**Chapter 20: Control Unit Operation**

Table of Contents

[20.1 Micro-Operations 3](#_Toc49000174)

[The Fetch Cycle 4](#_Toc49000175)

[The Indirect Cycle 6](#_Toc49000176)

[The Interrupt Cycle 7](#_Toc49000177)

[The Execute Cycle 8](#_Toc49000178)

[The Instruction Cycle 10](#_Toc49000179)

[20.2 Control of The Processor 11](#_Toc49000180)

[Functional Requirements 11](#_Toc49000181)

[Control Signals 13](#_Toc49000182)

[A Control Signals Examples 15](#_Toc49000183)

[Internal Processor Organization 17](#_Toc49000184)

[20.3 Hardwired Implementation 19](#_Toc49000185)

[Control Unit Inputs 19](#_Toc49000186)

[Control Unit Logic 21](#_Toc49000187)

We have previously learnt that a machine instruction set can help us define the processor, since we will know the effect of each op-code, addressing mode and user-visible register, which tells us what functions the processor must perform. This, along with external interfaces and interrupt handling methods, gives us a list of things needed to specify the function of a processor.

* Operations (Op-Codes)
* Addressing Modes
* Registers
* I/O Module Interfaces
* Memory Module Interfaces
* Interrupts

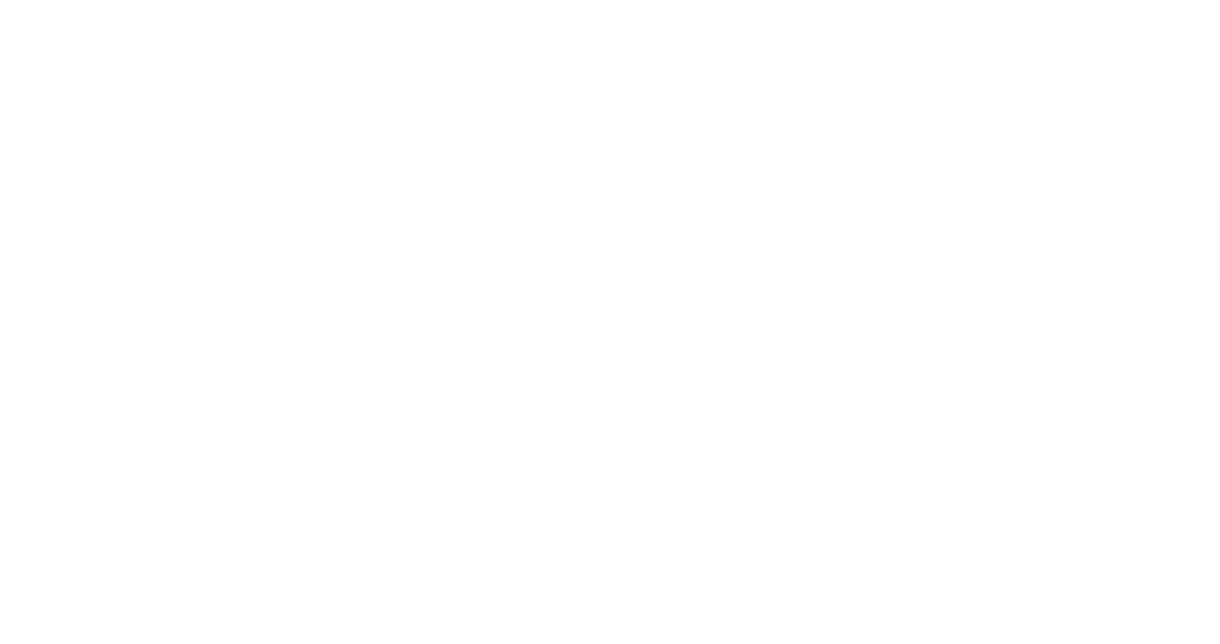
The first three items we get from the instruction set, the fourth and fifth from the system bus definition and the last partially by the system bus and partially by support offered by the processor to the OS. This list can be said to be the functional requirements of a processor, which determines what a processor must do. Now we want to look at how these functions are performed, or rather, how the various elements of the processor are controlled to provide these functions. This is where we bring in the control unit, which controls the operation of the processor.

## 20.1 Micro-Operations

We have seen that a program is executed as a series of machine instructions, each taking up one cycle. Of course, this is not the same as the original written sequence of instructions, as we saw with branching instructions. What we are talking about here is the time sequence of instructions.

We have also seen that each instruction cycle can be divided into smaller units, or stages.

To design a control unit, we need to break down the description further. We saw in Chapter 14 that a further breakdown is possible. In fact, we will see that each stage involves a series of smaller steps, each of which involves the processor registers. These steps are called micro-operations, named so since the steps are very simple and accomplish very little.



