The Official
DIY Calculator Data Book

Clive "Max" Maxfield and Alvin Brown

Contents

Chapter 1

The Virtual Microcomputer Powering the *DIY Calculator*

Microprocessors and microcomputers

In the 1800s, mathematical tables such as logarithmic and trigonometric functions were generated by teams of mathematicians working day and night on primitive mechanical calculators. Due to the fact that these people performed *computations* they were referred to as *computers*. Over the course of time, however, the term “computer” became associated with machines that could perform computations automatically.

Although we typically think of computers in the context of electronic implementations based on silicon chips, they can, in fact, be realized in a variety of ways, including mechanical, hydraulic, and pneumatic systems. Furthermore, computers may be analog or digital in nature (or, in some cases, a hybrid of both). Analog information represents a continuously varying quantity, such as a light controlled by a dimmer switch; while digital information represents a quantity that can be considered to be in one of a number of discrete states, or *quanta*, such as a traditional light switch which is either OFF or ON. This data book focuses on digital electronic implementations, because these account for the overwhelming majority of today’s computing systems.

In its broadest sense, a computer is a device that can accept information from the outside world; process that information using logical and mathematical operations; make decisions based on the results of this processing; and, ultimately, return the processed information to the outside world in its new form.

The "brain" of the computer is its *central processing unit (CPU)*, which is where all of the number crunching and decision making is performed. The CPU communicates with other components in the system using three groups of signals called the *control bus*, *address bus*, and *data bus*. The *read-only memory (ROM)* and *random access memory (RAM)* devices are used to store programs, data, and intermediate results, while the input and output ports allow the CPU to communicate with the outside world (Figure 1-1).

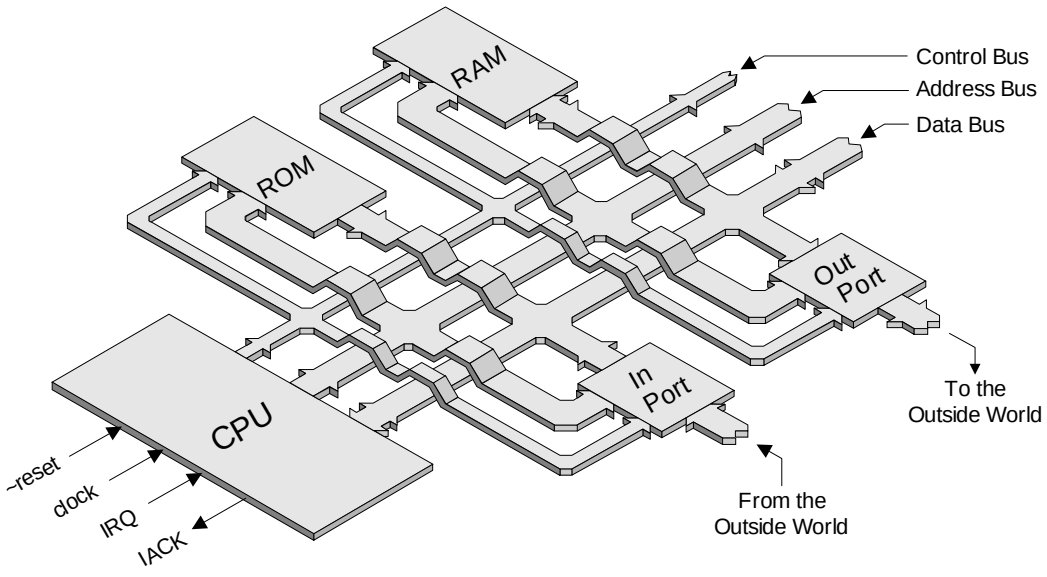


Figure 1-1. The main components forming a simple microcomputer

In today's terminology, a *microprocessor* is commonly accepted to be a CPU implemented as a single integrated circuit, while a *microcomputer* is a computer using a microprocessor as its CPU. This data book describes the microprocessor and microcomputer used to power the *DIY Calculator*. One unique aspect of the *DIY Calculator* is that it is implemented as a virtual machine, which is delivered on a CD-ROM accompanying the book *How Computers Do Math*.

