Design of a Simple Processor

P J Narayanan
IIIT Hyderabad

Contents

Chapter 1

Introduction

Chapter talks about:

- Objectives of studying processor design
- Why this book is needed

Chapter 2

Preliminaries: Digital Circuit Elements

Talks about what a student needs to know to take advantage fo this book:

- Flip-Flops, Registers, Counters
- Multipurpose Registers
- Edge Triggered, Level Triggered
- Programmable ALUs as combinatorial elements with select
- Timing diagrams: Clock, signal, etc.
- Tri-State output, High-Impedence state

The reader of this book is expected to know about the basic digital logic elements, both combinational and sequential. The combinational elements include gates, multiplexers, decoders, etc. The sequential elements include latches, flip-flops of different kinds, registers, etc. Associated concepts like level triggered and edge triggered operations and timing diagrams for such elements are also necessary to fully appreciate the rest of the book. Another useful component is ROM or the Read Only Memory. It is a highly flexible component to implement combinational circuits.

The *tristate* or high-impedence state is very important to fully comprehend a bus based design. It is not necessary for the reader to know the underlying electronics or physics of these devices; a functional understanding of how the devices and modes work will suffice and is expected.

We will summarize the key aspects of the required background in the rest of the chapter. We will use the *description* of a multipurpose register as the vehicle to convey the background

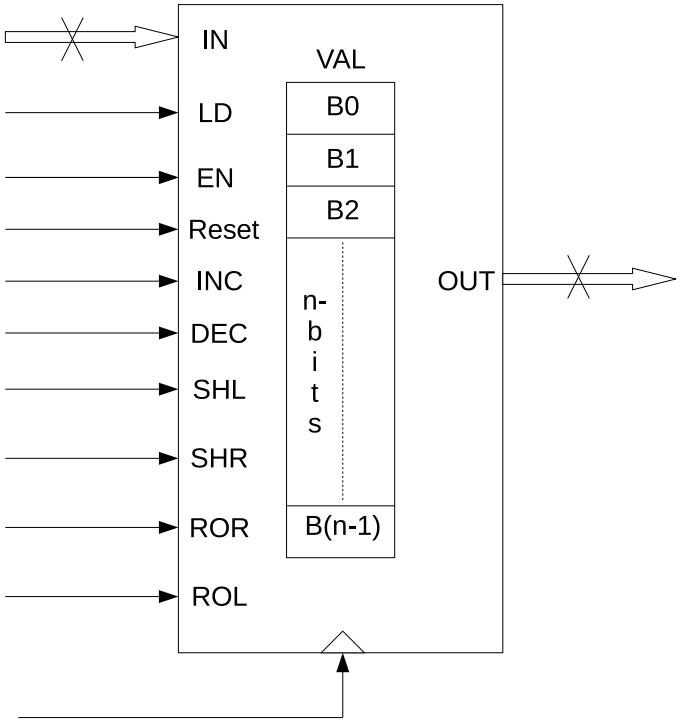

Figure 2.1: Our multipurpose register with n output lines, n input lines, and 10 control lines. The input and output lines are ordinarily connected together on a ICs that implement such a register to reduce the pincount.

requirements. We will not go through the exercise of *designing* the device. However, a student should be able to design such a register based on the digital logic background they have already had.

2.1 A Multipurpose Register

We consider the multipurpose register shown in Figure 2.1. It can store an n-bit quantity (i.e., has a width of n, which could be 8. 16, 32, etc.) The register has the following inputs and outputs.

- Input lines $\texttt{IN}_n$ are logically connected internally to the inputs of the n flip-flops inside the register.
- Output lines $\texttt{OUT}_n$ are the lines that hold the value stored in the register when the register is accessed for read. The $\texttt{OUT}_n$ lines are *tristate*-capable. They are in the *high-impedence*

or floating state ordinarily, except when the outputs are **enabled**. Recall that multiple *tristate*-capable output lines can be connected or wired together, as long as all but one of them is in the high-impedence state **always**. In most cases, the `IN` and `OUT` lines may be on the same electric wires or pins for outside the register.

- The `CLK` input line controls the timing of most operations the register is involved with. The clock synchronizes the outputs to its rising or falling or edge and is an essential part of all sequential circuits. A system clock of relatively high frequency is typically connected to the `CLK` line.

- The input `EN` controls the high-impedence state of the output lines $\texttt{OUT}_n$. When `EN` is 0, the `OUT` lines will be in the high-impedence state. When `EN` is 1, the $\texttt{OUT}_n$ lines will be driven to the electric levels corresponding to the value stored in the register. The lines will resume the floating state (and hence ready to be driven by another source) when `EN` returns to 0.

  The values will be driven at with the *rising edge* of the clock after `EN` line is set to 1, in our register. The values are guaranteed to be available at the `OUT` lines a very small duration (called the access time) later, typically 1 nanosecond or so.

- The input `LD` loads the register from the input, which changes the value stored in it. When `LD` is 1, the value indicated by the electric levels of the $\texttt{IN}_n$ lines will replace the previous value stored in the register.

  The load actually happens at the *falling edge* of the clock in our register. Thus, the `IN` lines should have their stable values a very short duration (called the setup time) before the falling edge. This allows for the `EN` and `LD` signals to be active together. The previous value will be available at the output right after the rising edge. The internal value will change only after the falling edge.

- The input `RESET` controls resetting of the register. When `RESET` is 1, a 0 value will be written to all bits stored in the register. We will assume a synchronous reset, which means the new value will be stored only at the next *falling edge* of the clock.

- Inputs `INC` and `DEC` control the incrementing or decrementing the value stored in the register, interpreted as a number. Thus, if `INC` is 1, the values stored internally will change to 1 more than the previous value, when the binary string is interpreted as a number. Usually, `IN` and `OUT` signals are numbered from 0 to $n-1$, with 0 being the least and $n-1$ being the most significant bit. If `DEC` is a 1, the value will be decremented by 1.

  The new values will replace the old ones at the falling edge of the clock cycle for which `INC` or `DEC` is active.

- Inputs `ROL, ROR` control the left or right rotation of the register contents. Again, the new values are available only at the falling edge of the clock. Rotate left will send the value of the bit at position i to position $(i + 1)$. Similarly for rotate right.

- Inputs `SHL, SHR` control the left or right shift of the register contents. Again, the new values are available only at the falling edge of the clock. Shift left will send bit at position i to position $i + 1$. Similarly for shift right. We assume a 0 will fill the void during the shift.

The register has input lines, output lines, a clock line, and several control input lines.Of the control inputs, `EN` controls whether the output lines are in the floating state or reflect the values stored in the register. It is important to recognize that the value stored internally in the register is not necessarily brought out always. We will denote the internal value using $\texttt{VAL}_n$. The other control inputs indicate an *action* to be performed by the register during the next cycle. The action actually "happens" internally only at the falling edge (or the end) of the clock cycle, resulting in the `VAL` being modified.

The register designed above has 8 possible actions. It should be obvious that at most one of the 8 control lines can be active (i.e., in a 1 state) for any cycle. The behaviour of the register is unspecified if multiple control lines that specify an action are set. `EN` can, however, be active along with one of the other control inputs. The contents of the register not modified if no control signal is active in a clock cycle.

2.2 Read Only Memory (ROM)

A read only memory is a memory with fixed content, divided into words. The contents can be accessed by supplying an address to the memory. The word corresponding to the address will be read and presented to the outside world. Figure 2.2 presents the schematic of a typical ROM.

A ROM has k address lines which can refer to 2^k distinct "locations". Each location stores an n-bit quantity. Thus, the contents of the ROM is a $2^k \times n$ table with previously stored values. Each bit of output can be thought of implementing *any* function of k binary variables, with a value for each combination of those. Thus, such a ROM can implement n independent, general combinational circuits of k binary signals. The n outputs corresponding to the current combination of inputs will be accessed using the input lines as address into the ROM.

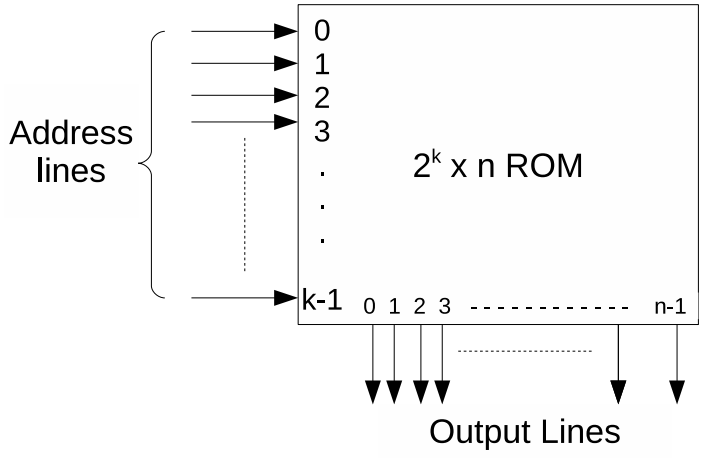

Figure 2.2: A Read Only Memory (ROM) has k address lines and a word size of n. It can be used to implement n independent combinational circuits of k variables.

2.3 Timing Diagrams

Figure 2.3 shows the timing diagrams of a the key operations on the register. Among the actions, only load, increment, and reset are shown. The other operations are similar. All actions that change the value take effect at the falling edge of the clock, while the enable that puts the value stored in the register to the outside take effect at the rising edge of the clock.