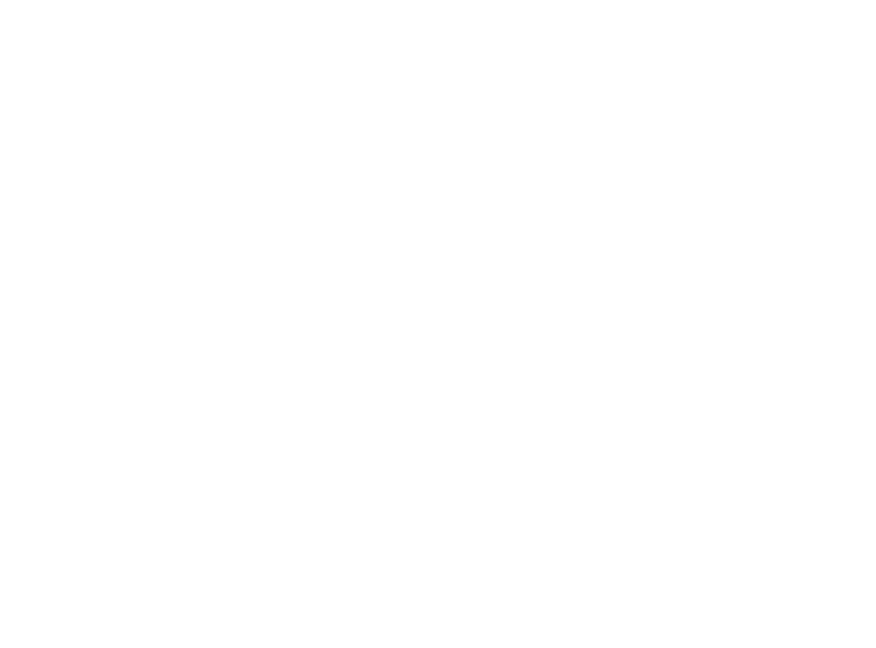
Micro-operations are the functional, or atomic operations of a processor. In this chapter, we will first see how a single instruction cycle can be broken down into micro-operations, and then see how micro-operations serve as a guide to the design of the control unit.

### The Fetch Cycle

In the fetch cycle, four registers get involved.

* **Memory Address Register (MAR)**: This is connected to the address lines of the system bus. It specifies the address in memory for a read or write operation.
* **Memory Buffer Register (MBR)**: This is connected to the data lines of the system bus. It contains the value to be stored in memory or the last value read from memory.
* **Program Counter (PC)**: This holds the address of the next instruction.
* **Instruction Register (IR)**: This holds the last (current) instruction.

Consider this example.



First, we get the address of the next instruction in the PC. That is moved to the MAR. From there it is moved to the address bus. The control unit issues a READ command on the control bus, and the results appear on the data bus. This copied into the MBR. We also need to increment the PC, which we can do simultaneously since it does not interfere with our READ operation. Finally, the contents of the MBR are moved to the IR, freeing up the MBR for use during a possible indirect cycle.

Thus, one fetch cycle has three steps and four micro-operations, each of which involves moving data into or out of a register. As long as they do not interfere with one another, we can perform them simultaneously.

: MAR 🡨 (PC)

: MBR 🡨 Memory

PC 🡨 (PC) + I (I is the instruction length)

: IR 🡨 (MBR)

Here, we have assumed that the clock is available for timing purposes and that it emits regularly spaced clock pulses, each of which defines a time unit. Thus, each time unit is equal and each micro-operation is performed in a single time unit.

Notice how the third micro-operation, which increments the PC, takes place along with the second. We could also have grouped it with the fourth. The grouping of micro-operations follows two simple rules:

1. The proper sequence of events must be followed. For example, the second micro-operation cannot be performed before the first since the second micro-operation makes use of the value obtained by the MAR after the first micro-operation.
2. Conflicts should be avoided, i.e. we should not try to read and write from the same register in the same time unit since it would cause unpredictable results. For example, the second and forth micro-operations must be in different time units, since both use the MBR.

Another point to consider is that one of the micro-operations involves addition. We can use the ALU for this. The ALU might also be involved in other micro-operations, depending on its functionality and the processor’s organization. This will be discussed in more detail later on.

### The Indirect Cycle

The next step is to get the source operands. If the instruction specified an indirect address, an indirect cycle is needed.

: MAR 🡨 (IR(Address))

: MBR 🡨 Memory

: 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 has a direct address instead of an indirect one. The IR is now in the same state it would be in if we had not used indirect addressing at all.

### The Interrupt Cycle

When the execute cycle is finished, we make a test to check if any enabled interrupts occurred. If they have, the interrupt cycle begins. The nature of this cycle varies from machine to machine. A simple sequence is given here.

: MBR 🡨 (PC)

: MAR 🡨 Save Address

PC 🡨 Routine Address

: Memory 🡨 (MBR)

First, we put the contents of the PC in the MBR so they can be saved to let us return from the interrupt. Then the MAR is loaded with the address of the location where the contents of the PC are to be saved, and the PC is loaded with the address of the start if the interrupt routine. These two could be separate micro-operations as well, but are most likely not since most processors have multiple types of interrupts which can take a few micro-operations to obtain. Finally, the contents of the MBR, which has the old contents of the PC, is put in memory.

### The Execute Cycle

The first three cycles we saw were small and fixed. The same micro-operations are always executed. The execute cycle on the other hand, has to deal with a variety of op-codes which means there are many different sequences of micro-operations that can occur. The control unit examines the op-code and generates a sequence of micro-operations based on it. This is called instruction decoding.

ADD R1, X - Add contents of location X to register R1

: MAR 🡨 (IR(Address))

: MBR 🡨 Memory

: R1 🡨 (R1) + (MBR)

First, the address from the IR is loaded to the MAR. The referenced location is read and the contents are put in the MBR. Finally, the ALU adds the contents of R1 and the MBR. Of course, this is a simplified view. More micro-operations could be needed to extract the register reference from the IR and maybe to stage the ALU inputs and outputs in some intermediate registers.

ISZ X - Increment contents of location X. If the result is 0, skip the next instruction.

