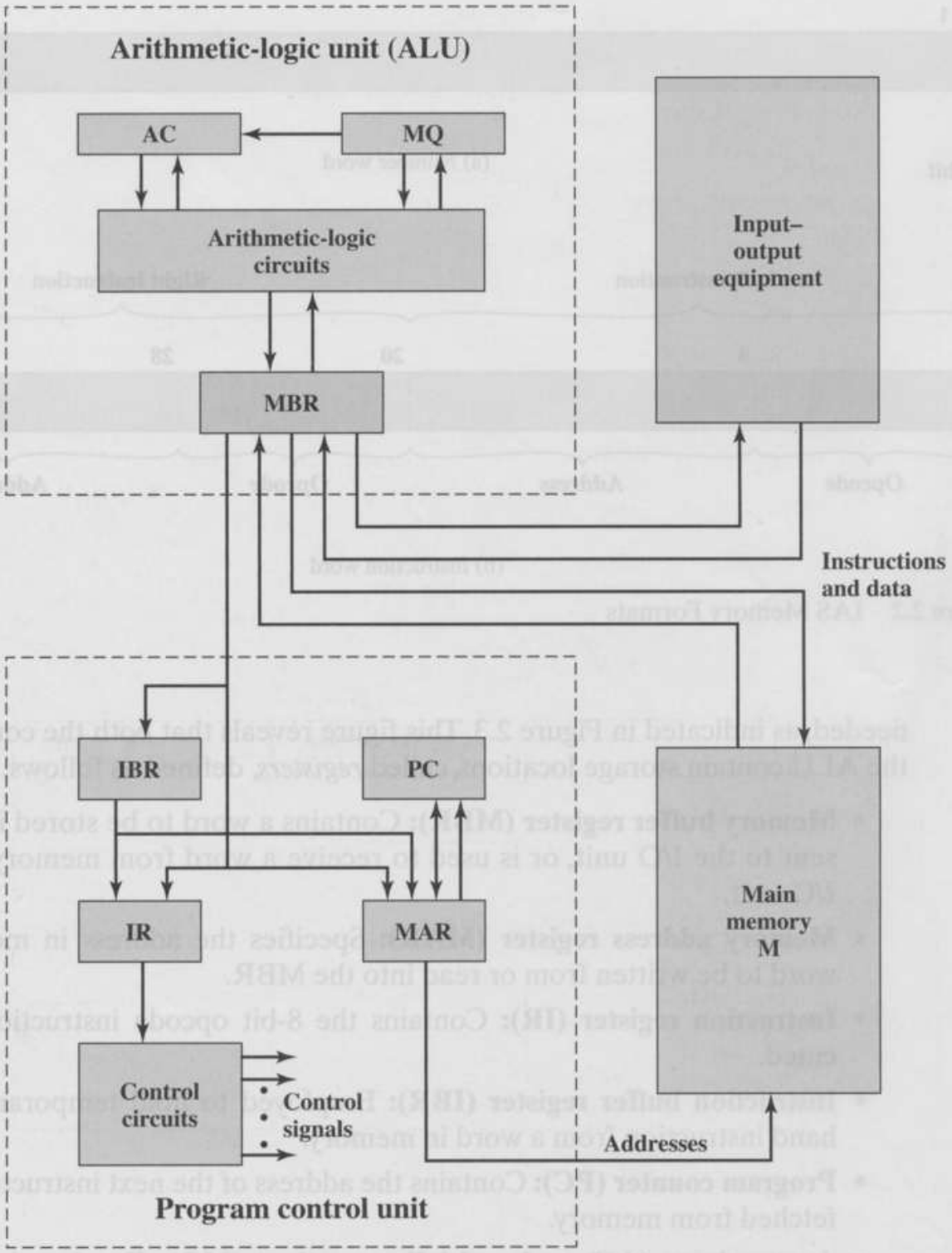


Figure 2.3 Expanded Structure of IAS Computer

1.2.2 INSTRUCTION INTERPRETATION CYCLE

Interpretation of an instruction proceeds in three steps or cycles. The instruction is fetched, decoded, and executed. These three steps are discussed in the following sub sections.

Instruction Fetch

A partial flow chart for the instruction fetch cycle is shown in Figure 1.6. Because two instructions are fetched at once, the first step is to determine if a fetch from memory is required. This test is made by testing the least-significant bit (LSB) of the program counter. Thus, an instruction fetch from memory occurs only on every other state of the PC or if the previous instruction is a taken branch. The fetch from memory places a left (L) and a right (R) instruction in the instruction buffer register (IBR).

Instructions are executed, except for the case of a branch instruction, left, right, left, right, etc. For example, consider that an R instruction has just been completed.

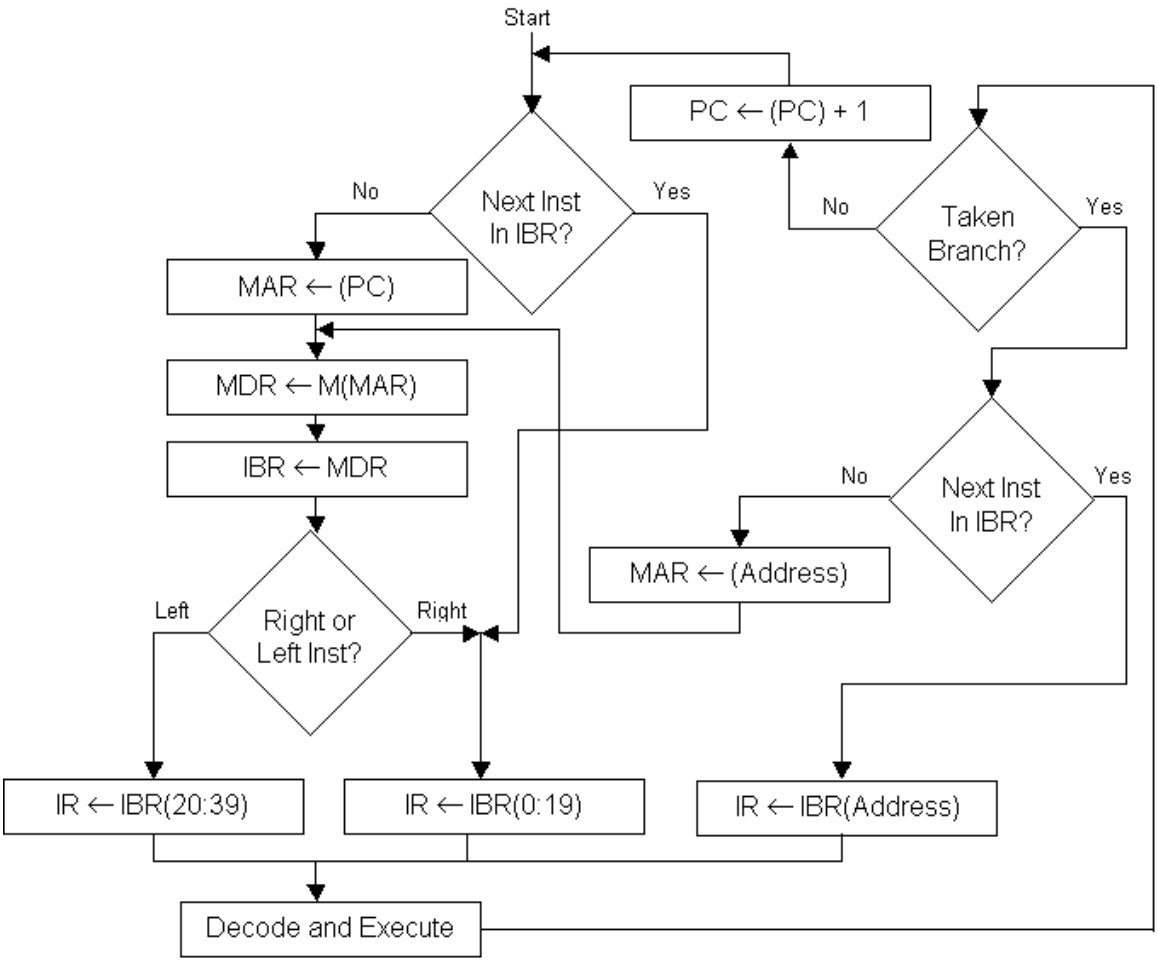


Figure 1.6 Instruction fetch cycle

There is no instruction in the IBR and a reference is made to memory to fetch an instruction pair.