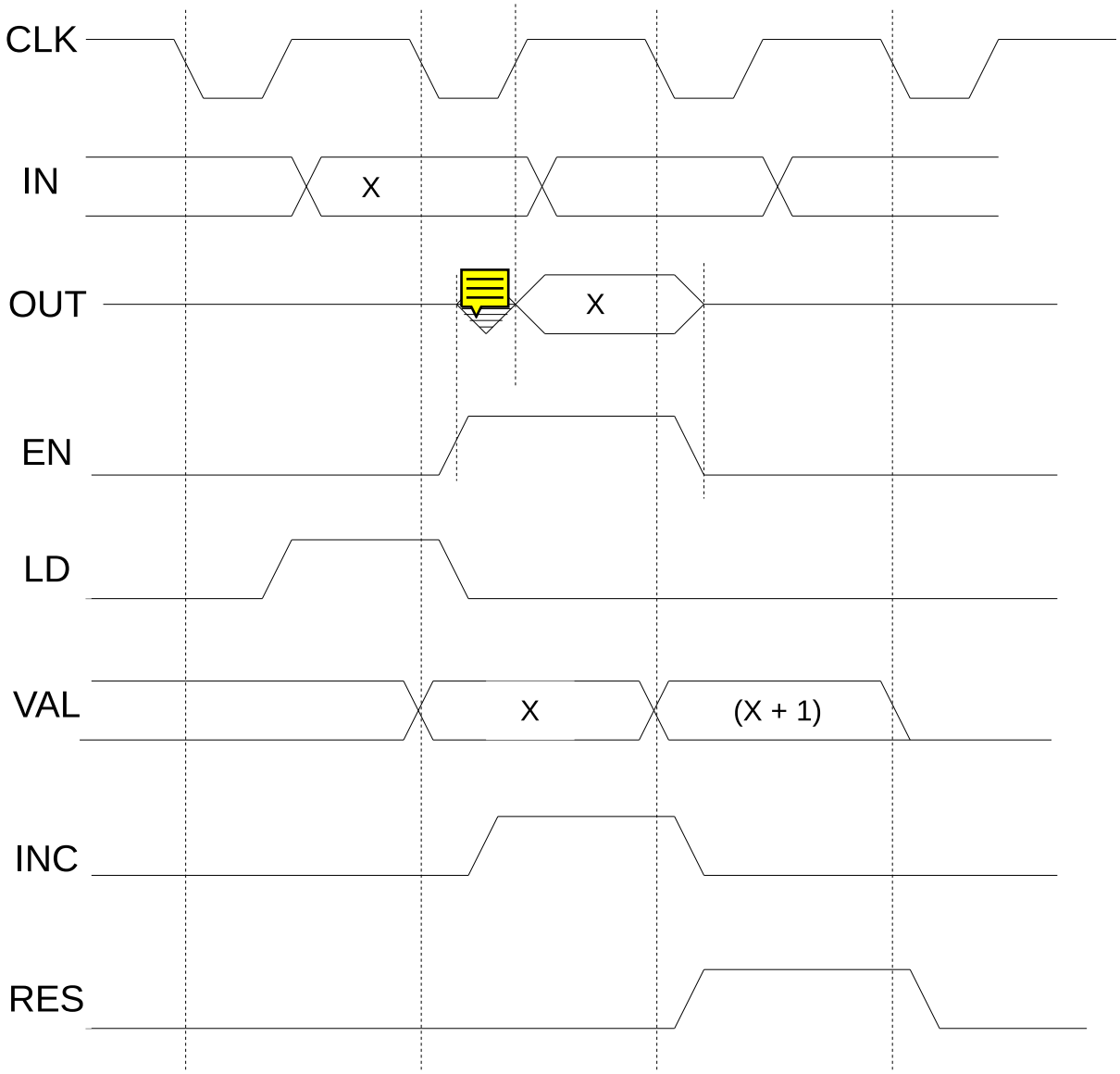

Figure 2.3: Timing of the multipurpose register for different actions. The impact of `LD`, `INC`, and `RESET` are shown. Other actions behave similarly. `EN` directly controls the high-impendence state of the output lines.

Chapter 3

Devices Connected on a Bus

In this chapter, we will look at the concept of a *bus* with different digital elements or devices connected to it. Our system, has a single, central *bus* to which all its components are connected. The bus itself can be thought of as a set of wires that run from one end to the other. These common wires transmit signals electrically among the devices connected to it. The number of wires or the *bus width* depends on the number of signals to be carried.

Figure 3.1 shows the picture of a central *bus* to which 4 registers are connected. Each register is connected to the bus as follows:

- The n lines of `IN` are connected to corresponding wires of the bus.
- The n lines of `OUT` are connected to corresponding wires of the bus.

It can be seen that corresponding input and output signals of a register are electrically connected together. Moreover. the input and outputs signals of *all* registers are connected together! Ordinarily, one cannot connect two outputs together without damaging them as each will try to drive the output to possibly different electrical levels. However, the outputs of our registers are tristate buffers. They can be connected together if no more than one is driving its output. We can guarantee this by ensuring that at most one register has its `EN` line at level 1.

3.1 Data Transfer over the Bus

Let us concentrate on the registers A and B in Figure 3.1. Their control lines are respectively indexed using `A` and `B`. Thus, the enable, load, and increment control signals of register A are respectively lablelled $\texttt{E}_\texttt{A}$, $\texttt{L}_\texttt{A}$, and $\texttt{I}_\texttt{A}$. The same for register B are labelled $\texttt{E}_\texttt{B}$, $\texttt{L}_\texttt{B}$, and $\texttt{I}_\texttt{B}$.

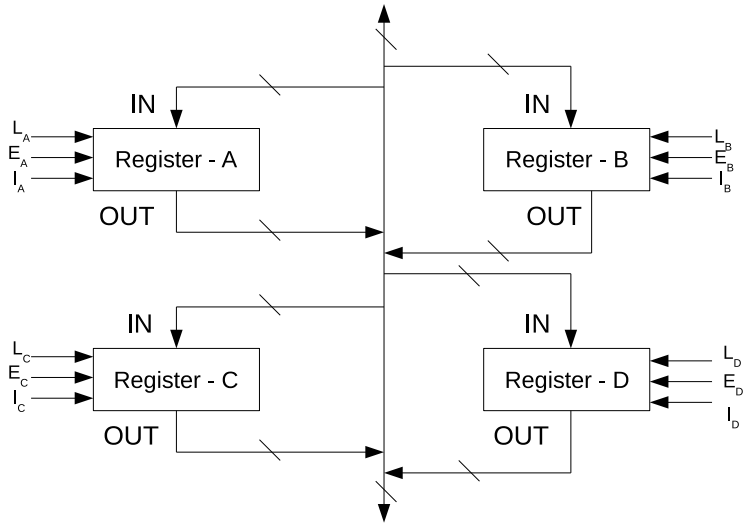

Figure 3.1: A central bus with a few registers connected to it.

Consider a clock cycle in which the signals E_A and L_B are active (i.e., is at 1) as shown in Figure 3.2. We assume all other signals are at level 0. Let us analyze what will happen at the bus and the registers A and B during the clock cycle.

The register A gets the `EN` signal and drives the output lines with its contents shortly after the rising edge of the clock. Since the output lines are connected to the bus lines, the bus will hold the bits stored in register A shortly. These lines are also connected to the input lines of all registers including B. Register B sees its `LD` signal active during the clock. The data on its input lines will replace the contents of register B at the falling edge of the clock. The timing diagram for for the bus and the registers for the whole clock is shown in Figure 3.3.

Let us reflect on what happened in the clock cycle. Data moved from register B through its

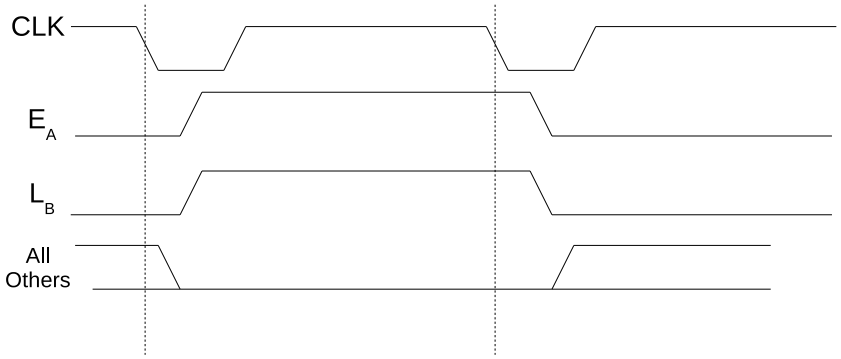

Figure 3.2: Only E_A and L_B are active during this clock cycle

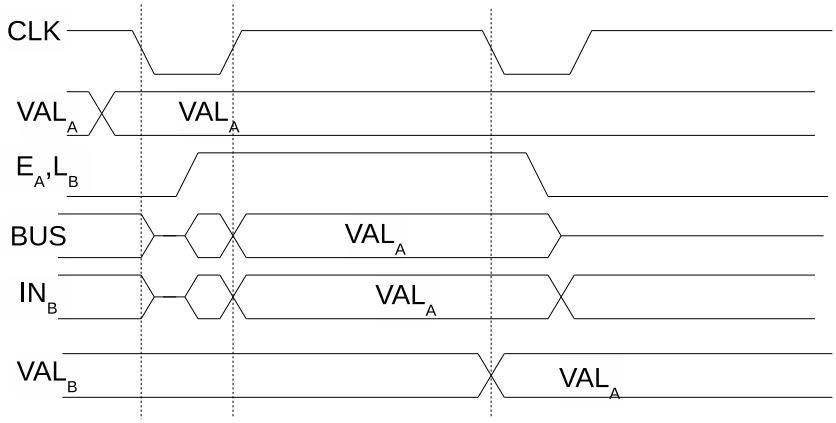

Figure 3.3: Timing diagram for the whole clock for the control inputs shown in Figure 3.2

output lines to the bus and then to the input lines of register B and got stored in it! In effect, we have moved the data to register B from register A, like a assignment instruction! At the level of the hardware, activating 2 signals simultaneously was all that was done. The rest happened due to the bus connection, the clock, and the design of the register.

We will denote the input combination at a clock cycle (say clock cycle i) by indicating the control signals that are active during it, as:

Ck i: $\mathtt{E_A}$, $\mathtt{L_B}$.

This combination will effectively perform the assignment: $\mathtt{B} \leftarrow \mathtt{A}$. You should note that the above description indicates the set of signals that are active during a clock cycle. There is no ordering between the signals. We could as well written it as:

Ck i: $\mathtt{L_B}$, $\mathtt{E_A}$.

Let us next consider the case when the following combination of 4 signals are active during a clock cycle:

Ck j: $\mathtt{L_B}$, $\mathtt{E_D}$, $\mathtt{L_C}$, $\mathtt{L_A}$

What happens to the registers in that clock cycle? The contents of register D are copied simultaneously to registers A, B, and C via the bus! Of course, this should not come as a surprise as each registers is capable of loading from the input lines independent of the others.

Let us now examine the following combination at a clock cycle:

Ck j: $\mathtt{E_B}$, $\mathtt{L_A}$, $\mathtt{I_B}$.

What happens in that clock cycle? The contents of register B is copied to register A. However,

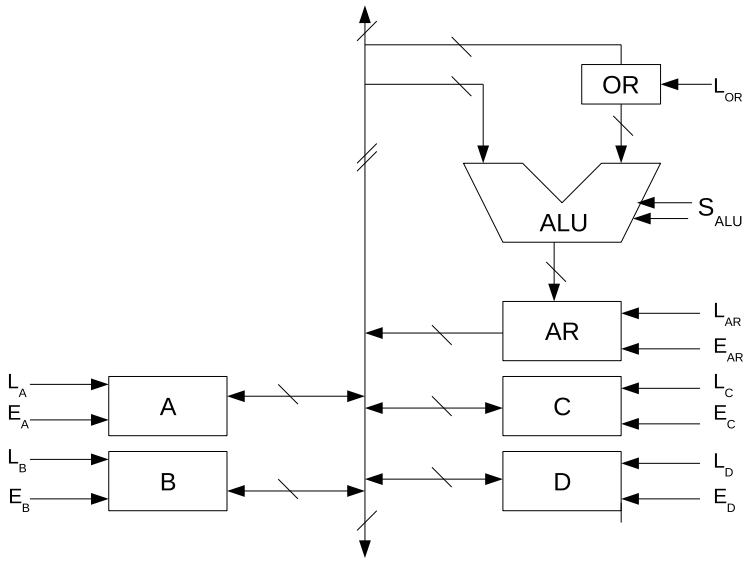

Figure 3.4: Single bus configuration with registers and an ALU

the contents of B get incremented simultaneously also. So, does the old value get written into A or the new value? If register B held the number 10 before this clock, what will be written into A: 10 or 11? Let us recollect when in the cycle increment (and other content-changing actions) actually changes the internally stored value. Contents of B (or $\texttt{VAL}_B$) change at the falling edge of the clock, which is when the contents of the bus are loaded into A too. Which will happen first? It turns out that the design of the circuit will ensure that the new contents of register B will not be seen on the bus in such cases. Thus, A will contain the old value of B, namely, 10 in our example.