: MAR 🡨 (IR(Address))

: MBR 🡨 Memory

: MBR 🡨 (MBR) + 1

: Memory 🡨 (MBR)

If ((MBR) = 0) then (PC 🡨 (PC) + I)

The new feature here is the conditional action. The test and the increment can be performed as one micro-operation. Also, note that this can be done in the same time unit in which the updated MBR value is stored back in memory.

BSA X - The address of the location immediate after the BSA instruction is stored in location X, and execution continues at location X + 1. The saved address will be used to return. This technique is used to support subroutine calls.

: MAR 🡨 (IR(Address))

MBR 🡨 (PC)

: PC 🡨 (IR(Address))

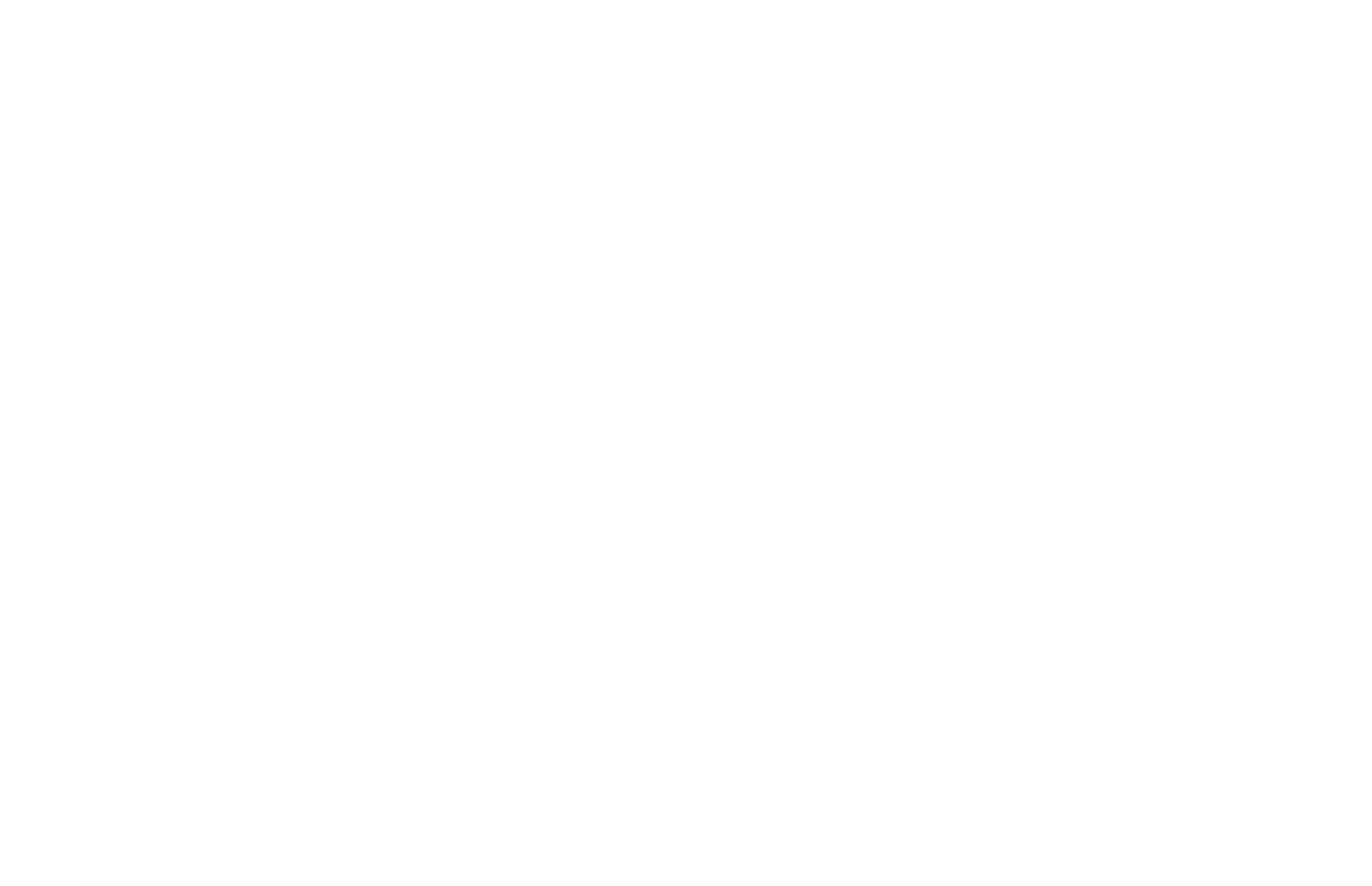
Memory 🡨 (MBR)

: PC 🡨 (PC) + I

The address of the PC is originally the address of the next sequential instruction. This is saved at the address designated by the IR. The IR is also incremented to provide the address of the next instruction.

### The Instruction Cycle

We still need to tie together the sequence of micro-operations we have seen so far. To do this, we assume a new 2-bit register called the instruction cycle code (ICC). The ICC designates the state of the processor in terms of what part of the cycle it is in, fetch indirect, execute and interrupt, represented as 00, 01, 10 and 11 respectively.



At the end of each of the four cycles, the ICC is set appropriately. The indirect cycle is always followed by the execute cycle and the interrupt cycle is always followed by the fetch cycle. For both the fetch cycle and the execute cycle, the next cycle depends on the state of the system.