Normally, the L instruction is then executed. The path follows to the left, placing the instruction into the instruction register (IR). The R instruction remains in the IBR for use on the next cycle, thereby saving a memory cycle to fetch the next instruction.

If the prior instruction had been a branch to the R instruction of the instruction pair, the L instruction is not required, and the R instruction is moved to the IR. In summary, the instruction sequence is as follows:

Sequence	*Action*
L followed by R	No memory access required
R followed by L	Increment PC, access memory, use L instruction
L branch to L	Memory access required and L instruction used
R branch to R	Memory access required and R instruction used
L branch to R	If in same computer word, memory access not required
R branch to L	If in same computer word, memory access not required

After the instruction is decoded and executed, the PC is incremented for the next instruction and control returns to the start point.

Decode and Execute

Instruction decode is only indicated in Figure 1.6. However, the instruction has been placed in the IR. As shown in Figure 1.7, combinatorial logic in the control unit decodes the op-code and decides which of the instructions will be executed. In other words, decoding is similar to the CASE statement of many programming languages. The flow charts for two instruction executions are shown in Figure 1.7: numbers 21 and 6. After an instruction is executed, control returns to the instruction fetch cycle, shown in Figure 1.6.

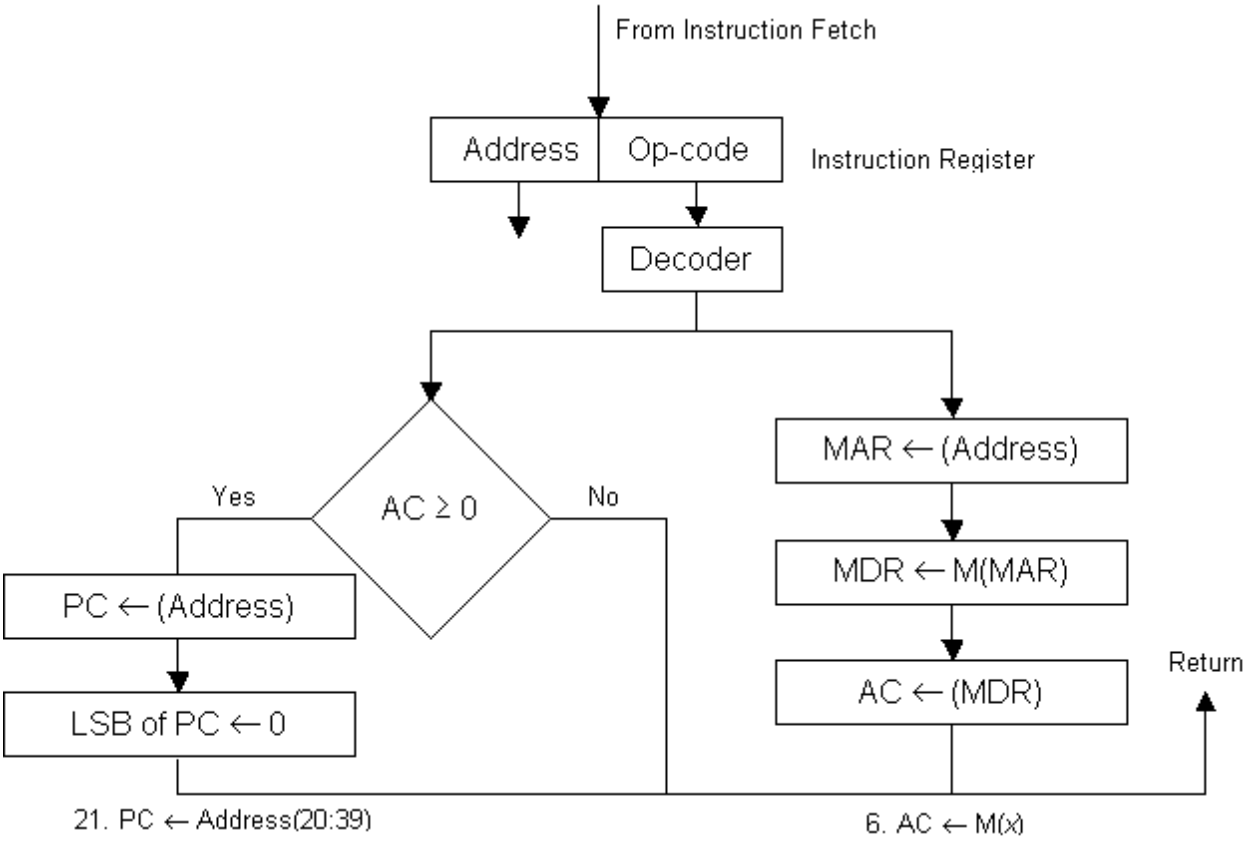


Figure 1.7 Decode and execute

The sequencing of the instruction interpretation cycle is controlled by a hardwired state machine, discussed in Chapter 5. Each of the states is identified in flowchart form, flip flops are assigned to represent each state, and the logic is designed to sequence through the states. After the invention of microprogramming, the flow chart is reduced to a series of instructions that are executed on the micromachine. In other words, a second computer, rather than a hardwired state machine, provides the control.

Addressing Modes & CPU Internals

- In the following descriptions I will use a hypothetical 3-operand ADD instruction:

  ADD Dest, Src1, Src2

  and use different addressing modes for Src2.

- Note that I am ignoring the Src1 and Dest operands. These, too, could use any of the following addressing modes.

Immediate Addressing

ADD Dest, Src1, literal value

- The operand's value is encoded in the instruction as a literal value.
- It may be treated as signed or unsigned depending on the instruction.
- MIPS example: `addi $t1, $t2, 100`
- Literal values are known on some architectures as immediate values.

Register Mode

ADD Dest, Src, Register

- A register number is specified in the instruction; the operand's value comes from the register.
- MIPS example: `add $t1, $t2, $t3`

Absolute or Direct Mode

ADD Dest, Src, memory address

- The instruction encodes a memory address; the operand's value is loaded from that address.
- Note that most RISC ISAs only provide memory addressing for load & store instructions.
- CISC ISAs usually allow memory addressing for many instructions.
- MIPS example: `lw $t1, 5000`

Register Indirect Mode

ADD Dest, Src, (Register)

- The instruction encodes a register number.
- The value in the register is treated as an address; the operand's value is loaded from that address.
- The register is effectively a **pointer**: instead of holding a value, the register points at the location in memory where that value exists.

- MIPS example: `lw $t1, ($t2)`
- The idea of a pointer is used in many languages to create dynamic data structures, e.g. linked lists.
- In Java, a linked list of ints can be created with:

```
public class IntList
{
   private int     x;
   private IntList next;
}
```

although the `next` variable is strictly a **reference**, not a pointer: Java stops you from seeing the address.

- In C, we can do:

```
struct IntList
{
  int             x;
  struct IntList *next;
}
```

and the `next` variable is a pointer which holds an address and which is directly visible to the programmer.