This is true for all content-changing actions on the register. Thus, a register can be used both as a source of data to the bus as well as a destination for change in its own contents *in the same clock cycle* without any worry. We will use this feature at many places during the design of our simple processor.

3.1.1 Arithmetic Processing and Data Transfer

Consider the configuration shown in Figure 3.4, which has a few registers and an ALU. Assume the registers are made using the multipurpose registers. The ALU is a combinational circuit with 2 inputs and two lines to select the function to be performed. The two select lines can be in one of 4 combinations. We refer to them symbolically as `ADD, SUB, AND, PASSO`. Each of these refers to a combination of the 2 binary inputs, but we refer to them symbolically for

convenience. `ADD` produces the sum of the inputs interpreted as binary integers at the output. `SUB` produces the result of subtracting the input on the right (input 1) from the input on the left (input 0). `PASSO` copies the values at the left input to the output, ignoring the other one completely.

The ALU has its left input connected to the bus and the right input to a register called `OR` or *operand register*, whose inputs are connected to the bus. The output of the ALU is connectted to the input of a register called `AR` or *accumulator register*, whose output is connected to the bus. Every register has all the control signals of our generic multipurpose register.

Consider the following combination of signals active in a clock cycle:

Ck i: $\mathtt{E}_{\mathtt{OR}}$, $\mathtt{E}_{\mathtt{A}}$, $\mathtt{L}_{\mathtt{AR}}$, $\mathtt{SALU}_{\mathtt{ADD}}$.

Two enables are simultaneously active. Does it cause conflict at the bus? No, since the output of `OR` is not connected to the bus, there will be no conflict. The last term above indicates that the select lines of the ALU are set to the combination `ADD`.

What happens at the above clock cycle? The ALU performs addition of its inputs, which are the contents of `OR` register and register A. The sum is written to `AR` register. This can also be written as `AR` $\leftarrow$ `A + OR`, a combination of an arithmetic operation and data movement.

Let us consider the impact of the following multicycle combination of signals:

Ck 10: $\mathtt{E}_{\mathtt{A}}$, $\mathtt{L}_{\mathtt{OR}}$
Ck 11: $\mathtt{E}_{\mathtt{B}}$, $\mathtt{E}_{\mathtt{OR}}$, $\mathtt{L}_{\mathtt{AR}}$, $\mathtt{SALU}_{\mathtt{SUB}}$
Ck 12: $\mathtt{E}_{\mathtt{AR}}$, $\mathtt{L}_{\mathtt{C}}$

What does the 3 clock cycle combination achieve? The contents of A are copied to `OR` in cycle 10. The contents of `OR` are subtracted from the contents of B and the result is loaded to `AR` in cycle 11. The contents of `AR` are copied to register C in cycle 12. The following "operation" or assignment is performed by the above sequence: `C` $\leftarrow$ `B - A`.

It is easy to see how addition, subraction and logical AND with any combination of 2 registers as input and any register as output can be implemented using a very similar 3-cycle sequence of combinations of signals. For instance, we can implement `D` $\leftarrow$ `C` $\wedge$ `D`, where $\wedge$ stands for logical AND, as follows:

Ck 10: $\mathtt{E}_{\mathtt{C}}$, $\mathtt{L}_{\mathtt{OR}}$
Ck 11: $\mathtt{E}_{\mathtt{D}}$, $\mathtt{E}_{\mathtt{OR}}$, $\mathtt{L}_{\mathtt{AR}}$, $\mathtt{SALU}_{\mathtt{AND}}$
Ck 12: $\mathtt{E}_{\mathtt{AR}}$, $\mathtt{L}_{\mathtt{D}}$

It is clear that we can make the hardware perform several steps by carefully selecting the control signals to different units that are active in each clock cycle. We can also get larger "operations"

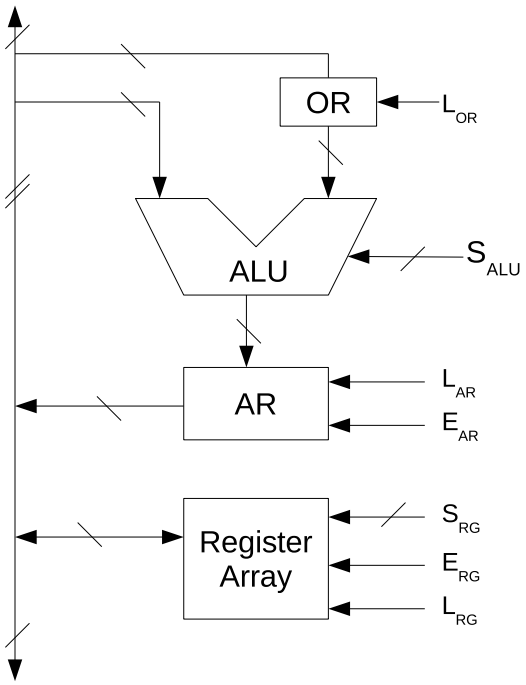

Figure 3.5: Enhanced single bus system with an ALU and a register array

implmented using multiclock sequences of such combinations. Each such clock cycle is typically referred to as a *microcycle*, which is the basic time unit in which something happens within the processor.

3.2 Enhanced Singlebus Architecture

Let us add a few more components to the single bus organization seen earlier. The modified *architecture* is shown in Figure 3.5. In this context, the word *architecture* refers to the internal arrangment of components the *processor* we are building.. As the name indicates, a single bus architecture has all its components connected to its central bus. It has the same ALU, `OR` and `AR` registers as before. However, the registers have been consolidated into a *register array* or a *register file*. These are numbered `R0` through `R11`. The register file has a single enable input E_{RG} and a single load L_{RG} and 4 select lines S_{RG}. The select lines identify which register of the file is being operated on, with enable or load controlling the action performed on it. The new design of the register array needs only 6 control signals – 4 for select and 2 for enable/load – as opposed to 22 that would be needed were each register to have its individual enable and load signals. Some generality is, however, lost as only one register can be selected for writing. The registers

are also simpler devices as they need to do only parallel load and read; reset, increment, shift, etc., are not possible on them. These registers are temporary store of information, with support for load and store only.

We also have an enhanced ALU in place, with 3 select lines and supporting 8 operations: none, add, subtract, logical and, or, xor, and pass left. The ALU takes the left operand from the bus and the right one from `OR`.

Let us see how operations involving registers and the ALU can be performed on this architecture. We will use a simple type of operation: every one of them has `AR` as one of the operands and has it at the destination. The opertions are thus `ADD R1`, `XOR R11`, `SUB R7`, `OR R0` etc. We will implement these actions internally using multiple cycles as before.

`ADD R1`:

Ck 0: E_{RG}, L_{OR}, $S_{RG} \leftarrow$ 1

Ck 1: E_{AR}, L_{AR}, $S_{ALU} \leftarrow$ ADD

`XOR R11`:

Ck 0: E_{RG}, L_{OR}, $S_{RG} \leftarrow$ 11

Ck 1: E_{AR}, L_{AR}, $S_{ALU} \leftarrow$ XOR

`SUB R7`:

Ck 0: E_{RG}, L_{OR}, $S_{RG} \leftarrow$ 7

Ck 1: E_{AR}, L_{AR}, $S_{ALU} \leftarrow$ SUB

`OR R0`:

Ck 0: E_{RG}, L_{OR}, $S_{RG} \leftarrow$ 0

Ck 1: E_{AR}, L_{AR}, $S_{ALU} \leftarrow$ OR

We also need a way to load values from register to `AR` and to save results from `AR` to a register. We call these `load` and `store` respectively. `LOAD R4` and `STOR R9` can be implemented as follows.

`LOAD R4`: Ck 0: E_{RG}, L_{AR}, $S_{RG} \leftarrow$ 4, $S_{ALU} \leftarrow$ PASS0

`STOR R9`: Ck 0: E_{AR}, L_{RG}, $S_{RG} \leftarrow$ 9

The load is an example when selection of an ALU function and a register are simultaneously performed.

3.3 An Accumulator Architecture

The style of operations in which an accumulator register is a source and the destination of all operations provides a simple model and was popular in early processors. Several real processors followed this model, known as the *accumulator architecture*. We will use it to simplify matters for this book. We can think of Figure 3.5 as defining the essential elements of a very simple processor that follows the accumulator architecture.

The architecture refers to the organization of the physical components, such as the ALU, registers, bus, etc., that make up the processor. The next task is to define a low level language

that implements an accumulator based computing model. Programmers can write code using this language, which can be run on the processor. We will see the layers of design that make this possible in sufficient detail in the rest of the book. The next topic to discuss is the set of instructions that we will use. We will do so in the next chapter.

Chapter 4

Basics Instructions for the Simple Processor

We now describe the set of "instructions" that our simple processor will support at the lowest level. The idea is to devise a small set of instructions necessary for the purpose of our understanding. We are not ambitious in this aspect and will stick to a small number of representative instructions, rather than attempt to be complete or optimal in our choice.

4.1 Machine Instructions and Assembly Instructions

A digital processor can handle only binary strings at the very lowest level. Thus, all instructions to be carried out by a digital processor needs to be coded or represented as binary strings. Different basic instructions have to be coded as unambiguous binary strings. The hardware is capable of looking at a string, comprehending it (or decoding it), and carrying out the corresponding instruction. A sequence of such strings to achieve something forms a program.

We follow the basic *von Neumann* or a a stored-program model for our processor, shown in Figure 4.1. In this model, the program and any data it manipulates are stored in the memory. Instructions that make up the program are stored sequentially in memory. Each instruction is a binary string that encodes the operations to be performed without ambiguity. The processor *fetches* the instructions one by one from the memory and *executes* it or carries out the corresponding actions. Real work gets done as a side-effect of executing these instructions, as the instructions can read data stored in memory, perform arithmetic, logic, and other operations on the data, and store the results back into the memory. Special instructions can also control the input and output from the processor, but we will not consider those for the simple processor.