## 20.2 Control of The Processor

### Functional Requirements

Now that we have reduced instructions to their most fundamental level, we can define exactly what the control unit must make happen. Thus, we will look at the functional requirements of the control unit, those functions which it must perform. This is of course the basis for the design and implementation of the control unit.

The steps we must follow are:

1. Define the basic elements of the processor.
2. Describe the micro-operations that the processor performs.
3. Determine the functions that the control-unit must perform to cause the micro-operations be performed.

Steps 1 and 2 are already complete.

The basic functional elements of the processor are:

* ALU
* Registers
* Internal Data Paths
* External Data Paths
* Control Unit

This is a complete list. The ALU is the functional essence of the computer. Registers are used to store data internal to the processor. Internal data paths move between registers and between registers and the ALU. External data paths link registers to memory and I/O modules via the system bus. The control unit causes operations to happen inside the processor.

All micro-operations fall into one of the following categories:

* Transfer data from one register to another.
* Transfer data from a register to an external interface (e.g. the system bus).
* Transfer data from an external interface to a register.
* Perform an arithmetic or logic operation, using registers for input and output.

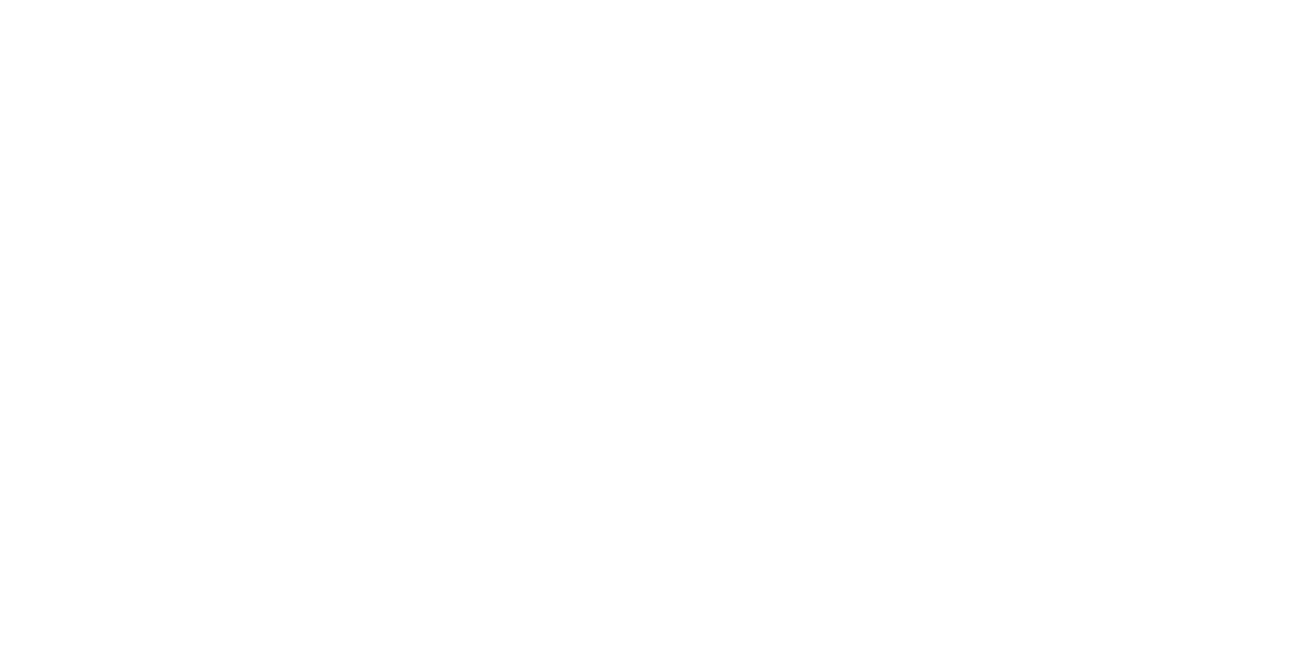
The control unit performs two basic tasks:

* **Sequencing**: Make the processor step through a series of micro-operations in a proper sequence, based on the program being executed.
* **Execution**: Make each micro-operation be performed.

The control unit meets this functional description using control signals.

### Control Signals

For the control unit to perform its function, it needs inputs to see the state of the system and outputs to control the behaviour of the system. Internally, it must have the logic needed to perform its sequencing and execution functions. This section will concentrate on the interaction between the control unit and the other elements of the processor.



The inputs are:

* **Clock**: This is how the control unit keeps time. It causes one micro-operation (or a set of simultaneous micro-operations) to be performed for each clock pulse. This is called the processor cycle time or the clock cycle time.
* **Instruction Register**: The op-code and addressing mode of the current instruction are used to determine what micro-operations must be performed during the execute cycle.
* **Flags**: These are used to determine the status of the processor and the outcome of previous ALU operations.
* **Control Signals from Control Bus**

The outputs are:

* **Control Signals Within the Processor**: These are of two types, ones that cause data to be moved from one register to another, and ones that activate specific ALU functions.
* **Control Signals to Control Bus**: These are of two types, control signals to memory and control signals to I/O modules.

Three types of control signals are used: ones that activate an ALU function, ones that activate a data path and ones that are signals on the external system bus or some other external interface. Ultimately, all these signals are binary inputs to individual logic gates.