Indexed Absolute Mode

ADD Dest, Src, base(Register)

- The instruction encodes a literal value, the *base*, and a register number.
- The literal value, with the register's value added, is treated as an address; the operand's value is loaded from that address.
- The register is known as an **index** register.
- MIPS example: `lw $t1, 300($t2)`
- This allows the low-level implementation of arrays.
- Assume that an array of 25 characters, `name[]`, is stored starting at memory address 50 onwards.
- The character `name[0]` is stored at location 50, `name[1]` at 51, `name[2]` at 52 etc.
- A program can loop with register `Rn` going from 0 up to 24.
- The operand `50(Rn)` would access each of the characters in the array from position 0 up to 24.

- Note that, for arrays of different type sizes, e.g. 32-bit ints, the index register has to be incremented by the byte size of the elements (i.e. 4) each time through the loop.
- This addressing mode is useful when the array is fixed in memory: for example, it is a global array defined at compile time.
- The rest of the addressing modes below are seen mainly on CISC architectures.

Base plus Offset Mode

ADD Dest, Src, BaseRegister(offset)

- This time, the offset is the fixed literal, and the base of the data structure is obtained from a register.
- This addressing mode is useful to access specific fields of a structure or object.
- Consider a structure of different-sized fields, e.g. a C struct:

```
struct Student
{
  int id;          # 32-bit value
  short age;       # 16-bit value
  char gender;     # 8-bit value
}
```

- The `id` field is at offset 0 in the struct, the `age` field is at offset 4, and the `gender` field is at offset 6.
- Imagine you have a Student struct variable `s1`, and you want to set the `age` in the struct to 23:

```
s1.age = 23;
```

- If register `R5` (playing the role of `s1`) points at the base of the struct, then the following would do the work:

```
StoreHalfword R5(4), 23
```

as we know the `age` field is 4 bytes from the address that `R5` points to.

Base plus Index Mode

ADD Dest, Src, BaseRegister(IndexRegister)

- Instead of encoding the base as a fixed literal value, the base value comes from a register. The values of the two registers are added together to form the address of the value to access.

- The base register is effectively a pointer to some data structure like an array, and the index pointer is the index into that array.
- This is useful when the data structure's location is not defined at compile-time, but is created at run-time by the program:
    - the structure might be a local variable, created when the function starts.
    - the structure might be dynamically allocated, e.g. with C's `malloc()` or Java's `new` operator.
- Assuming that the base register points at the base of the structure, this addressing mode can access the fields or elements in the structure.

Base plus Index plus Offset Mode

ADD Dest, Src, BaseRegister(IndexRegister)offset

- The instruction encodes two registers and a literal value which is the offset
- The values of all three are added together to form the address of the value to access.
- This addressing mode is useful when there is an array of structs/objects, and you want to access a specific field within one of the objects, e.g.
- `studentlist[57].age = 23;`
-
- The base register would point at the base of the array, the index would hold 57 * sizeof(Student), and the offset would be the offset of the `age` field from the base of each struct/object.

Register Pre-increment/Pre-decrement Mode

ADD Dest, Src, +Register or -Register

- The value in the register is pre-incremented or pre-decremented before it is used.
- This mode often appears in combination with the other addressing modes above.
- Used in many 1960s and 1970s ISAs, its use on the PDP-11 CPU inspired the pre-increment & pre-decrement operators in C, which have been adopted by its decendants like Java, i.e. `++x`;

Register Post-increment/Post-decrement Mode

ADD Dest, Src, Register+ or Register-

- The value in the register is incremented or decremented after it is used.
- This mode often appears in combination with the other addressing modes above.

Memory Indirect Mode

ADD Dest, Src, (memory address)

- The value at the given memory address is fetched, and this is used as a pointer to the actual memory location.
- If you ever get to write a substantial amount of C programming, you will see this in action as a pointer to a pointer.
- As shown above, there is a single indirection: fetch the value to get the address, fetch the value at that address.
- On some systems, e.g. the PDP-10 minicomputer, the word size (i.e. data bus size) was much bigger than the address bus size, so there were extra bits left over in each word which were not needed as an address.
- One specific bit in a word was treated as an indirect bit:
    - if it was set, then the CPU would treat the value as another address, and go and fetch the value at this address.
- This implies that you could chain addresses: one address could point at another one, which could point at a further one etc.
- One prank was to set the indirect bit in a word in memory, and put the address of that word in the word itself.
    - Accesses to this location would cause the CPU to go into an endless loop of indirections.
    - On some implementations of the PDP-10 ISA, this would lock the CPU up hard, and it would need to be powered down to fix the problem.

Data Transfer & Manipulation

Computer provides an extensive set of instructions to give the user the flexibility to carryout various computational task. Most computer instruction can be classified into three categories.

(1) Data transfer instruction
(2) Data manipulation instruction
(3) Program control instruction

Data transfer instruction cause transferred data from one location to another without changing the binary instruction content. Data manipulation instructions are those that perform arithmetic logic, and shift operations. Program control instructions provide decision-making capabilities and change the path taken by the program when executed in the computer.

(1) Data Transfer Instruction

Data transfer instruction move data from one place in the computer to another without changing the data content. The most common transfers are between memory and processes registers, between processes register & input or output, and between processes register themselves

(Typical data transfer instruction)

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

(2) Data Manipulation Instruction

It performs operations on data and provides the computational capabilities for the computer. The data manipulation instructions in a typical computer are usually divided into three basic types.

(a) Arithmetic Instruction
(b) Logical bit manipulation Instruction
(c) Shift Instruction.

(a) Arithmetic Instruction

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	Add
Subtract	Sub
Multiply	MUL
Divide	DIV
Add with Carry	ADDC
Subtract with Basses	SUBB
Negate (2's Complement)	NEG

(b) Logical & Bit Manipulation Instruction

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-Or	XOR
Clear Carry	CLRC
Set Carry	SETC
Complement Carry	COMC
Enable Interrupt	ET
Disable Interrupt	OI

(c) Shift Instruction

Instructions to shift the content of an operand are quite useful and one often provided in several variations. Shifts are operation in which the bits of a word are moved to the left or right. The bit-shifted in at the and of the word determines the type of shift used. Shift instruction may specify either logical shift, arithmetic shifts, or rotate type shifts.

Name	Mnemonic
Logical Shift right	SHR
Logical Shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL

Rotate mgmt through carry	RORC
Rotate left through carry	ROLC

Computer Bus Structure

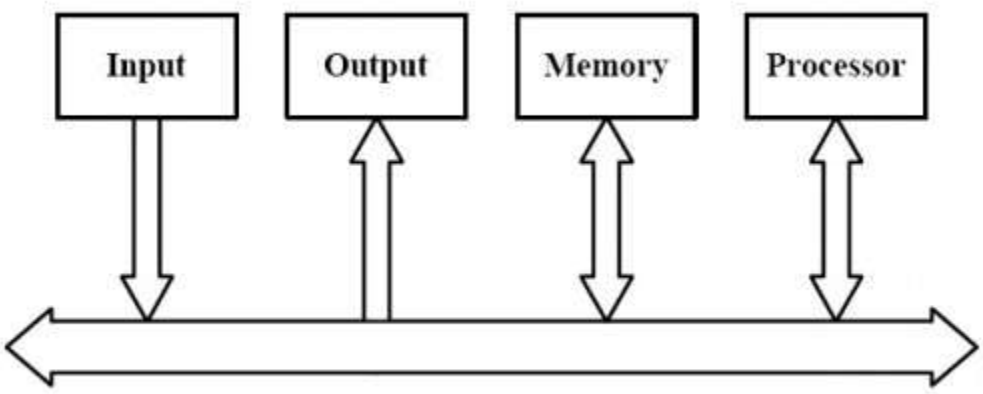
A ***bus*** is a collection of wires that connect several devices within a computer system. When a word of data is transferred between units, all its bits are transferred in parallel. A computer must have some lines for addressing and control purposes.

Three main groupings of lines:

1. ***Data Bus***. This is for the transmission of data.