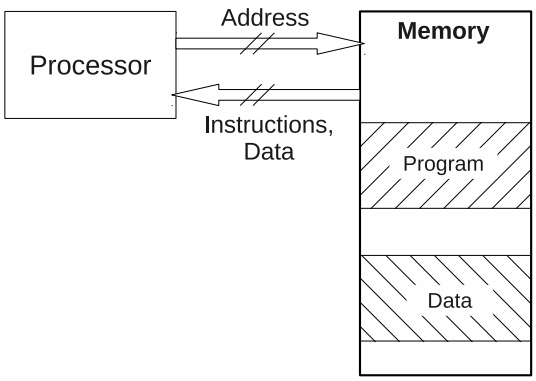

Figure 4.1: The stored program or von Neumann model

We follow this scheme for our simple processor. Coded instructions are *fetched* from memory and *executed* by our processor. In fact, the processor is engaged in a perpetual loop of fetch and execute, with the real work done as the side effect of executing the instructions. We will first look at the execute step of the processor. We will discuss the mechanism of fetching later. We will also discuss how the endless fetch-execute loop is realized within the processor.

The binary coded instructions are referred to as *machine instructions*, following the *machine language*. This is really no "language" but an encoding scheme that which makes unique decoding of the instructions possible. Encoded instructions are called *machine code* or *opcode* for operation code. These are understood by the processor naturally. Conversely, that is the only "language" understood by the processor as it can understand no high level language (like C/Python).

Machine instructions are meant only for the processor; they require tremendous effort to interpret by us. A mapping of the machine instructions for easier grasp by humans is used widely by processors. This representation is essentially a one-to-one mapping from machine instructions, using mnemonics or nearly comprehensible short words and symbolic representation of internal resources like the registers. Such a representation of the basic instructions is called the *assembly language* consisting of *assembly instructions*. For example, the machine instruction for `add` may be `0x10` (hexadecimal for 16) and for `logical or`, `0x50`. The corresponding assembly instructions may use the mnemonic `ADD` and `OR`, which are far more comprehensible to humans.

For all practical purposes, each assembly instruction maps to a machine instruction. Thus, they can be used nearly interchangeably. We will now design the basic instructions for our simple processor. We will design the assembly instructions and the corresponding machine instructions together. There is no recommended way of doing this, though some of the design may be clearer after we are done with the whole design.

4.2 Arithmetic and Logic Instructions

Let us assume the word length of our processor is 8 bits. Thus, all entities we will handle are 8-bits wide, which includes the coded instructions as well as data elements. Our instruction set will have the arithmetic and login instructions listed earlier, namely, **add, subtract, and, or,** and **xor**.

Assembly Instruction	Machine Code	Action
`add <R>`	10-1F	[AR] $\leftarrow$ [AR] $+$ [<R>]
`sub <R>`	20-2F	[AR] $\leftarrow$ [AR] $-$ [<R>]
`xor <R>`	30-3F	[AR] $\leftarrow$ [AR] $\oplus$ [<R>]
`and <R>`	40-4F	[AR] $\leftarrow$ [AR] $\wedge$ [<R>]
`or <R>`	50-5F	[AR] $\leftarrow$ [AR] $\vee$ [<R>]
`cmp <R>`	60-6F	[AR] $-$ [<R>]

Table 4.1: Assembly and machine codes of arithmetic and logic operations with corresponding action. `<R>` stands for a register specified using the rightmost 4 bits of the machine code.

Table 4.1 lists the arithmetic and logic instructions with a register the argument. The `<R>` in the first column of the table is a parameter that can be replaced by one of `R0` to `R11`, with the corresponding number appearing in the lower half of the machine code, given in the second column. Thus, `ADD R1` will be coded as `0x11`, `XOR R8` as `0x38`, and `OR R11` as `0x5B`. Any opcode in that range can be unambiguously understood too. Thus, `0x27` stands for `SUB R7`, `0x42` for `AND R2`, etc. The last instruction performs a comparison of the register and `AR` without changing the value of the accumulator. This may seem pointless as the results are not used. However, the arithmetic and logic operations have other side-effects based on the results of the operation. This could include overflow, carry generation, value being negative, etc. These find use in controlling loops in conjunction with conditional branching instructions we will encounter later.

The last column of Table 4.1 lists the action corresponding to the instruction. We follow the convention that `[R1]` with the square brackets stands for the contents of register `R1`. The square brackets will be used with this meaning in this document, unless otherwise specified.

We will allow another variation of the above arithmetic and logic instructions in which the actual operand is specified in the instruction itself. Such instructions are frequently needed to initialize variables to a constant, such as the loop counter to 0. Such instructions are set to provide their arguments in the *immediate mode*. Table 4.2 lists the assembly instructions and the corresponding machine codes for the immediate mode instructions.

These instructions need to hold the operands along with the instruction. Since the operand is

Assembly Instruction	Machine Code	Action
`adi xx`	01	`[AR]` $\leftarrow$ `[AR]` $+$ `xx`
`sbi xx`	02	`[AR]` $\leftarrow$ `[AR]` $-$ `xx`
`xri xx`	03	`[AR]` $\leftarrow$ `[AR]` $\oplus$ `xx`
`ani xx`	04	`[AR]` $\leftarrow$ `[AR]` $\wedge$ `xx`
`ori xx`	05	`[AR]` $\leftarrow$ `[AR]` $\vee$ `xx`
`cmi xx`	06	`[AR]` $-$ `xx`

Table 4.2: Assembly and machine codes of arithmetic and logic operations with operands specified in the instruction itself in an immediate mode.

also of the same width as the instruction, these can't be hidden in the machine code itself. We assume the operand, indicated by `xx`, is stored in the word that immediately follows the machine code that indicates such an operaiton. Thus, all instructions in Table 4.2 will need 2 words in the machine code, with the opcode occupying the first word and the operand occupying the second.

4.3 Data Movement Instructions

We need instructions to move data from and to the accumulator to get our work done. We have seen the instructions to move contents of `AR` from or to a register. We also need instructions to move from `AR` to and from the memory, which lies outside the processor. Registers are not sufficient to hold all our data, such as the array of marks obtained by all students. These are kept in the memory and is brought in and out of the processor as needed.

The `movs` instruction moves a register to the accumulator and the `movd` instruction moves the accumulator to a register. The register number involved is embedded into the opcode as a parameter as before. An additional instruction `movi` is provided to move an immediate constant directly to a register.

The **load** and **stor** instructions involve a register and a memory location. Memory resides outside of the processor. The memory is divided into words which are stored sequentially. The capacity of the memory is a certain number of words. Each word has a unique *address* that starts with `0` for the first word and runs sequentially till the address `(capacity - 1)`. To access the memory, one needs to give it an address to indicate which of its words is to be accessed. The contents of the corresponding memory word will be given to the processor on a read. The processor has to supply the contents to be written memory for a write. The number of bits of address determines the maximum capacity of memory that can be used.

Assembly Instruction	Machine Code	Action
`movs <R>`	70-7F	[AR] ← [<R>]
`movd <R>`	80-8F	[<R>] ← [AR]
`movi <R> xx`	90-9F	[<R>] ← xx
`stor <R>`	A0-AF	[[AR]] ← [<R>]
`load <R>`	B0-BF	[<R>] ← [[AR]]

Table 4.3: Instructions to move data from and to the accumulator and other registers and memory.

We assume that the memory address is represented using one word of 8 bits in our processor. Thus, the maximum memory capacity is $2^8 = 256$ words in our simple processor. The `load` and `stor` instructions use the contents of `AR` as the address. The word read from the memory is stored into the register specified in the instruction for the `load` instruction. The value to be written is available in such a register for the `stor` instruction. The action given in Table 4.3 indicates the associated action for these instructions. In the notation with two square brackets `[[AR]]`, the inner brackets indicate contents of `AR` and the outer ones indicate its use as address of the memory.

4.4 Instruction Fetch and Execute

We will look at the process of instruction fetching and execution. The processor works autonomously as a continuous fetch-and-execute engine, with no other input than an external clock. Since instructions as in the machine code are stored in memory, they have to be brought to the processor one by one and executed. The instruction at address $(i + 1)$ has fetched and executed after instruction i, since the instructions of a program are stored consecutively in the memory. The processor has to do all these by itself.

Processors have a special register inside them that manages the process of instruction fetch by keeping track of the address of the next instruction to be fetched at all times. This register is called the *program counter* or the `PC`. The processing of an instruction begins with fetching its opcode from the memory word whose address is in the PC. The contents of the PC are incremented while this happens to hold the address of the next instruction in the sequential order. The opcode is brought to the processor and appropriate action is performed in the execution phase. Once this is completed, the next instruction is processed by fetching it from the memory using PC as the address. This goes on for ever inside the processor until a special `STOP` instruction is encountered. Executing this instruction stops all activities of the processor. Figure 4.2 illustrates this process.

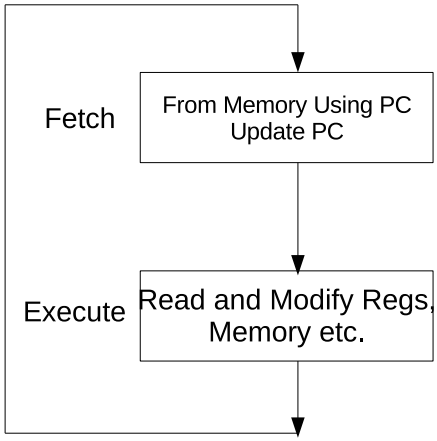

Figure 4.2: Processor runs a fetch followed by execute cycle endlessly.

4.5 Starting the Fetch-Execute Process

How does this whole process begin, however? It is clear that once one instruction is done with, the next one is taken up by incrementing the PC until the `STOP` instruction is encountered. Thus, once the execution of a program starts, everything goes on as the program indicates. A program can be started by loading the address of its first instruction into the PC. However, how does the very first program start when the computer's power is turned on?

We know *Operating System* (OS) is the program that controls our computer. The OS itself is loaded into the processor's memory from the hard disk on boot up prior to taking over the system. Which program loads the operating system? How does that program get the control at the very beginning?

The modern PCs have a program called the BIOS (Basic Input Output System), which is the very first one to get control of the processor. How does the BIOS get control? The processor hardware has a special feature to load a value of `0` to the PC when power is turned on or when the reset button of the computer is pressed. Thus, the very first program that gets control is the one that is saved at memory address `0`. The computer manufacturers have placed a special read only memory at address 0 that has the BIOS program, which knows how to load the operating system from the boot record and proceed accordingly.

Thus, the PC is a register with an increment facility to move to the next instruction after the current one and a reset to 0 facility to start off at with a known program on power on or reset. The modified single bus architecture shown in Figure 4.3 shows as additional components the PC as well as the memory access mechanism, which is discussed next.