To see how the control unit maintains control, consider the fetch cycle again. The control unit knows where it is in the instruction cycle, so it knows when the fetch cycle must be performed. The first step is to transfer the contents of the PC to the MAR. The control unit activates the control signal that opens the gates between the bits of the PC and the bits of the MAR. Next, a word must be read from memory into the MBR and the PC must be incremented. This requires a few signals to be sent together:

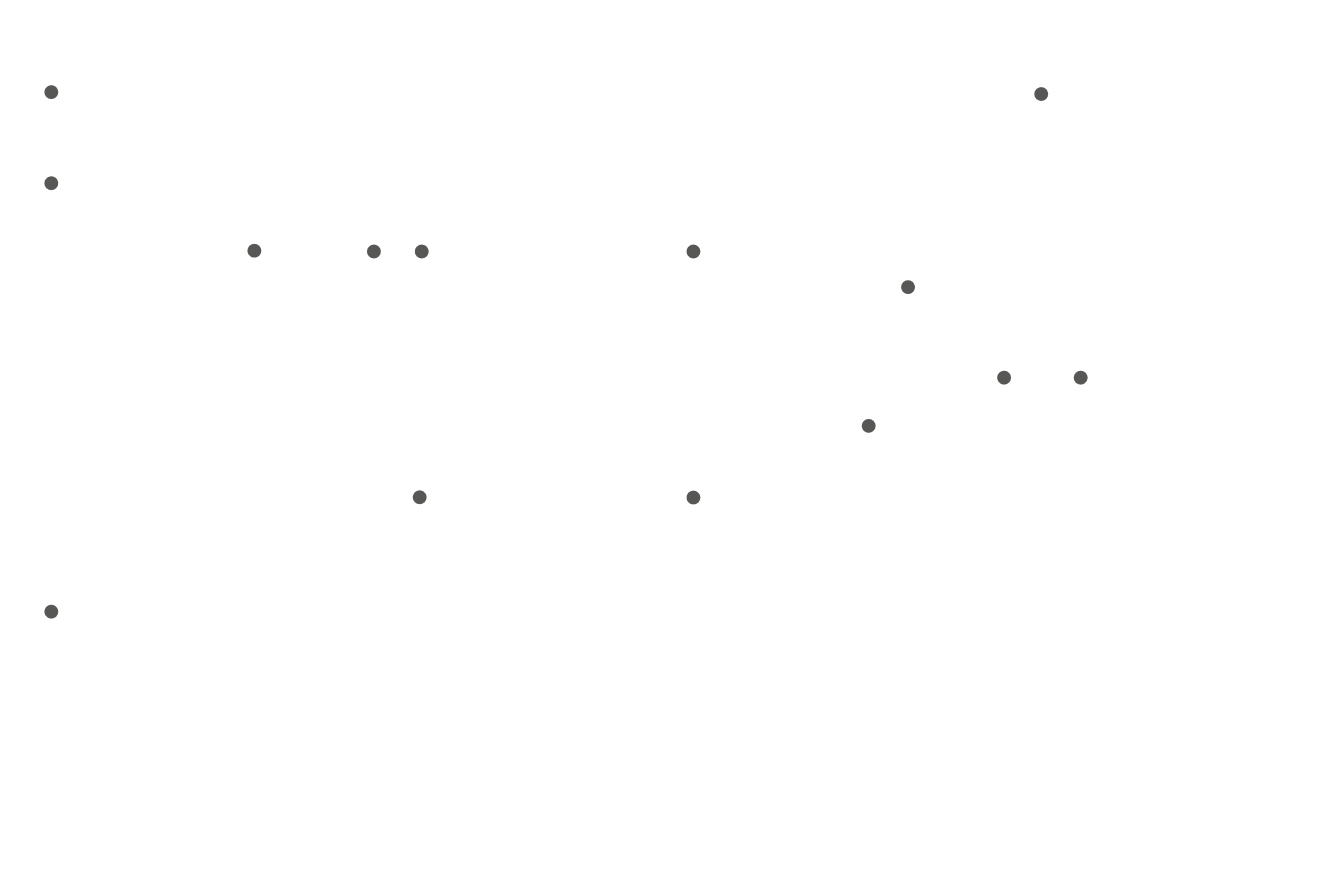
* A control signal that opens gates allowing the contents of the MAR onto the address bus
* A memory read control signal on the control bus
* A control signal that opens gates allowing contents of the data bus to be stored in the MBR
* Control signals to logic that adds 1 to the contents of the PC and stores the result back on the PC

Finally, a control signal is sent to open the gates between the MBR and the IR.

The control unit must also decide whether to perform an indirect cycle or an execute cycle next, so it must check the IR to see if an indirect memory reference is made.

The indirect and interrupt cycles work similarly. For the execute cycle, the control unit examines the op-code first and decides which micro-operations to perform based on that.

### A Control Signals Example



This is a simple processor with a single AC. The data paths are shown but the control paths are not. Terminations for control signals are indicated as and shown with a circle. They indicate gates that, when open, allow data flow between places.

The control unit gets inputs from the clock, the IR and flags. At each clock cycle, the control unit reads all its inputs and emits a set of control signals.