2. ***Address Bus***. This specifies the location of data in MM.

Control Bus. This indicates the direction of data transfer and coordinates the timing of events during the transfer.



Single Bus Structure

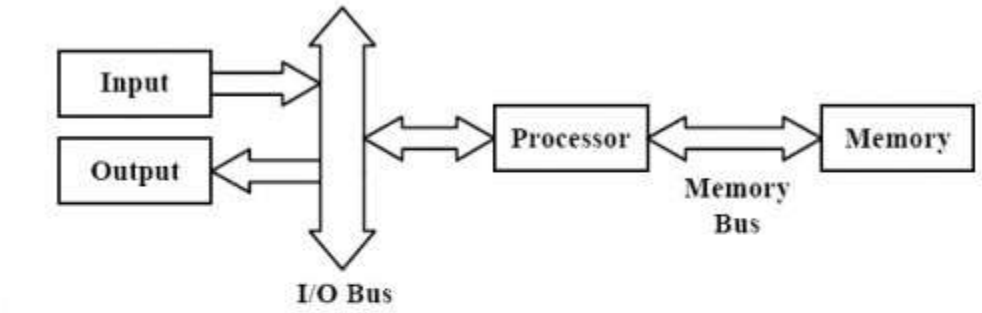
All units are connected to a single bus, so it provides the sole means of interconnection. Single bus structure has advantages of simplicity and low cost.

Single bus structure has disadvantages of limited speed since usually only two units can participate in a data transfer at any one time. This means that an arbitration system is required and that units will be forced to wait.

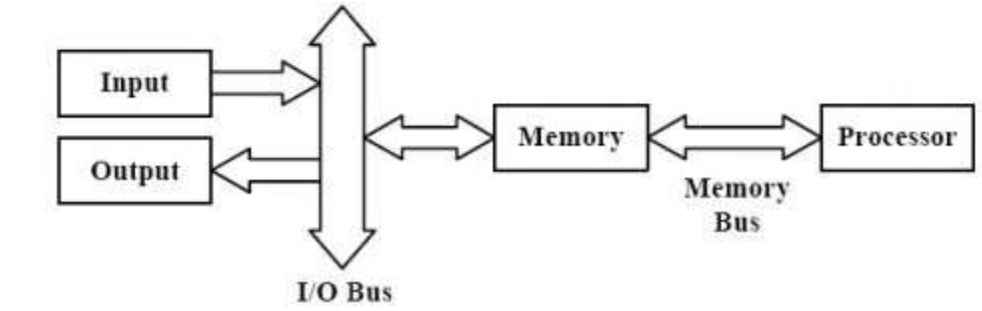
Only two units can actively use the bus at any given time. Bus control lines are used to arbitrate multiple requests for the use of the bus.

Buffer Registers are used to hold information during transfers.

Configuration 1



Configuration 2



Two Bus Structure

In the first configuration, the processor is placed between the I/O unit and the memory unit. The processor is responsible for any data transfer between the I/O unit and the memory unit. The processor acts as a "messenger." In this structure, the processor performance and capability is not being maximized. Most of the time, the processor is doing data transfer between these units instead of performing more complex applications. Also, the processor is idle most of the time waiting for these slow devices.

In the second configuration, I/O transfers are made directly to or from the memory. A special purpose processor called peripheral processor or I/O channel is needed as part of the I/O equipment to control and facilitate such transfers. This special processor is the direct memory access(DMA) controller. It allows main memory to perform data transfer between I/O units.

MICRO-PROGRAMMED VERSUS HARDWIRED CONTROL UNITS:

The Basic Computer

Every student of computer science knows that all traditional digital computers have two principal functional parts: the data path section in which processing occurs and the control section which is responsible for decoding instructions and leaving the correct sequence of control signals to make the processing happen in the data path.. Basically there are two types of control units: hard-wired controllers and micro-programmed controllers. In order to appreciate the difference and see how computers really work, we present a very simple computer. A block diagram of its data path sections is shown in Figure 1.

A single 12-bit-wide bus provides for exchange of information between pairs of registers within the data path section. The registers and the 256 X 12 bit RAM memory are controlled by 16 control signals. Most of the registers have Load (L) and Enabled (E) signals. An active L signal to a register causes the contents of the bus to be clocked into that register on the next rising pulse from the system clock. An active E signal enables the tristate outputs of the register, thereby making its contents available to the bus. Therefore, a register transfer from, for example, register A to register B would require active EA and LB control signals.

Processing of data is done by the Arithmetic-Logic-Unit (ALU), a circuit that is capable of adding or subtracting the 12-bit numbers contained in its two input registers: the accumulator (ACC) and register B. The operation performed by the ALU is selected by the Add (A) or Subtract (S) control signals. The accumulator also contains a single flip-flop that is set whenever its contents are negative (i.e., whenever the leading bit is set--meaning a negative 2's complement number). The value of this "negative flag" provides input to the controller/sequencer, and, as we shall see, permits implementation of conditional branching instructions.

The machine's RAM memory is accessed by first placing the 8-bit address in the Memory Address Register (MAR). An active Read (R) control signal to the RAM will then cause the selected word from the RAM to appear in the Memory Data Register (MDR). An active Write (W) signal, on the other hand, will cause the word contained in the MDR to be stored in the RAM at the address specified by the MAR. Since there are no input or output ports in this simple computer, all I/O is memory mapped. In other words, several memory locations are reserved for input/output devices. Memory reads from any of those locations will cause data from the corresponding input device to appear in the MDR; memory writes to them will cause data in the MDR to be sent to the corresponding output device. A word stored in any given memory location may

be data to be manipulated by the computer or a coded instruction that specifies an action to be taken.

The data path section also contains a Program Counter (PC) whose function it is to point to the address in RAM of the next instruction to be executed. The Increment Program Counter (IP) control signal causes the contents of the PC to increase by one. Since, as we shall see, instructions on this machine are one word long, this provides a simple mechanism for sequential instruction execution. In addition there is an Instruction Register (IR) which holds the instruction that is about to be execute and provides its opcode to the controller/sequencer.

The Computer's Instruction Set

An instruction on our simple computer consists of one 12-bit word. The leading four bits form the operation code (opcode) which specifies the action to be taken, and the remaining 8 bits, when used, indicate the memory address of one of the instruction's operands. For those instructions that have two operands, the other operand is always contained within the accumulator.

Table 1 gives eight instructions that form the instruction set we have chosen for our machine. Also shown in the table is the sequence of control signals necessary for execution of each of the instructions in the machine's instruction set and for fetching the next instruction. In each case the register transfers required for execution of each step are shown. For example, in the case of the LDA (load accumulator) instruction, the first step consists of copying the address of the operand, contained in the least significant 8 bits of the instruction register, to the memory address register. Thus the EI (enable IR) and LM (load MAR) control signals are active. The next step is to read the operand from memory into the memory data register. An active R (memory read) signal performs that task. The last step required to execute the LDA instruction is to copy the contents of the memory data register to the accumulator. Active ED (enable MDR) and LA (load accumulator) do the trick.

The Hard-Wired Control Unit

Figure 2 is a block diagram showing the internal organization of a hard-wired control unit for our simple computer. Input to the controller consists of the 4-bit opcode of the instruction currently contained in the Instruction Register and the negative flag from the accumulator. The controller's output is a set of 16 control signals that go out to the various registers and to the memory of the computer, in addition to a HLT signal that is activated whenever the leading bit of the op-code is one. The controller is

composed of the following functional units: A ring counter, an instruction decoder, and a control matrix.

The ring counter provides a sequence of six consecutive active signals that cycle continuously. Synchronized by the system clock, the ring counter first activates its T0 line, then its T1 line, and so forth. After T5 is active, the sequence begins again with T0. Figure 3 shows how the ring counter might be organized internally.