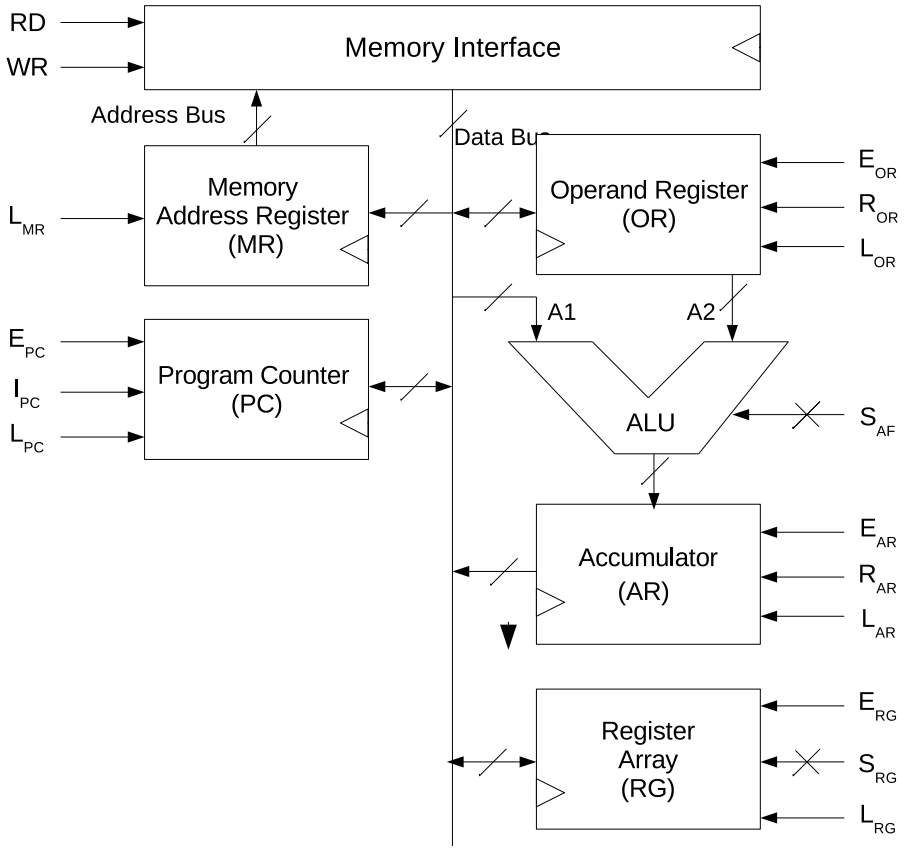

Figure 4.3: The single bus processor with PC and memory access mechanism

4.6 Accessing the Memory

How does the simple processor access the memory? A memory interface that supplies the address to the memory along with the signals to indicate if a read or a write is desired. Data should be presented separately for writes; data supplied by the memory should be used inside the processor for reads.

We assume an external memory interface consisting of address lines, data lines, and two control lines. The data lines are connected directly to the data lines of the bus, as if the memory is a large register array, but ouside of the processor. The address has to be supplied separately, prior to the read or write operation. We assign a *memory address register* (MAR) to hold the address. The MAR is connected to the bus like other registers and can be written to from the bus. There is usually no need to enable the MAR to the internal bus. It can be assumed to be enabled always to the external memory interface. Two control lines `RD` and `WR` are sent to the memory to indicate memory read and write respectively. Figure 4.3 shows these components.

4.6.1 No operation and Stop

We will introduce two more simple instructions. The first one is the `NOP` instruction for no operation and the second is the `STOP` instruction to stop the processor. `NOP` does nothing; that is, when it is executed nothing at all changes in the processor or memory. It may seem superfluous, but comes into use when nothing is required from the processor other than spending the required clocks to fetch and execute this instruction. The `STOP` instruction, on the other hand, terminates the endless fetch-execute cycle that the processor is engaged in. The processor enters a state of complete inaction when the `STOP` instruction is executed. There is no way to come out of this state through a program as the processor has stopped looking at programs! Thus, the only way to come out is a hard **reset** through a reset button or through power cycling. Needless to say, `STOP` has to be the last instruction, if at all. Ordinarily, one would like to assign the responsibility of stopping the processor to a supervisory program like the OS. Table 4.4 describes these two instructions.

Assembly Instruction	Machine Code	Action
`nop`	00	-
`stop`	07	(Stops fetch)

Table 4.4: Instructions for no operation and stopping the processor

We will now look at how the simple instructions described in this chapter can be implemented on our processor hardware in terms of the control signals.

Chapter 5

Implementing the Instructions

In this chapter, we will see how the instructions of our simple processor can be implemented on the processor that uses the single bus architecture shown in Figure 4.3. The architecture has the components and associated control signals, as shown the figure. Since the instructions differ only in their execution phases, we will first explore how each instruction is *executed* on the single bus architecture.

5.1 Arithmetic and Logic Instructions

Instruction	**Control Signals**	**Select Signals**
`add <R>`	Ck 3. E_{RG}, L_{OR} Ck 4: E_{AR}, L_{AR}, End	$S_{RG} \leftarrow$ <R> $S_{ALU} \leftarrow$ ADD
`sub <R>`	Ck 3: E_{RG}, L_{OR} Ck 4: E_{AR}, L_{AR}, End	$S_{RG} \leftarrow$ <R> $S_{ALU} \leftarrow$ SUB
`xor <R>`	Ck 3: E_{RG}, L_{OR} Ck 4: E_{AR}, L_{AR}, End	$S_{RG} \leftarrow$ <R> $S_{ALU} \leftarrow$ XOR
`and <R>`	Ck 3: E_{RG}, L_{OR} Ck 4: E_{AR}, L_{AR}, End	$S_{RG} \leftarrow$ <R> $S_{ALU} \leftarrow$ AND
`or  <R>`	Ck 3: E_{RG}, L_{OR} Ck 4: E_{AR}, L_{AR}, End	$S_{RG} \leftarrow$ <R> $S_{ALU} \leftarrow$ OR
`cmp <R>`	Ck 3: E_{RG}, L_{OR} Ck 4: E_{AR}, End	$S_{RG} \leftarrow$ <R> $S_{ALU} \leftarrow$ CMP
`nop`	Ck 3: End	-

Table 5.1: Microinstructions to implement the arithmetic and logic instructions

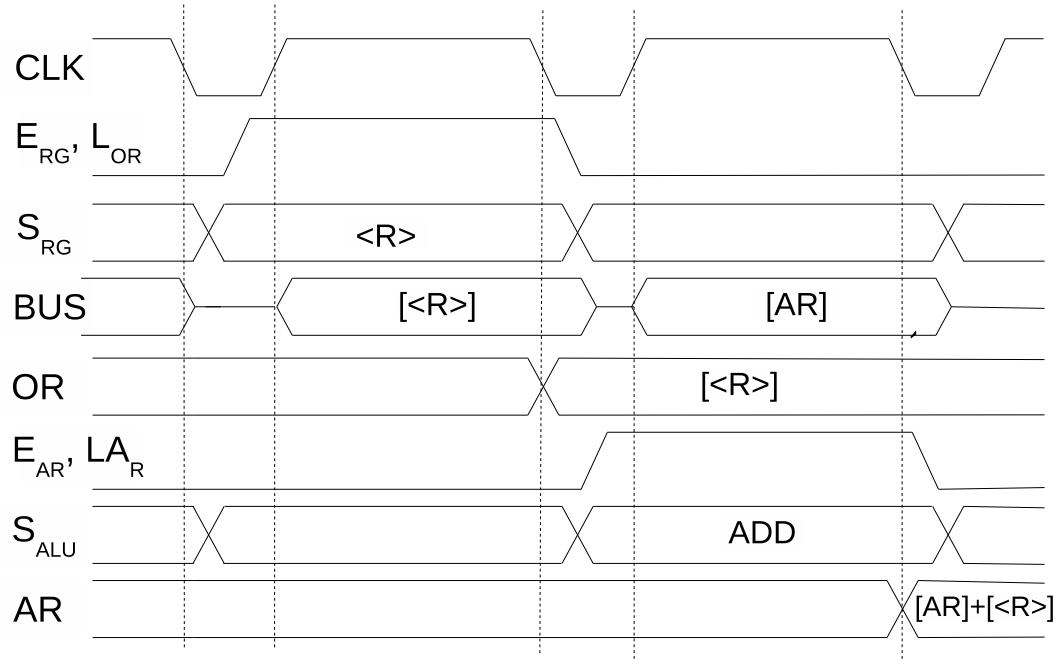

Figure 5.1: Timing diagram for the execution step of the `ADD` instruction

We first look at the instructions from Table 4.1. Similar to the operations we did in Section 3.2, we can describe the execution phase of each of the instructions as given in Table 5.1.

It is clear from Table 5.1 that the implementations of all the instructions are very similar. Each takes two clock cycles to complete. The first clock cycle loads the selected register to the operand register `OR`. This is achieved by moving the value of the selected register from the register file to `OR`. The specific register is selected by setting the S_{RG} control lines the same as the lower 4 bits of the instruction. The second clock cycle performs the selected operation on `OR` and the contents of `AR`, which are enabled onto the bus, and stores the result into `AR`. Figure 5.1 shows the timing diagram for the add instruction; the others are obviously similar. Please note that the compare instruction is slightly different. The L_{AR} signal is missing as the instruction does not change the contents of `AR`. That instruction still needs 2 microcycles since the actual compare (or subtraction) happens at the ALU only in the second clock, whose side effects happen then. The `NOP` instruction is also given in the above table as it is the easiest to implement!

A control signal named `End` is part of the last step of each instruction's execution. This is a special signal that indicates that the execution of the instruction is over. The processor will use this as the cue to attempt fetching the next instruction in the endless cycle shown in Figure 4.2.

A clock cycle in which one basic operation is performed (recall the timing diagrams from Chapter 2) is called a *microcycle*. The combination of control signals that are active (or at level 1) in a

microcycle determines what operation is performed in that cycle. The operation performed in a microcycle is often referred to as a *microinstruction*. The execution of each machine instruction needs one or more microcycles. Faster instructions take fewer microcycles and vice versa. Table 5.1 shows `NOP` can be executed in 1 clock cycle while the others need 2 cycles.

You may have noticed that the labels of the microcycles for excution started with a 3 instead of a 1. The reason for this will be clear soon.

5.2 Data Movement Operations

Table 5.2 lists the implementation of all data movement operations and the arithmetic/logic operations involving immediate operands. We look at the data movement instructions first. Moving from `AR` to a register is achieved using the `movd` instruction. It is quite straightforward to implement as seen in Table 5.2 and needs only one cycle to execute. Moving from a register to `AR` is achieved by loading the value onto the bus first. The ALU has an operation to pass the first or the bus argument unchanged from the input to output. If we load the register to the bus by setting the right register select value and choose the pass option of the ALU by setting the right ALU function select value, the register contents are available at the input of `AR` in the same clock cycle. If L_{AR} is also active in that clock, the data will go from the register to ALU input through the bus, pass through the ALU to `AR` and be stored into it all in one clock cycle! The design of the system with enables taking place at the rising edge and loads taking place at the falling edge makes this possible. The `movs` instruction needs only one microcycle for its execution as a result.

We look at the two data movement instructions, namely, `load` and `stor`, next. The `load` instruction reads a value from the memory to a register. The address of the memory location is given in `AR`. The memory sits outside of the processor and is accessed by giving it an address through the `MAR` register and a command through the `RD` and `WR` lines, as is appropriate. The first microcycle of execution moves the address from `AR` to `MAR` for both instructions. The next microcycle asks the memory to read the location using `RD`. We assume the value will be available before the end of the same clock cycle on the processor's internal bus. For this purpose, the register is treated like an external register file, whose address (or select) is given through `MAR`. The value from the register can then be loaded onto the specified register. In practice, the memory is significantly slower than the registers and the read cannot complete in the same clock cycle. We will ignore that aspect as we are designing a very simple processor. Thus, the data movement instructions only take 2 clock cycles for their execution on our architecture.