A few micro-operations and their required control signals are listed in the table below. For simplicity, we are ignoring the data and control paths for incrementing the PC and for loading the fixed addresses into the PC and MAR.

|  |  |  |
| --- | --- | --- |
|  | **Micro-Operations** | **Active Control Signals** |
| Fetch | : MAR 🡨 (PC) |  |
| : MBR 🡨 Memory  PC 🡨 (PC) + 1 | , |
| : IR 🡨 (MBR) |  |
| Indirect | : MAR 🡨 (IR(Address)) |  |
| : MBR 🡨 Memory | , |
| : IR(Address) 🡨 (MBR(Address)) |  |
| Interrupt | : MBR 🡨 (PC) |  |
| : MAR 🡨 Save Address  PC 🡨 Routine Address |  |
| : Memory 🡨 (MBR) | , |

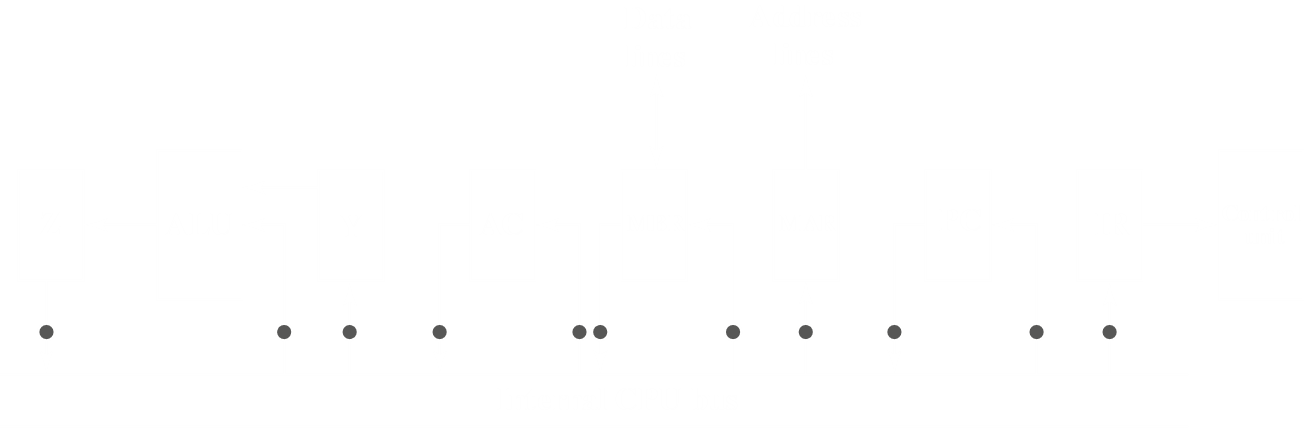
: Read control signal to system bus

: Write control signal to system bus

Notice how minimal the nature of the control unit is. It runs the entire computer based only on the knowledge of instructions to be executed and the nature of the results of arithmetic and logical operations (e.g. positive, negative, overflow, etc.). It never sees the data being processed or the actual results. And all of this it does with just a few control signals to points within the processor and to the system bus.

### Internal Processor Organization

Consider the diagram for the example we just saw. It is very complex. We could instead, use an internal bus to connect the ALU and all the processor registers, leaving just the various gates and control signals.



Note that control signals to the external system bus and to control the ALU are also present, but not shown.

Notice that two registers, and , were added. We need these for the ALU. Register is needed in case there are two input operands, since the ALU has no internal storage to store an extra operand. We could have used the AC to do this, but that limits the flexibility of the system and would not work if we had multiple general-purpose registers. Register is needed to store the output of the ALU before sending it to the bus, since the ALU is a combinational circuit and feeding the output directly to the bus would cause it to be fed back to the input.

Under this arrangement, an operation to add a value from memory to the AC would be:

: MAR 🡨 (IR(Address))

: MBR 🡨 Memory

: Y 🡨 (MBR)

: Z 🡨 (AC) + (Y)

: AC 🡨 (Z)

Other organizations are possible, but an internal bus or a set of internal buses is generally used. It simplifies the interconnection layout and the control of the processor. It also saves space.