The instruction decoder takes its four-bit input from the op-code field of the instruction register and activates one and only one of its 8 output lines. Each line corresponds to one of the instructions in the computer's instruction set. Figure 4 shows the internal organization of this decoder.

The most important part of the hard-wired controller is the control matrix. It receives input from the ring counter and the instruction decoder and provides the proper sequence of control signals. Figure 5 is a diagram of how the control matrix for our simple machine might be wired. To understand how this diagram was obtained, we must look carefully at the machine's instruction set (Table 1). Table 2 shows which control signals must be active at each ring counter pulse for each of the instructions in the computer's instruction set (and for the instruction fetch operation). The table was prepared by simply writing down the instructions in the left-hand column. (In the circuit these will be the output lines from the decoder). The various control signals are placed horizontally along the top of the table. Entries into the table consist of the moments (ring counter pulses T0, T1, T2, T3, T4, or T5) at which each control signal must be active in order to have the instruction executed. This table is prepared very easily by reading off the information for each instruction given in Table 1. For example, the Fetch operation has the EP and LM control signals active at ring count 1, and ED, LI, and IPC active at ring count 2. Therefore the first row (Fetch) of Table 2 has T0 entered below EP and LM, T1 below R, and T2 below IP, ED, and LI.

Once Table 2 has been prepared, the logic required for each control signal is easily obtained. For each an AND operation is performed between any active ring counter (Ti) signals that were entered into the signal's column and the corresponding instruction contained in the far left-hand column. If a column has more than one entry, the output of the ANDs are ORed together to produce the final control signal. For example, the LM column has the following entries: T0 (Fetch), T3 associated with the LDA instruction, and T3 associated with the STA instruction. Therefore, the logic for this signal is:

LM = T0 + T3*LDA + T3*STA

This means that control signal LM will be activated whenever any of the following conditions is satisfied: (1) ring pulse T0 (first step of an instruction fetch) is active, or (2) an LDA instruction is in the IR and the ring counter is issuing pulse 3, or (3) and STA instruction is in the IR and the ring counter is issuing pulse 3.

The entries in the JN (Jump Negative) row of this table require some further explanation. The LP and EI signals are active during T3 for this instruction if and only if the accumulator's negative flag has been set. Therefore the entries that appear above these signals for the JN instruction are T3*NF, meaning that the state of the negative flag must be ANDed in for the LP and EI control signals.

Figure 6 gives the logical equations required for each of the control signals used on our machine. These equations have been read from Table 2, as explained above. The circuit diagram of the control matrix (Figure 5) is constructed directly from these equations.

It should be noticed that the HLT line from the instruction decoder does not enter the control matrix, Instead this signal goes directly to circuitry (not shown) that will stop the clock and thus terminate execution.

A Micro-programmed Control Unit

As we have seen, the controller causes instructions to be executed by issuing a specific set of control signals at each beat of the system clock. Each set of control signals issued causes one basic operation (micro-operation), such as a register transfer, to occur within the data path section of the computer. In the case of a hard-wired control unit the control matrix is responsible for sending out the required sequence of signals.

An alternative way of generating the control signals is that of micro-programmed control. In order to understand this method it is convenient to think of sets of control signals that cause specific micro-operations to occur as being "microinstructions" that could be stored in a memory. Each bit of a microinstruction might correspond to one control signal. If the bit is set it means that the control signal will be active; if cleared the signal will be inactive. Sequences of microinstructions could be stored in an internal "control" memory. Execution of a machine language instruction could then be caused by fetching the proper sequence of microinstructions from the control memory and sending them out to the data path section of the computer. A sequence of microinstructions that implements an instruction on the external computer is known as a micro-routine. The instruction set of the computer is thus determined by the set of micro-routines, the"microprogram," stored in the controller's memory. The control

unit of a microprogram-controlled computer is essentially a computer within a computer.

Figure 7 is a block diagram of a micro-programmed control unit that may be used to implement the instruction set of the computer we described above. The heart of the controller is the control 32 X 24 ROM memory in which upt to 32 24-bit long microinstructions can be stored. Each is composed of two main fields: a 16-bit wide control signal field and an 8-bit wide next-address field. Each bit in the control signal field corresponds to one of the control signals discussed above. The next-address field contains bits that determine the address of the next microinstruction to be fetched from the control ROM. We shall see the details of how these bits work shortly. Words selected from the control ROM feed the microinstruction register. This 24-bit wide register is analogous to the outer machine's instruction register. Specifically, the leading 16 bits (the control-signal field) of the microinstruction register are connected to the control-signal lines that go to the various components of the external machine's data path section.

Addresses provided to the control ROM come from a micro-counter register, which is analogous to the external machine's program counter. The micro-counter, in turn, receives its input from a multiplexer which selects from : (1) the output of an address ROM, (2) a current-address incrementer, or (3) the address stored in the next-address field of the current microinstruction. The logic that selects one of these three alternatives will be explained shortly.

The controller's address ROM is fed by the outer computer's instruction register. The address ROM maps the op-code of the instruction currently contained in the op-code field of the instruction register to the starting address of the corresponding microroutine in the control ROM. Address zero of the address ROM contains the control-ROM address of the fetch routine; each other addresses in the address-ROM corresponds to one of the op-codes of the computer's instruction set. Table 3 shows the contents of the address ROM for the instruction set of our simple computer. To see how the address ROM works, let us assume that an ADD instruction has been fetched into the outer computer's instruction register. Since the op-code of the ADD instruction is 3, the number stored at location 3 of the address ROM (a 9) is the starting address in the control ROM of the microroutine that implements the ADD instruction.

Details of a microinstruction's next address field are shown in Figure 8. The 5-bit CRJA (Control ROM Jump Address) sub-field holds a microinstruction address. Thus, the address of the next microinstruction may be obtained from the current microinstruction. This permits branching to other sections within the microprogram.

The combination of the MAP bit, the CD (condition) bit, and the negative flag from the accumulator of the external machine provide input to the logic that feeds the select lines of the multiplexer and thereby determine how the address of the next microinstruction will be obtained.

If the MAP bit is one, the logic attached to the multiplexer's select lines produces a 01 which selects the address ROM. Therefore, the address of the micro-routine corresponding to the instruction in the outer machine's instruction register will be channeled to the control ROM. It should be clear that the MAP bit must be set in the last microinstruction of the "fetch" micro-routine, since it is at that moment that we want the newly-fetched instruction to be executed.

If the MAP bit is zero and the CD bit is zero, (unconditional branch), the multiplexer logic produces a 10, which selects the CRJA field of the current instruction. Therefore, the next instruction will come from the address contained in the current instruction's next-address field. With MAP=0 and CD=1 (conditional branch), the logic that feeds the multiplexer will produce either a 00 or a 10, depending on the value of the negative flag. If the flag is set, it is a 10, which selects the jump address contained in the current microinstruction. If the negative flag is cleared, the select lines to the multiplexer receive a 00, which causes the incrementer to be selected. The next microinstruction will come from the next address in sequence. It should be noticed that with this scheme, if we are not doing branching, the CRJA field should contain the address of the next microinstruction and the CD bit should be cleared. This will cause "branch to the next microinstruction" to occur. The one exception to this rule is the case of the last microinstruction within a micro-routine. Normally we would then want to branch back to the "fetch" micro-routine. Since this routine starts at control-ROM location 00000, that address should be contained in the CRJA field and CD should be 0.

The HLT bit is used to terminate execution. If it is set, the clock that synchronizes activities within the entire machine is stopped.