The third data movement operation uses an immediate argument. This is very similar to `load`

Instruction	Control Signals	Select Signals
movs <R>	Ck 3: E_{RG}, L_{AR}, End	$S_{RG} \leftarrow$ <R>, $S_{ALU} \leftarrow$ PASS0
movd <R>	Ck 3: E_{AR}, L_{RG}, End	$S_{RG} \leftarrow$ <R>
load <R>	Ck 3: E_{AR}, L_{MR} Ck 4: RD, L_{RG}, End	- $S_{RG} \leftarrow$ <R>
stor <R>	Ck 3: E_{AR}, L_{MR} Ck 4: E_{RG}, WR, End	- $S_{RG} \leftarrow$ <R>
movi <R> xx	Ck 3: E_{PC}, L_{MR}, I_{PC} Ck 4: RD, L_{RG}, End	- $S_{RG} \leftarrow$ <R>
adi xx	Ck 3: E_{PC}, L_{MR}, I_{PC} Ck 4: RD, L_{OR} Ck 5: E_{AR}, L_{AR}, End	- - $S_{ALU} \leftarrow$ ADD
sbi xx	Ck 3: E_{PC}, L_{MR}, I_{PC} Ck 4: RD, L_{OR} Ck 5: E_{AR}, L_{AR}, End	- - $S_{ALU} \leftarrow$ SUB
xri xx	Ck 3: E_{PC}, L_{MR}, I_{PC} Ck 4: RD, L_{OR} Ck 5: E_{AR}, L_{AR}, End	- - $S_{ALU} \leftarrow$ XOR
ani xx	Ck 3: E_{PC}, L_{MR}, I_{PC} Ck 4: RD, L_{OR} Ck 5: E_{AR}, L_{AR}, End	- - $S_{ALU} \leftarrow$ AND
ori xx	Ck 3: E_{PC}, L_{MR}, I_{PC} Ck 4: RD, L_{OR} Ck 5: E_{AR}, L_{AR}, End	- - $S_{ALU} \leftarrow$ OR
cmi xx	Ck 3: E_{PC}, L_{MR}, I_{PC} Ck 4: RD, L_{OR} Ck 5: E_{AR}, End	- - $S_{ALU} \leftarrow$ CMP

Table 5.2: Microinstructions to implement the data movement and arithmetic/logic instructions with an immediate operand

except for the specification of the source memory address. The source value is stored immediately along with the instruction. As we have seen before, the immediate argument `xx` is stored in a memory location with address *addr* +1 if the opcode for `movi` is stored in a memory location with address *addr*. Moreover, we assume that as the opcode for `movi` is fetched from *addr*, the `PC` value is incremented by 1 to point to the next instruction. This is done assuming that the instruction fetched needs only one memory word, which is the case with all instructions we have seen so far.

Thus, when the execution of `movi` starts, the `PC` is pointing to the word following the opcode.

This word holds the immediate operand `xx`. Thus, the situation is similar to `load`, except for the `PC` supplying the address of the operand instead of `AR`. Thus, the execution of `movi` proceeds very similarly. However, the `PC` needs to point to the next real opcode at the end of executing `movi`. We achieve this by incrementing `PC` while it is loaded onto `MAR`, by enabling the $\mathtt{I}_{\mathtt{PC}}$ control signal. Since the `PC` is implemented using an increment-capable register and given our timing, the value changes only at the falling edge of the clock. The correct value will thus be used as memory address.

The only difference between an instruction that uses a register argument and one that uses an immediate argument is the source of the argument. Earlier, we loaded the source from the selected register in one clock to `OR` through the bus (Table 5.1). We have to get it from the memory when using immediate operands, but we know that the `PC` holds the operand's address when execution starts. All arithmetic and logic instructions can be implemented keeping this in mind as shown in Table 5.2. Note that these instructions require 3 microinstructions and need 3 clock cycles each for their execution.

5.3 Instruction Fetch

We are now ready to tackle instruction fetch. We know it involves reading a word from the memory, using the `PC` value as the address. This can be achieved using the following two microinstructions.

Ck 1: $\mathtt{E}_{\mathtt{PC}}$, $\mathtt{L}_{\mathtt{MR}}$, $\mathtt{I}_{\mathtt{PC}}$
Ck 2: RD, $\mathtt{L}_{\mathtt{IR}}$

Instruction fetch requires 2 microcycles; in the first cycle, `PC` value is loaded to `MAR`. The `PC` is simultaneously incremented, so that the next fetch will be from the next word in memory. In the second cyle, the memory word at the address given by `MAR` is read and the value obtained is loaded into a special *instruction register* or `IR`. The instruction register holds the entire opcode, which then needs to be decoded or deciphered to select one of the possible actions.

Every instruction, thus, needs 2 clock cycles for fetch, before its execution can possibly begin. This does *not* depend on the specific instruction but is common for *all* instructions. Now you can understand why the labels of the microcycles of the execution phase in Tables 5.1 and 5.2 started with 3. By doing so, we can directly see how many clock cycles are needed to complete each instruction, including its fetch and execution. The total time to run a program is the sum of the times of its individual instructions.

Chapter 6

Complete Processor Architecture and Instructions

It is now time to reveal the overall architecture of our simple processor. Figure 6.1 gives the internal architecture of the entire processor. The instruction register is shown there. There are a number of new components like the flag register, the microprogram sequencer, etc. The box on the periphery is the collection all control signals. We can treat the collection as a *microinstruction word*. Such words are stored in the *microprogram memory*, which is implemented using a ROM. The address to this ROM is supplied by the *microprogram sequencer* in each clock cycle. The microprogram sequencer generates the address based on the instruction in the instruction register as well as the status of the flags. The purpose of these components will be explained in the rest of this chapter.

We will now look at the remaining few instructions to make our processor more complete. We had no branching instructions so far; all programs have to be a strictly linear sequence of instructions. That is obviously not a very desirable situation. We need the capability to branch or to break the sequential flow of instructions to implement any sort of loops. Additionally, we need the capability to branch based on some condition based on the values of registers. We use two forms of branching: one based on an immediate address and the other based on a register value. They are *conditional* and can use the status of a flag bit to take the branch or skip it.

A processor also needs instructions to invoke functions or subroutines. These are program fragments to achieve something that may be needed to be done often. These functions can be *called* from any part of the program. On completion of the function, control should *return* to the place in the program from where the function was called. The call and return are also conditional instructions.

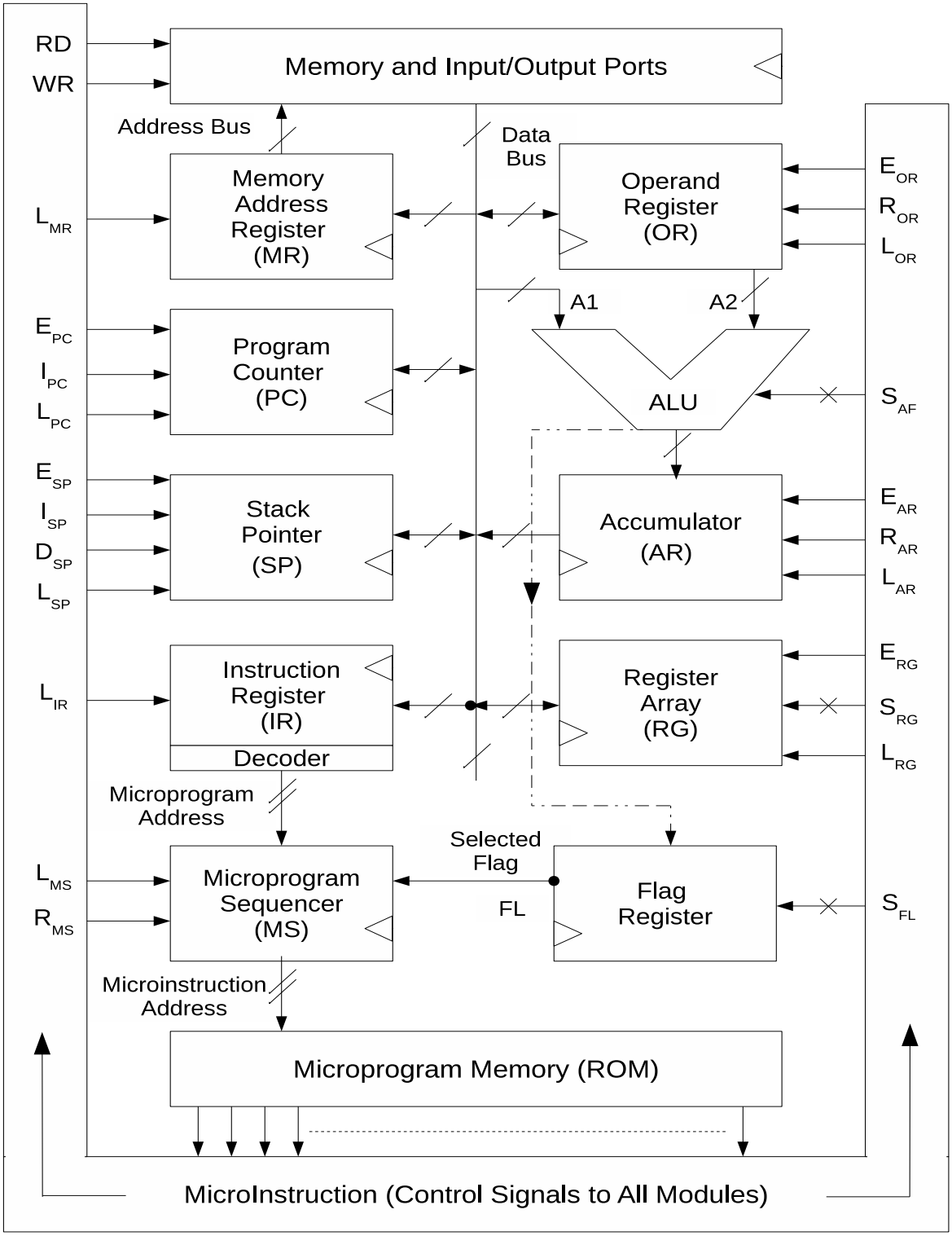

Figure 6.1: The complete architecture of the simple, singlebus processor

6.1 Branching and Stack Instructions

We now describe the remaining instructions in our processor. These are used for branching or changing the program flow and to manipulate the system stack.