===================== Skipping Subsection: The Intel 8085 =====================

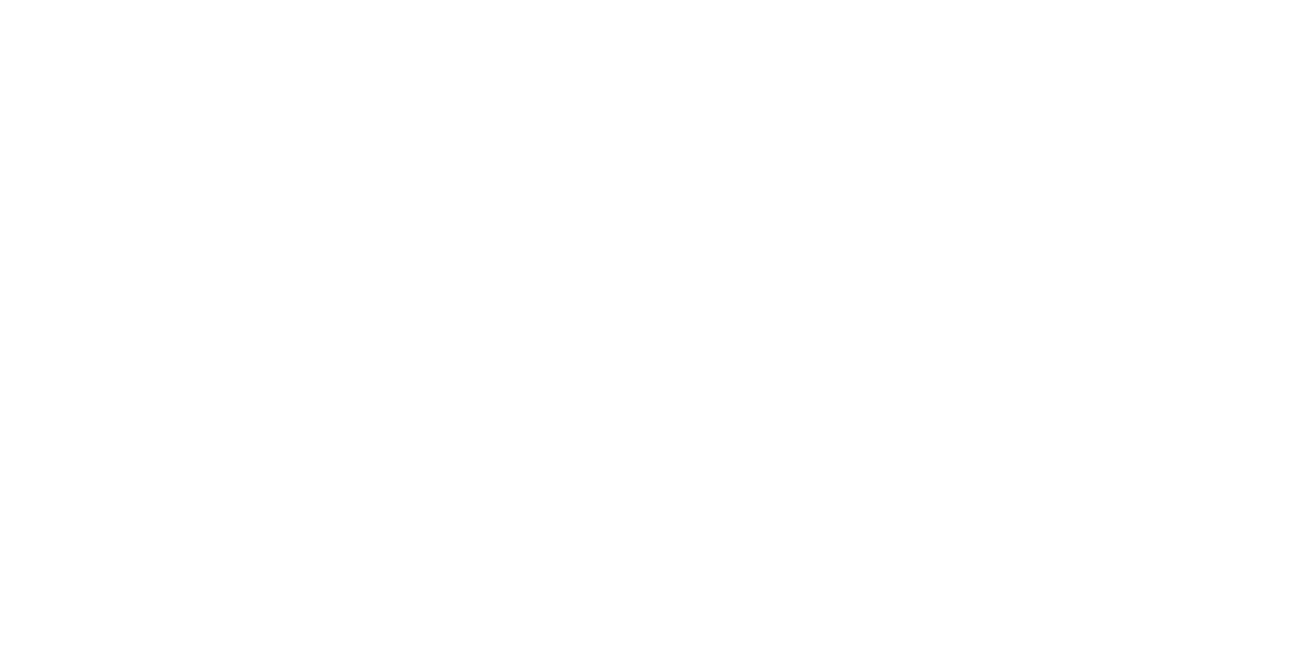
## 20.3 Hardwired Implementation

Now we will study the actual implementation of the control unit. Many techniques are used, but most fall into one of two categories:

* Hardwired Implementation
* Microprogrammed Implementation

In hardwired implementation, the control unit is essentially a state machine circuit. The input logic signals are transformed into output logic signals, which are the control signals. We shall be studying this approach in this section. The microprogrammed implementation will not be studied here.

### Control Unit Inputs



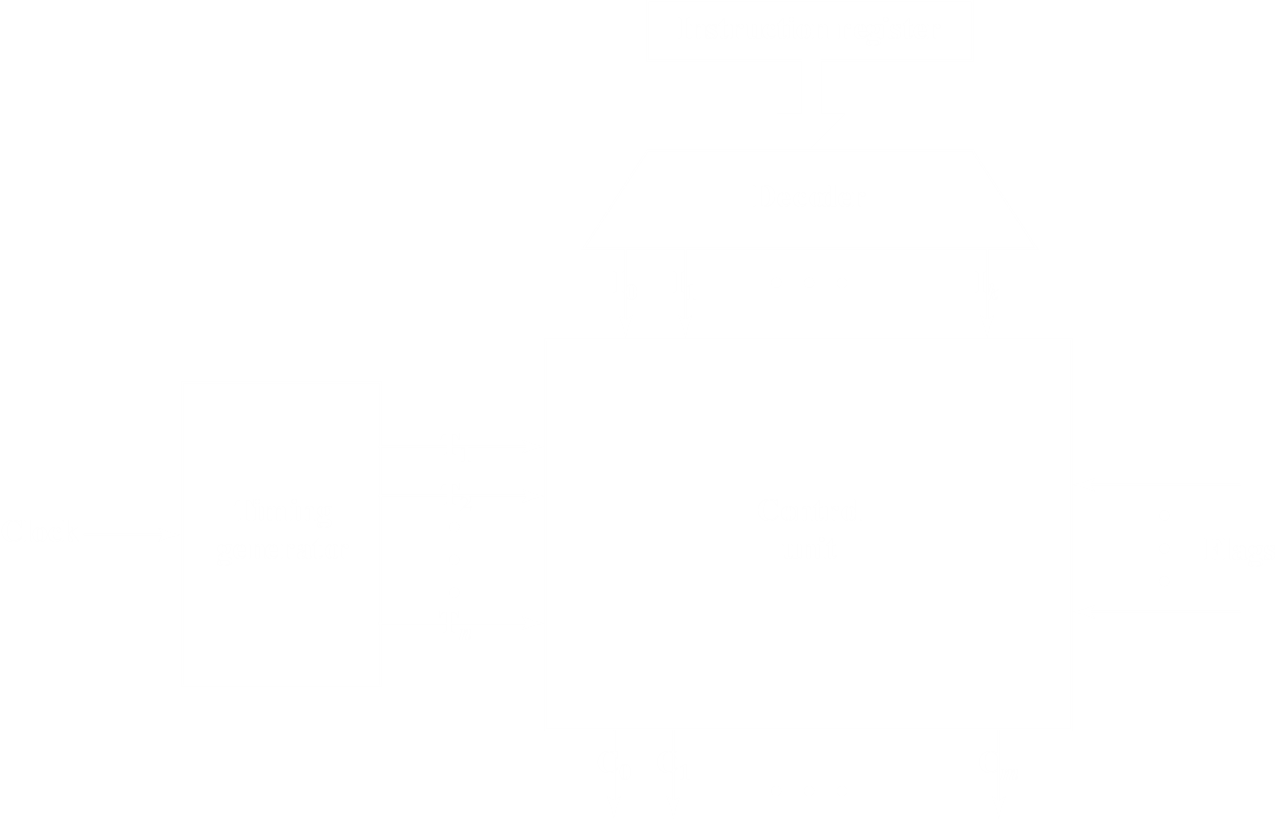
The key inputs are the IR, the clock, flags and control bus signals. The flags and control bus signals send individual bits that have some meaning, e.g. overflow. The other two inputs are not directly useful to the control unit.