Notice that the micro-counter is triggered by a rising clock edge, and the microinstruction register by a falling edge. Thus, we see that on each positive edge, the micro-counter receives the address of the microinstruction and presents it to the control ROM, which has until the next negative edge to output the addressed control word to the microinstruction register. Since all operation in the data path section are positive-edge triggered, there is adequate time for the signals specified in the control word contained in the microinstruction register to go out to all sections of the external machine. The sequence of latching the address of microinstruction i+1 into the micro-counter while microinstruction i executes (positive edge) and then presenting the

control word of microinstruction i+1 to the microinstruction register (negative edge) continues until a set HLT bit stops the clock.

Table 4 shows a microprogram which, when loaded into the control ROM, will implement the instruction set of the computer we have been describing. For each microinstruction, the control ROM address has been expressed in hexadecimal, and the contents in binary. The order of the bits in the control signal field is the same as that shown in table 2: IP, LP, EP, LM, R, W, LD, ED, LI, EI, LA, EA, A, S, EU, LB, reading from left to right. The last four columns of table 4 express the status of the CD, MAP, and HLT bits, and the Control ROM Jump Address, expressed in hexadecimal. In order to clarify how the microprogram works, a description is now given of the "fetch" and JN (jump on negative) micro-routines.

The "fetch" micro-routine occupies control ROM addresses 0, 1, and 2. The active EP and LM control-signal bits in its first microinstruction cause a register transfer from the program counter to the memory address register to occur. The MAR will now contain the address in RAM of the next instruction. Since CD and MAP are both zero (unconditional branch), the next microinstruction will come from the address stored in the CRJA field (01) -- the next consecutive location. The microinstruction stored at that location has only the R bit active. Thus, the word stored in the memory location being accessed by the MAR (presumably the next instruction) will be gated to the Memory Data Register (MDR). The zeroes in CD and MAP again cause the microinstruction to be fetched from the address specified in the CRJA field, (02). Active control signal bits for that microinstruction are ED, LI, and IP. The first two transfer the word in the MDR to the Instruction Register, and the last increments the program counter. The new instruction is safely in the IR, and the PC is pointing to the next instruction in sequence. We have completed an instruction fetch. Since the MAP field in the last microinstruction of this "fetch" micro-routine is equal to 1, the address of the next microinstruction is determined by the address ROM, which, in turn, depends upon the opcode of the instruction that has just been loaded into the instruction register.

When the JN instruction is executed, control is supposed to be transferred to the address specified by the least significant eight bits of the number contained in the instruction register if the negative flag is set. If the negative flag is not set, execution should continue with the next instruction in sequence. Let us see how the micro-routine stored at control-ROM locations 0F, 10, and 11 implement this conditional jump. In the first microinstruction, none of the control signal bits is set. Thus, nothing will occur in the data path section of the computer. However, the fact that the CD bit is set means that IF THE NEGATIVE FLAG IS SET, the next microinstruction will be fetched from the control-ROM address specified in the CRJA field (11 in this

case). The microinstruction stored at that location has the EI and LP control signal bits set. Thus, the contents of the instruction register (the least significant eight bits) will be transferred to the program counter. The zeroes stored in the CD and MAP bits cause the next microinstruction to be fetched from the address contained in the CRJA field -- a 00 in this case. This is the start of the "fetch"micro-routine. Thus we see that if the negative flag is set, the JN micro-routine places the jump address in the program counter and transfer to the fetch routine. When that fetch is performed, control will have been transferred to the jump address.

If, on the other hand, the negative flag is NOT SET when the JN micro-routine executes, then the set CD bit in its first microinstruction causes the current address stored in the micro-counter to be incremented. Thus, the next microinstruction would be fetched from location 10. That microinstruction also has no active control signals bits, but with CD=0 and CRJA=00, the next microinstruction will be the first one in the "fetch" routine. Notice that in this case, the JN instruction simply returns us to the next fetch. Since the program counter has not been altered, that fetch will be from the next sequential memory location, as usual.

Hardwired vs. Micro-programmed Computers

It should be mentioned that most computers today are micro-programmed. The reason is basically one of flexibility. Once the control unit of a hard-wired computer is designed and built, it is virtually impossible to alter its architecture and instruction set. In the case of a micro-programmed computer, however, we can change the computer's instruction set simply by altering the microprogram stored in its control memory. In fact, taking our basic computer as an example, we notice that its four-bit op-code permits up to 16 instructions. Therefore, we could add seven more instructions to the instruction set by simply expanding its microprogram. To do this with the hard-wired version of our computer would require a complete redesign of the controller circuit hardware.

Another advantage to using micro-programmed control is the fact that the task of designing the computer in the first place is simplified. The process of specifying the architecture and instruction set is now one of software (micro-programming) as opposed to hardware design. Nevertheless, for certain applications hard-wired computers are still used. If speed is a consideration, hard-wiring may be required since it is faster to have the hardware issue the required control signals than to have a "program" do it.

FIGURES:

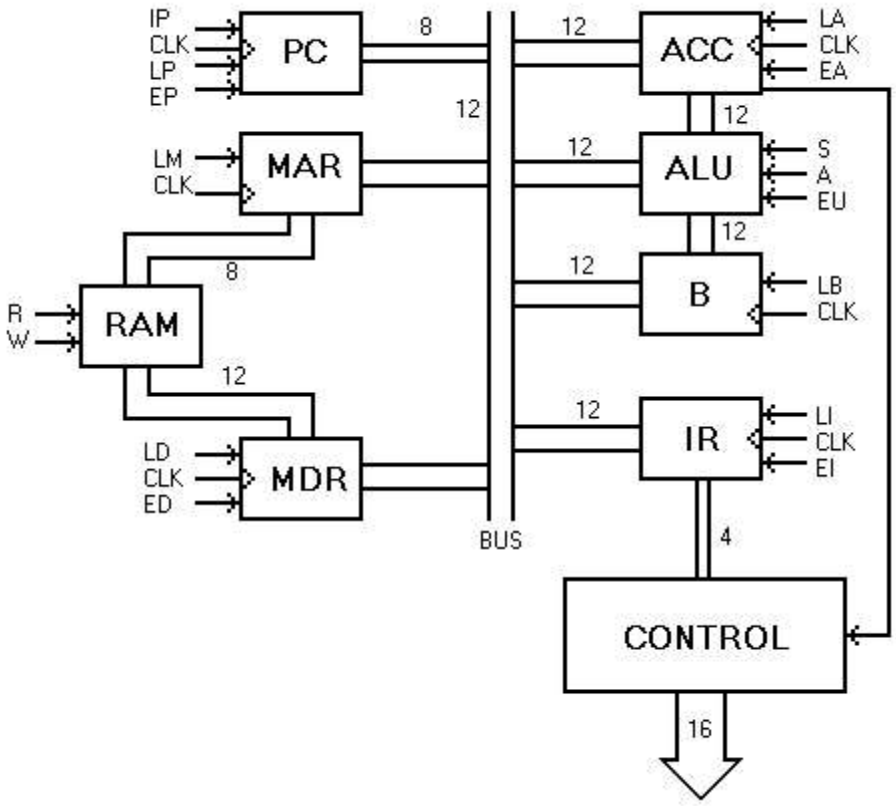


Figure 1. A Simple Single-Bus Basic Computer.