6.1.1 Jump Instructions

Branching involves the shifting of the program execution from one point in the program to another. This is a change in the control flow of the program and may be used to perform different actions on the basis of the results achieved so far. Jump is a type of branching where the control is transferred absolutely, without any memory of the branching point. Branching is natural in our every day activities. For instance, each student starts off in the morning from the hostel by branching to the location of the first class. At the end of the first class, the student branches to the next. This is continues to classes, lunch, labs, etc., ending up in the hostel at the end. Each of these branching typically results in a reasonable amount of time spent at the destination. Figure 6.2 shows how program control is transferred from one place to next by jumps.

The branching may be *conditional*, based on a current state of the processor. The condition of one of the flag bits can be used for this. If the jump is conditioned on a flag bit being set, the branching happens only if that flag has a value of 1 and the program proceeds with the instruction at the branch *address*. If the flag is at state 0, execution proceeds normally with the next instruction as if the conditional branch instruction is a `nop` instruction. Table 6.1 gives the jump instructions available on our processor and their opcodes. Conditionality is shown as taking the branch only when the selected flag is 1.

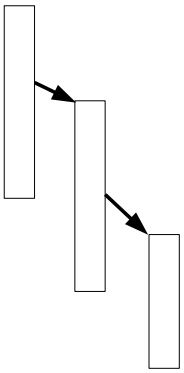

Figure 6.2: Jump results in branching to a new place in the code.

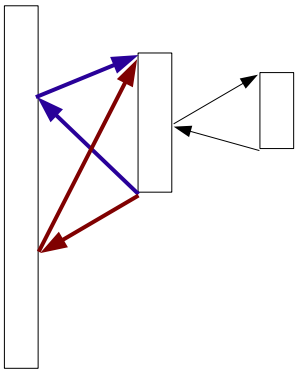

Figure 6.3: Call results in branching but with the provisioning of coming back to the same place.

The `<FL>` flag for all conditional instructions can take one of the following values: `u, z, nz, c, nc, p, m, op`. These respectively stand for *unconditional, zero, non-zero, carry, no-carry, positive, minus,* and *odd-parity*. Unconditional case is always true, irrespective of the state of the flag bits. Zero condition is true when the `Z` bit is set and the non-zero condition is true otherwise. Similarly, the carry and no-carry conditions are true when the `C` bit of the flags is 1 or 0. The plus condition is true when the sign bit `S` is 0 and the minus condition is true otherwise. The odd-parity condition is true if the flag bit `P` is 1. The flag bits are set by selected arithmetic and logic instructions and represent a limited state of the processor, as explained in Section 6.1.4.

6.1.2 Call and Return Instructions

Another type of branching instructions is used to invoke functions or subroutines. These are the *call* and the *return* instructions. A subroutine or a function is a block of code that may be invoked from any place in a program. The function performs the operations and the program proceeds from where the function was invoked. Thus, the branching to the function needs to remember the point of branching, so that it can return there at the end of the function. A *call* instruction invokes the function and a *return* instruction brings the control back to where the call happened. Both call and return may also be conditional, to be performed based on a flag bit. Figure 6.3 shows schematically how the control is transferred and later comes back to the same place when call and return are used. Note that the calls can be nested; each call has to come back to the correct place from which it is called as different colours indicate in Figure 6.3.

Call and return are familiar notions in everyday life too, since we are involved with many types of activities. Consider the scenario in which student is studying for the semester examinations that are a week away. The news comes that the marks for the last assignment are being given away. The student immediately rushes to the room where the marks are given, but only after keeping a bookmark on the page she was reading. On the way, the student gets a phone call from home which is attended to. The call places her in the mood of home but when it ends, she returns to the assignment mindset and continues with it. After the marks are obtained, the student rushes back to the hostel and resumes the studies. Similar instances in which one suspends what one is doing to attend temporarily to something are plenty in life.

How is this implemented in practice on the computer and in our minds? The state before branching has to be remembered to facilitate coming back exactly to it. This may be the page that one is reading or the address of the address of the next instruction for a program. If we need to support nested calls with correct returns, a series of *return address* values may need to be remembered. The last remembered address is the most important one, as that enables the return from the most recent call operation. It is clear that the return addresses will be used in

Assembly Instruction	Machine Code	Action
jmpd<FL> xx	E0-E7	[PC] ← xx if <FL> = 1
jmpr<FL>	E8-EF	[PC] ← [AR] if <FL> = 1
cd<FL> xx	F0-F7	[SP] ← [SP] - 1, [[SP]] ← [PC], [PC] ← xx if <FL> = 1
cr<FL>	F8-FF	[SP] ← [SP] - 1, [[SP]] ← [PC], [PC] ← [AR] if <FL> = 1
ret<FL>	08-0F	[PC] ← [[SP]], [SP] ← [SP] + 1 if <FL> = 1
push <R>	C0-CF	[SP] ← [SP] - 1, [[SP]] ← [<R>]
pop <R>	D0-DF	[<R>] ← [[SP]], [SP] ← [SP] + 1

Table 6.1: Branching and stack instructions, opcodes, and action

the *reverse* order from which they are saved. One can imagine stacking up the return addresses on a single pile, with the latest one placed on the top. This is like a vertical pile of books with all new additions taking place at the top of the pile. Removals are also performed one at a time and from the top of the pile. Such a structure is called a *stack* and plays and important role in function invocation on all modern processors.

A call instruction, thus, remembers the point to return to by placing the address of the next instruction in the stack. The address of the next instruction is available in PC and thus, the PC has to be added to the top of stack. The top of the stack needs to be adjusted as more additions may happen to the stack we use up the most recent value. The address to branch to is given by the call instruction. The value on the top of the stack is used as the address from which the next instruction is fetched when returning. This value is loaded onto the PC and the top of stack is adjusted to reflect this fact. The operation that adds a new element to the stack is called *push* and the operation that removes an element is called *pop*. Table 6.1 lists the call and return instructions and their opcodes.

6.1.3 Stack Manipulation Instructions

The stack is an important component of the program execution aspect of the processor. The stack is stored in the memory so that its capacity can be reasonably large. However, a special register called the *stack pointer (SP)* resides inside the processor, which holds the address of the next empty location in the stack. Figure 6.1 shows the stack pointer among the registers of the processor. Traditionally, SP values grown downwards. That is, SP is decremented by 1 when a value is pushed onto it and it is incremented by 1 when something is popped out of it. Thus, SP is initialized to the highest address that belongs to the stack when it is empty. The stack managed by SP is usually called the *system stack* and plays a critical role in the program flow

involving functions and subroutines.

We may need to add more elements to the stack in addition to the return address. Two instructions – `push` and `pop` – achieve that. These are like data movement operations, but one end of the movement is always the stack. These will complete the simple instruction set for our simple processor. Before looking at the implementation of these instructions, we discuss the flag bits of our processor. Table 6.1 lists the stack manipulation instructions and their opcodes.

6.1.4 Flags and Program Status Word

The current state of a processor is maintained in the form of a number of *flag* bits or flip-flops. These flags store limited history of the results of the computations performed by the processor. This may include aspects like: Did the last arithmetic operation result in an overflow? Did it result in a carry from the most significant bit? Was the result of the previous operation a zero? These, in conjunction with branching, are essential to control the program based on the results of operations. For example, if we want to run a loop ten times, we can repeat the code 10 times, which makes the code long. It also allows no flexibility to run the code 12 times, if we desire it. An alternative is to use a count (typically stored in a register) that is initialized to 10. After one set of computations is over, the count can be reduced by 1. The program can branch to the start of the computations if the count is still not zero. It is clear that the second option results in shorter code. Even better, if the count is initialized to 12 or 25, the code remains exactly the same.

Our simple processor has the following 4 flag bits: *zero, carry, sign,* and *parity*, with respective flags `Z, C, S,` and `P`. The zero flag is set if the previous ALU operation produced an exact `0` as the result. Similarly, the carry flag is set if the previous operation resulted in a carry-out or borrow-in from the most significant bit. The `S` bit copies the sign bit of the last arithmetic operation and becomes 1 if the result was negative. The parity bit counts the number of 1 bits in the result of the last operation. If that number is odd, the parity bit is 1 and vice versa. (Thus, the number of 1s in the last result plus the `P` flag will always be an even number.)

Figure 6.1 shows a thin line connecting the ALU to the flag register to indicate their link. The flag values are loaded from the ALU to the register only for valid operations. For instance, all arithmetic and logic instructions affect the value of `Z` and `P` flags. However, only arithmetic instructions affect the values of `S` and `C`. Instructions that do not use the ALU – such as the data movement instructions, branching instructinos, and the like – do not change the flag values. As we will see soon, the conditional branching instructions can use one of the flag values to decide if the branch should be taken or not. The select lines S_{FL} choose the appropriate flag.

Some processors group all flags into a special register word known as the *Program Satus Word* (PSW). Special instructions may move the PSW to or from internal registers or memory. This allows their manipulation as data.

6.2 Implementing the Branching and Stack Instructions

Instruction	Control Signals	Select Signals
jumpd<FL> xx	Ck 3: E_{PC}, L_{MR}, I_{PC}, E_{FL}, End if <FL>' Ck 4: RD, L_{PC}, End	$S_{FL} \leftarrow$ <FL> -
jmpr<FL>	Ck 3: E_{FL}, End if <FL>' Ck 4: E_{AR}, L_{PC}, End	$S_{FL} \leftarrow$ <FL> -
cd<FL> xx	Ck 3: E_{PC}, L_{MR}, I_{PC}, E_{FL}, End if <FL>' Ck 4: RD, L_{OR}, D_{SP} Ck 5: E_{SP}, L_{MR} Ck 6: E_{PC}, WR Ck 7: E_{OR}, L_{PC}, End	$S_{FL} \leftarrow$ <FL> - - - -
cr<FL>	Ck 3: E_{FL}, End if <FL>' Ck 4: D_{SP} Ck 5: E_{SP}, L_{MR} Ck 6: E_{PC}, WR Ck 7: E_{AR}, L_{PC}, End	$S_{FL} \leftarrow$ <FL> - - - -
ret<FL>	Ck 3: E_{FL}, End if <FL>' Ck 4: E_{SP}, L_{MR}, I_{SP} Ck 5: RD, L_{PC}, End	$S_{FL} \leftarrow$ <FL> - $S_{RG} \leftarrow$ <R>
push <R>	Ck 3: D_{SP} Ck 4: E_{SP}, L_{MR} Ck 5: E_{RG}, WR, End	- - $S_{RG} \leftarrow$ <R>
pop <R>	Ck 3: E_{SP}, L_{MR}, I_{SP} Ck 4: RD, L_{RG}, End	- $S_{RG} \leftarrow$ <R>

Table 6.2: Microinstructions to implement the branching and stack instructions

Table 6.2 gives the implementation of the branching and stack instructions presented in this chapter in terms of the available control signals. There are several poins to be noted.

The push and pop instructions are very similar to memory write and read instructions, but with SP supplying the address. Since SP points to the element at the top of the stack, it has to be decremented before being used as an address. Since the change in value (like decrement) takes place at the falling edge of the clock, we need to use a whole microcycle to just decrement SP

for the `push` instruction. This was not the case for the `pop` instruction. We could have used a special register for `SP` to avoid this problem, with decrement happening at the rising edge and increment at the falling edge. Our goal as microinstruction designers is to ensure the instruction is implemented correctly.