Consider the IR. The control unit uses the opcode to take different actions. Each op-code must have a unique logic input, and a decoder can take that and produce a single output. For binary inputs, each of the input patterns will give one unique output.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| I01 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| I02 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| I03 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| I04 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| O01 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| O02 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| O03 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| O04 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| O05 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| O06 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| O07 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| O08 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| O09 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| O10 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| O11 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| O12 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| O13 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| O14 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| O15 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| O16 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

The decoder for an actual control unit will be more complex, to account for variable-length op-codes.

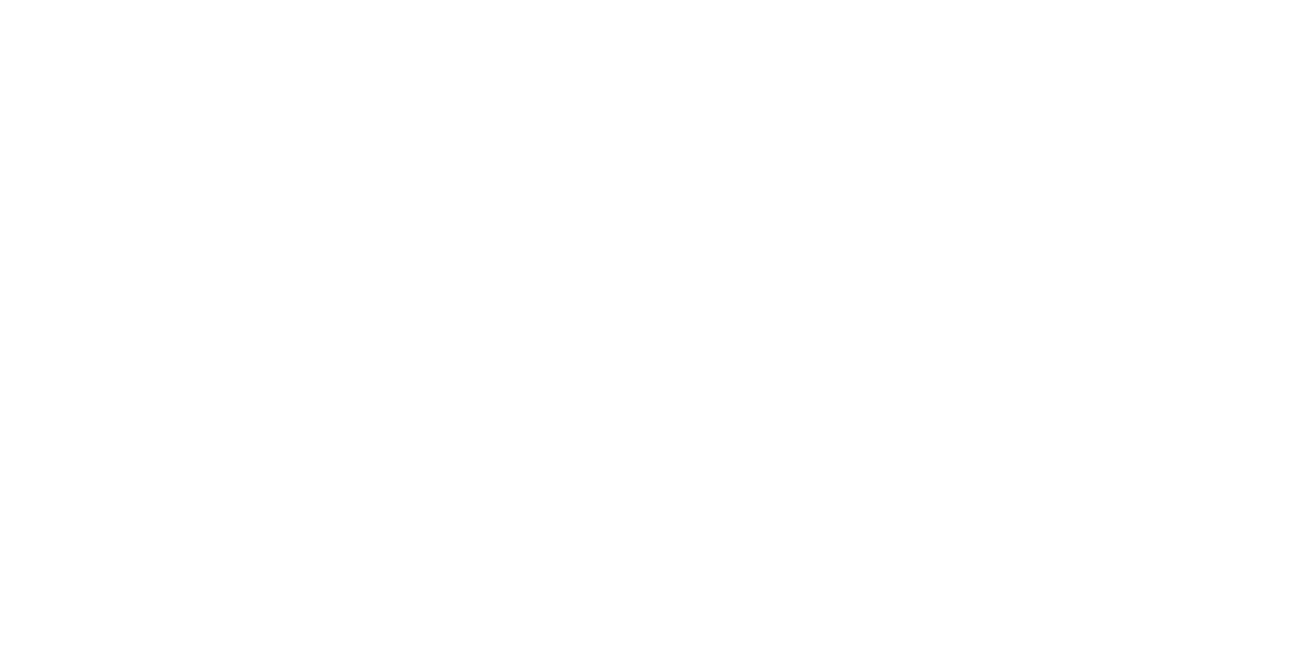
The clock issues repetitive pulses. The period of these must be long enough to allow the propagation of signals along the data paths and through the circuitry. However, the control unit emits different control signals at different time units within a single instruction cycle. This leads to the use of a counter as the input to the control unit. The counter must be reinitialized at the end of an instruction cycle.



### Control Unit Logic

All that remains is to discuss the internal logic of the control unit that produces output control signals based on its input signals.

Essentially, we have to derive a Boolean expression for each control signal as a function of its inputs. It is easier to understand this with the help of an example.



|  |  |  |
| --- | --- | --- |
|  | **Micro-Operations** | **Active Control Signals** |
| Fetch | : MAR 🡨 (PC) |  |
| : MBR 🡨 Memory  PC 🡨 (PC) + 1 | , |
| : IR 🡨 (MBR) |  |
| Indirect | : MAR 🡨 (IR(Address)) |  |
| : MBR 🡨 Memory | , |
| : IR(Address) 🡨 (MBR(Address)) |  |
| Interrupt | : MBR 🡨 (PC) |  |
| : MAR 🡨 Save Address  PC 🡨 Routine Address |  |
| : Memory 🡨 (MBR) | , |

Consider this figure and table once again. Say we want to find the expression for , which causes data to be read from the external data bus into the MBR.

Let us define two control signals and first.

00 Fetch Cycle

01 Indirect Cycle

10 Execute Cycle

11 Interrupt Cycle

is active two times, once during the second time unit of the fetch cycle and once during the second time unit of the indirect cycle. Thus, we can define it as

This expression is incomplete though. We also need during the execute cycle, which is not shown in the table. For now, let us consider that there are only three instructions for which we need to read from memory, LDA, ADD and AND. Thus,

Similarly, we can create Boolean expressions for every control signal generated by the processor. This would give us a set of Boolean expressions that define the behaviour of the control unit, and thus the processor itself.

To summarize, the control unit controls the state of the instruction cycle. At the end of each sub cycle, the control unit reinitializes the timing generator to go back to . It must also set up appropriate values of and to define the next sub cycle to be performed.

In a modern complex processor, the Boolean expressions needed to define the control unit will be much more complicated. The task of implementing a combinational circuit to satisfy all the equations becomes very complicated. This means that usually, a far simpler approach is used, microprogramming.