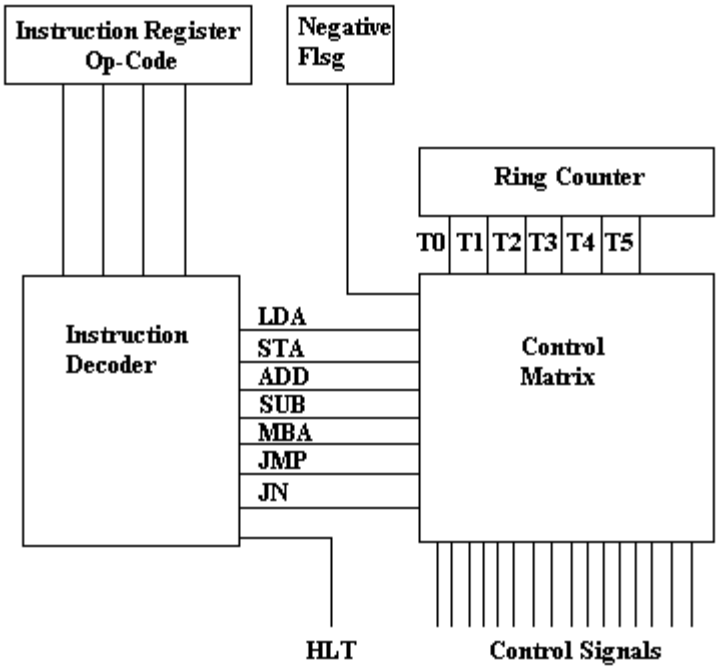


Figure 2. A Block diagram of the Basic Computer's Hard-wired Control unit

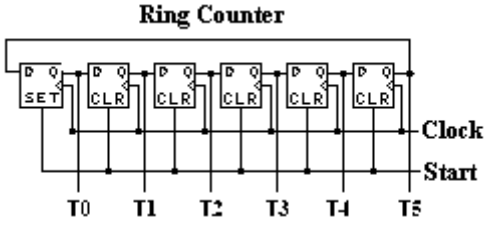


Figure 3. The Internal Organization of the Ring Counter

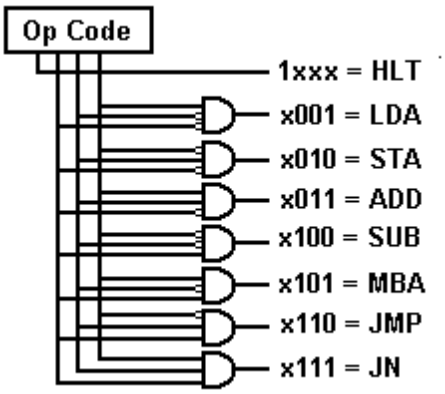


Figure 4. The Internal Organization of the Hard-wired Instruction Decoder

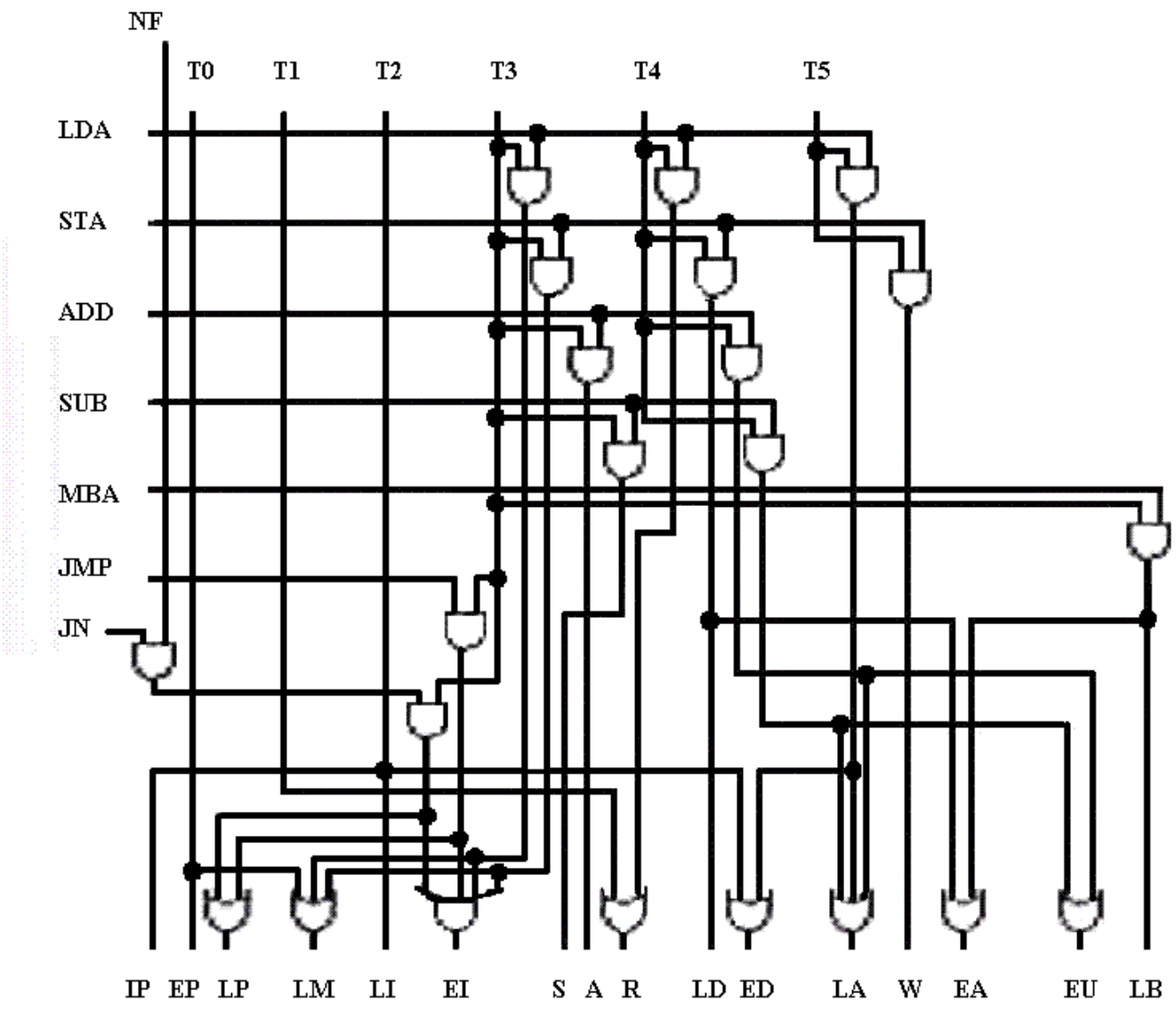


Figure 5. The Internal Organization of the Hard-wired Control Matrix

IP = T2
W = T5*STA
LP = T3*JMP + T3*NF*JN
LD = T4*STA
LA = T5*LDA + T4*ADD + T4*SUB
EA = T4*STA + T3*MBA
EP = T0
S = T3*SUB
A = T3*ADD
LI = T2
LM = T0 + T3*LDA + T3*STA
ED = T2 + T5*LDA
R = T1 + T4*LDA
EU = T3*ADD+T3*SUB
EI = T3*LDA + T3*STA + T3*JMP + T3*NF*JN
LB = T3*MBA

Figure 6. The logical equations required for each of the hardwired control signals on the basic computer. The machine's control matrix is designed from these equations.