The conditional jump instructions use a modified control signal, labelled "`End` if `<FL>'`". This means that the `End` control signal is activated only if the selected flag is at a 0 level. This is the case where the condition is *not* satisfied and hence the instruction has no impact. However, for instructions with immediate operands (such as `jumpd` and `cd`), the next word has to be jumped over so that the `PC` points to the next real instruction. At the same time, the value of `SP` should not be affected if the selected flag is not true! These requirements have made the `cd` and `cr` instructions long as can be seen in Table 6.2.

The `cd` instruction has the (undesirable ?) side effect of copying the address of the immediate operand to the `MAR` and `OR` registers, even if the branch is not taken. Fortunately, this is not a serious problem as these registers are not visible to the assembly language programmer. The `cr` and `cd` instructions are the longest instruction needing 5 clock cycles each to execute and 7 clock cycles in total. Clocks 3 and 4 of the `cr` instruction are not even using the bus. However, there is no way to reduce the number of clocks needed due to the dependencies of the results of one microcyle on later ones.

Chapter 7

Microinstructions and Microprogram Sequencing

In this chapter, we will complete the story of our simple processor, whose architecture is given in Figure 6.1 and whose complete instruction set is described in Table 7.1. It is clear that the main task is to generate the combination of control signals for each clock cycle for each instruction as given in the table. We can think of the control signals being generated using a combinational circuit, whose input is the opcode of the current instruction and the clock cycle number. The opcode is loaded into the `IR` register by the fetch process. The clock is available to the microprogram sequencer, which can count them starting with a 1 for the fetch.

The combinatational circuit can be implemented using a ROM, to generate all control signals. The word-width of the ROM equals the number of control signals used by the processor. The number of words in the ROM should exceed the number of distinct input combinations. The

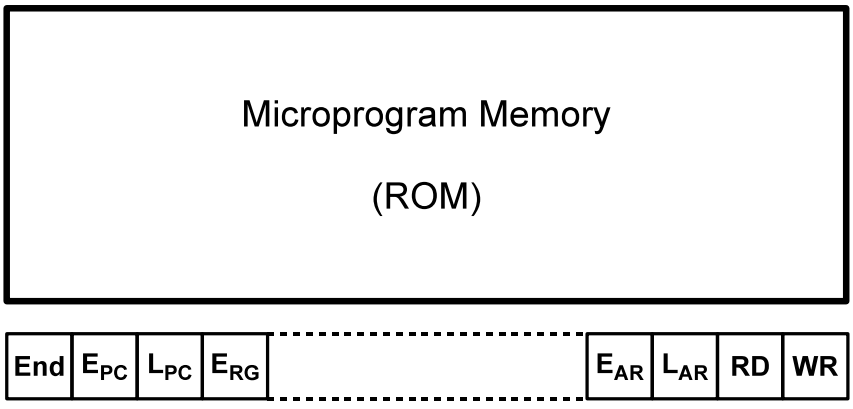

Figure 7.1: The microprogram word supplies each of the control signals directly

Instruction	Opcode	Clk	Control Signals	Select Signals
Fetch	-	**1**	E_{PC}, L_{MR}, I_{PC}	-
		2	RD, L_{IR}, L_{MS}	-
nop	00	**3**	End	-
adi xx	01	**3**	E_{PC}, L_{MR}, I_{PC}	-
		4	RD, L_{OR}	-
		5	E_{AR}, L_{AR}, End	$S_{ALU} \leftarrow$ ADD
sbi xx	02	**3**	E_{PC}, L_{MR}, I_{PC}	-
		4	RD, L_{OR}	-
		5	E_{AR}, L_{AR}, End	$S_{ALU} \leftarrow$ SUB
xri xx	03	**3**	E_{PC}, L_{MR}, I_{PC}	-
		4	RD, L_{OR}	-
		5	E_{AR}, L_{AR}, End	$S_{ALU} \leftarrow$ XOR
ani xx	04	**3**	E_{PC}, L_{MR}, I_{PC}	-
		4	RD, L_{OR}	-
		5	E_{AR}, L_{AR}, End	$S_{ALU} \leftarrow$ AND
ori xx	05	**3**	E_{PC}, L_{MR}, I_{PC}	-
		4	RD, L_{OR}	-
		5	E_{AR}, L_{AR}, End	$S_{ALU} \leftarrow$ OR
cmi xx	06	**3**	E_{PC}, L_{MR}, I_{PC}	-
		4	RD, L_{OR}	-
		5	E_{AR}, End	$S_{ALU} \leftarrow$ CMP
stop	07	**3**	End, StopClock	-
ret<FL>	08-0F	**3**	E_{FL}, End if <FL>'	$S_{FL} \leftarrow$ <FL>
		4	E_{SP}, L_{MR}, I_{SP}	-
		5	RD, L_{PC}, End	$S_{RG} \leftarrow$ <R>
add <R>	10-1F	3	E_{RG}, L_{OR}	$S_{RG} \leftarrow$ <R>
		4	E_{AR}, L_{AR}, End	$S_{ALU} \leftarrow$ ADD
sub <R>	20-2F	**3**	E_{RG}, L_{OR}	$S_{RG} \leftarrow$ <R>
		4	E_{AR}, L_{AR}, End	$S_{ALU} \leftarrow$ SUB
xor <R>	30-3F	**3**	E_{RG}, L_{OR}	$S_{RG} \leftarrow$ <R>
		4	E_{AR}, L_{AR}, End	$S_{ALU} \leftarrow$ XOR

Instruction	Opcode	Clk	Control Signals	Select Signals
and <R>	40-4F	**3**	E_{RG}, L_{OR}	$S_{RG} \leftarrow$ <R>
		4	E_{AR}, L_{AR}, End	$S_{ALU} \leftarrow$ AND
or <R>	50-5F	**3**	E_{RG}, L_{OR}	$S_{RG} \leftarrow$ <R>
		4	E_{AR}, L_{AR}, End	$S_{ALU} \leftarrow$ OR
cmp <R>	60-6F	**3**	E_{RG}, L_{OR}	$S_{RG} \leftarrow$ <R>
		4	E_{AR}, End	$S_{ALU} \leftarrow$ CMP
movs <R>	70-7F	**3**	E_{RG}, L_{AR}, End	$S_{RG} \leftarrow$ <R>, $S_{ALU} \leftarrow$ PASS0
movd <R>	80-8F	**3**	E_{AR}, L_{RG}, End	$S_{RG} \leftarrow$ <R>
movi <R> xx	90-9F	**3**	E_{PC}, L_{MR}, I_{PC}	-
		4	RD, L_{RG}, End	$S_{RG} \leftarrow$ <R>
stor <R>	A0-AF	**3**	E_{AR}, L_{MR}	-
		4	E_{RG}, WR, End	$S_{RG} \leftarrow$ <R>
load <R>	B0-BF	**3**	E_{AR}, L_{MR}	-
		4	RD, L_{RG}, End	$S_{RG} \leftarrow$ <R>
push <R>	C0-CF	**3**	D_{SP}	-
		4	E_{SP}, L_{MR}	-
		5	E_{RG}, WR, End	$S_{RG} \leftarrow$ <R>
pop <R>	D0-DF	**3**	E_{SP}, L_{MR}, I_{SP}	-
		4	RD, L_{RG}, End	$S_{RG} \leftarrow$ <R>
jumpd<FL> xx	E0-E7	**3**	E_{PC}, L_{MR}, I_{PC}, E_{FL}, End if <FL>'	$S_{FL} \leftarrow$ <FL>
		4	RD, L_{PC}, End	-
jmpr<FL>	E8-EF	**3**	E_{FL}, End if <FL>'	$S_{FL} \leftarrow$ <FL>
		4	E_{AR}, L_{PC}, End	-
cd<FL> xx	F0-F7	**3**	E_{PC}, L_{MR}, I_{PC}, E_{FL}, End if <FL>'	$S_{FL} \leftarrow$ <FL>
		4	RD, L_{OR}, D_{SP}	-
		5	E_{SP}, L_{MR}	-
		6	E_{PC}, WR	-
		7	E_{OR}, L_{PC}, End	-
cr<FL>	F8-FF	**3**	E_{FL}, End if <FL>'	$S_{FL} \leftarrow$ <FL>
		4	D_{SP}	-
		5	E_{SP}, L_{MR}	-
		6	E_{PC}, WR	-
		7	E_{AR}, L_{PC}, End	-

Table 7.1: The complete set of instructions and their microinstructions in the opcode order.

format of the microprogram word is shown in Figure 7.1, with each bit of the ROM output directly serving as the control input line. The set of control signals treated as a word is often referred to as the *microprogram word* or the *control word*. Each microinstruction or clock cycle of each instruction maps to a separate microword, typically.

Table 7.1 can be thought of as encoding the contents of the micropogram memory! The signals mentioned in the table will be active or set (to a 1 level) for the corresponding clock cycle with all others being inactive or at a 0 level. Thus, Clock 4 of the `push` instruction has $\texttt{E}_{\texttt{SP}}$ and $\texttt{L}_{\texttt{MR}}$ as active. The corresponding microword will have the bits for these signals as 1 with the rest 0. Similarly for all clock cycles for all instructions. We will store the microwords of the same instruction in consecutive locations. For example, both microwords of the `or` instruction will be in consecutive locations. So will all 5 microwords of the `cd` instruction. The microprogram memory will therefore consist of a minimum of 64 locations, as Table 7.1 has that many rows. This will require all microwords to be packed together tightly.

Once the control words are laid out in the microprogram memory with consecutive microinstructions of each instruction stored in consecutive microwords, what is remaining is to map the opcode of each instruction to its starting address. Assuming the starting address of each instruction is its row number (starting with 0) in Table 7.1, the 8-bit opcode needs to be mapped to a 6-bit starting address. This is performed by the *decoder* shown along with the instruction register in Figure 6.1.

The starting microprogram address is loaded onto the microprogram sequencer using $\texttt{L}_{\texttt{MS}}$. This will result in the first microinstruction corresponding to the current opcode to be taken up in the following clock cycle. Since we have stored the microinstructions for each instruction in consecutive locations of the microprogram memory, the *microprogram sequencer* in Figure 6.1 is a register which gets incremented by 1 every clock cycle at the falling edge. This continues till the last microinstruction of each opcode. We issue a signal `End` to denote that, which indicates that the execution of the instruction is completed. The processor is to proceed to fetching the next instruction. Fetch corresponds to a sequence of microinstructions starting at address 0. Thus, going to the fetch phase of the next instruction can be achieved by loading 0 to the microprogram sequencer. We can do that if we use the signal `End` to *reset* the microprogram sequencer, making it a register with load (using $\texttt{L}_{\texttt{MS}}$), reset (using $\texttt{R}_{MS}$, and increment (when neither of the above signals is active)!

The microinstructions for instruction fetch is stored starting address 0

Chapter 8

Simulator

-

Chapter 9

Conclusions

-