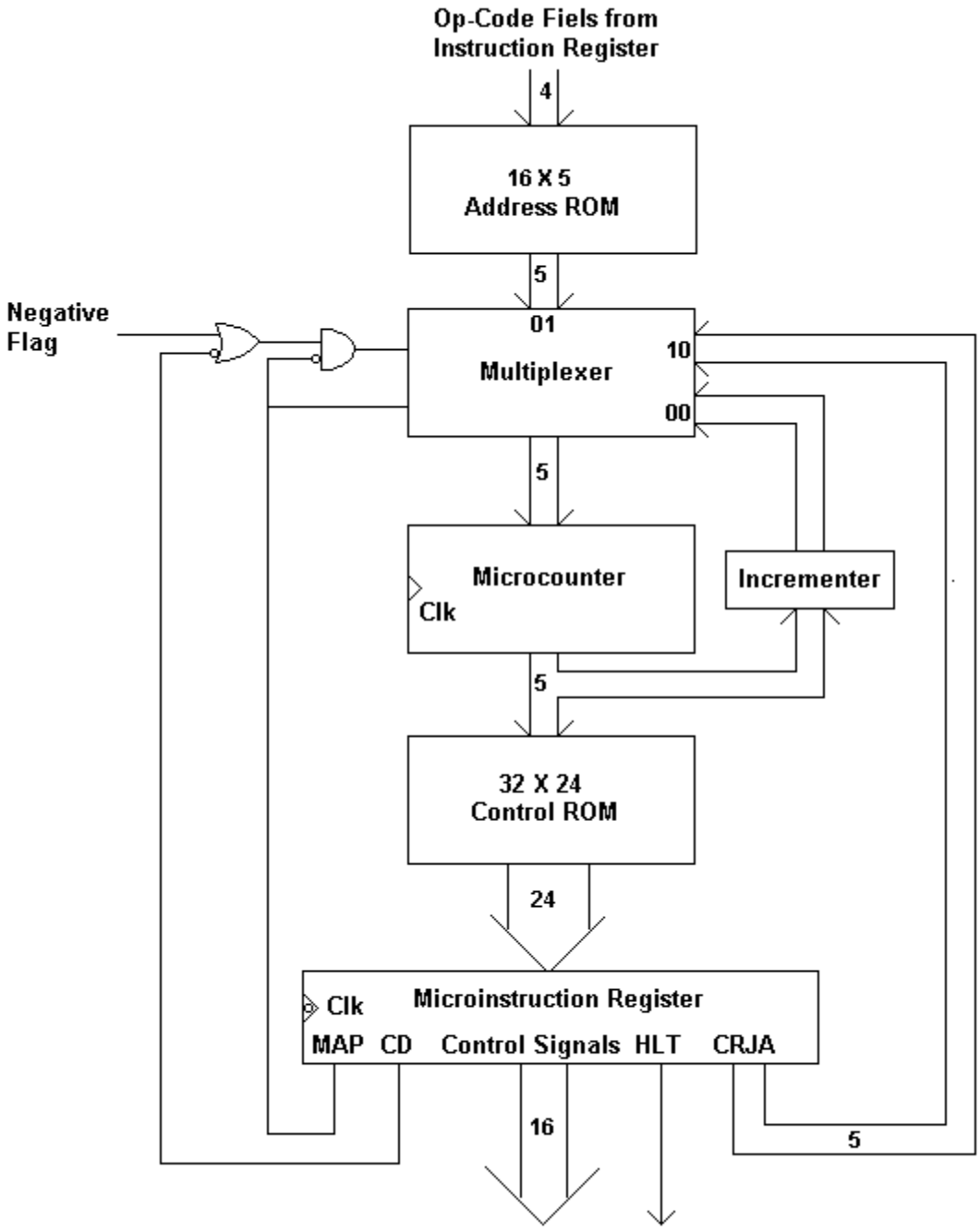


Figure 7. A Microprogrammed Control Unit for the Simple Computer

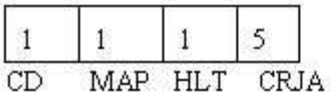


Figure 8. Next address field of the microinstruction register. CD is the condition bit, MAP causes the address of the next microinstruction to be obtained from the address ROM, HLT stops the clock, and CRJA is the control ROM jump address field.

TABLES:

Table 1. An Instruction Set For The Basic Computer

Instruction Code Mnemonic	Op-Execution	Register Action	Ring Transfers		Active Control Pulse	Control Signals
LDA (Load ACC)	1	ACC<--(RAM)	1. MAR <-- IR		3	EI, LM
			2. MDR <-- RAM(MAR)		4	R
			3. ACC <-- MDR		5	ED, LA
STA (Store ACC)	2	(RAM) <--ACC	1. MAR <-- IR		3	EI, LM
			2. MDR <-- ACC		4	EA, LD
			3. RAM(MAR) <-- MDR		5	W
ADD (Add B to ACC)	3	ACC <-- ACC + B	1. ALU <-- ACC + B		3	A
			2. ACC <-- ALU		4	EU, LA
SUB (Sub. B from ACC)	4	ACC <-- ACC - B	1. ALU <-- ACC - B		3	S
			2. ACC <-- ALU		4	EU, LA
MBA (Move ACC to B)	5	B <-- ACC	1. B <-- A		3	EA, LB
JMP (Jump to Address)	6	PC <-- RAM	1. PC <-- IR		3	EI, LP

JN (Jump if Negative)	7	PC <-- RAM if negative flag is set	1. PC <-- IR if NF set	3	NF: EI, LP
HLT	8-15	Stop clock			
"Fetch"		IR <-- Next Instruction	1. MAR <-- PC 2. MDR <-- RAM(MAR) 3. IR <-- MDR	0 1 2	EP, LM R ED, LI, IP

Table 2. A Matrix of Times at which Each Control Signal Must Be Active in Order to Execute the Hard-wired Basic Computer's Instructions

Control Signal: Instruction:	IP	LP	EP	LM	R	W	LD	ED	LI	EI	LA	EA	A	S	EU	LB
"Fetch"	T2		T0	T0	T1			T2	T2							
LDA				T3	T4			T5		T3	T5					
STA				T3		T5	T4			T3		T4				
MBA												T3				T3
ADD											T4		T3		T4	
SUB											T4			T3	T4	
JMP		T3								T3						
JN		T3*NF								T3*NF						

Table 3. The Microprogrammed Basic Computer's Address ROM

Instruction Mnemonic	Address-ROM Address (Instruction Op-Code)	Address-ROM Contents (Control-ROM Micro-Routine Start Address)
"Fetch"	0	00
LDA	1	03
STA	2	06
ADD	3	09
SUB	4	0B

MBA	5	0D
JMP	6	0E
JN	7	0F
Available for New Instructions	8-E	12-1E
HLT	F	1F

Table 4. The Microprogram that Implements the Basic Computer's Instruction Set

Microroutine Name	Address-ROM (Mnemonic)	Micro-Instruction Address (Op-code)	Control Signals: CD ILELRWLELELEASEL	MAP bit	HLT bit	Address of Next Micro-Instruction	Comment
"Fetch"	0	00	0011000000000000	0	0	0	
	01						Next CR Address = 01
		01	0000100000000000	0	0	0	
	02						Next CR Address = 02
		02	1000000110000000	0	1	0	
	xx						Get CR Address from Address ROM
LDA	1	03	0010000001000000	0	0	0	
	04						Nexr CR Address = 04
		04	0000100000000000	0	0	0	
	05						Next CR Address = 05
		05	0000000100100000	0	0	0	
	00						Next CR Address = 00 (Fetch)
STA	2	06	0010000001000000	0	0	0	
	07						Next CR Address = 07
		07	0000001000010000	0	0	0	
	08						Next CR Address = 08
		08	0000010000000000	0	0	0	
	00						Next CR Address = 00 (Fetch)
ADD	3	09	0000000000001000	0	0	0	
	0A						Next CR Address = 0A
		0A	0000000000100010	0	0	0	
	00						Next CR Address = 00 (Fetch)

```
SUB             4           0B          0000000000000100    0     0     0
   0C       Next CR Address = 0C
                            0C          0000000000100010    0     0     0
   00       Next CR Address = 00 (Fetch)

MBA             5           0D          0000000000010001    0     0     0
   00       Next CR Address is 00 (Fetch)

JMP             6           0E          0100000001000000    0     0     0
   00       Change PC; next CR Address is 00 (Fetch)

JN              7           0F          0000000000000000    1     0     0
   11       NF=0: INC CRJA; NF=1: Next CR Address = 11
                            10          0000000000000000    0     0     0
   00       Next CR Address = 00 (Fetch)
                            11          0100000001000000    0     0     0
   00       Change PC; next CR Address is 00 (Fetch)

Available for   8-E         12-
1E                                                      New
microinstructions can be added here

HLT             F           1F          0000000000000000    0     0     1
   xx       Stop Clock
```