The rest of this chapter describes the microcomputer at the system level, while the remainder of this data book concentrates on the CPU.

Logic 0 and Logic 1

Before we proceed, we should note that digital electronic computers are constructed from large numbers of primitive logic gates and functions, each of which is formed from a group of transistor switches. These switches can be in one of two states – **OFF** or **ON** – which physically correspond to two different voltage levels. For the purposes of these discussions, however, we have little interest in the actual voltages used. Furthermore, the terms **OFF** and **ON** are not particularly relevant or useful in this context. Thus, as opposed to thinking in terms of voltage levels or in terms of **OFF** and **ON**, we generally use more abstract terms called *logic 0* (or "false") and *logic 1* (or "true").

Bits, Bytes, and Nybbles

Sometime in the late 1940s, the American chemist, topologist, and statistician John Wilder Tukey realized that digital computers and the binary number system were destined to become increasingly important. In addition to coining the word "software," Tukey decided that saying "binary digit" was a bit of a mouthful, so he started to look for an alternative. He considered a variety of options – including *binit* and *bigit* – but eventually settled on *bit*, which is elegant in its simplicity and is used to this day.

Binary values of 1100_2 and 11001110_2 would be said to be four and eight bits wide, respectively. Groupings of four bits are relatively common, so they are given the special name of *nybble* (or sometimes *nibble*). Similarly, groupings of eight bits are also common, so they are given the special name of *byte*. Thus, *"two nybbles make a byte,"* which goes to show that computer engineers do have at least a rudimentary sense of humor.

The *DIY Calculator's* CPU

The CPU has four primary control signals: `clock`, `~reset`, `IRQ`, and `IACK`, which are more clearly illustrated in Figure 1-2.

An external *clock generator* supplies the clock signal, which switches back and forth between two voltage levels. This signal drives the CPU's clock input and is used to synchronize its internal and external actions (signal relationships and timing diagrams are presented in *Appendix C*).

Now consider the circuit connected to the `~reset` input in Figure 1-2. This signal's active state is a logic 0 value, which is indicated by the tilde "~" character in its name. The switch **S** is usually open, which means that the `~reset` input is connected to a logic 1 value through the resistor **R**. When the switch is closed, it connects the `~reset` input directly to a logic 0, which

causes the CPU to be initialized into a well-known state. The switch is of a type that springs open when released, thereby returning `~reset` to a logic 1 and freeing the CPU to start to perform its magic.

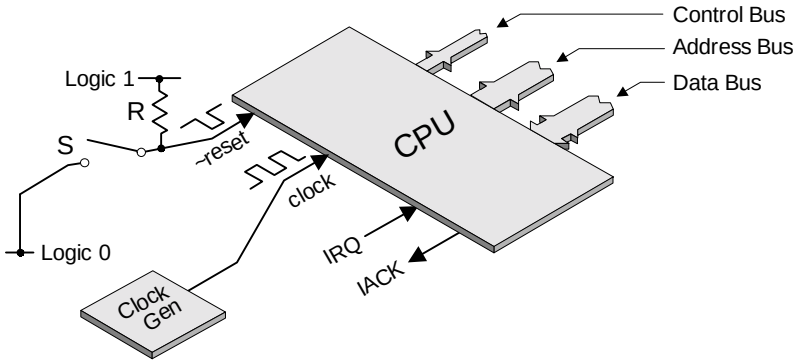


Figure 1-2. The CPU has four primary control signals

Sometimes even the best computer becomes hopelessly lost and confused; this is usually due to programming errors caused by its human operators. In this case, the system may be reinitialized by means of the `~reset` input as discussed above. Also, an additional circuit (not shown here) is used to automatically apply a logic 0 pulse to the `~reset` input when power is first applied to the system. This is referred to as a *power-on reset*.

The remaining control signals – `IRQ` ("*interrupt request*") and `IACK` ("*interrupt acknowledge*") – are discussed in more detail in *Chapter 3*.

The data bus

The term *bus* is used to refer to a group of signals which carry similar information and perform a common function. The *DIY Calculator's* data bus is 8-bits wide (Figure 1-3).

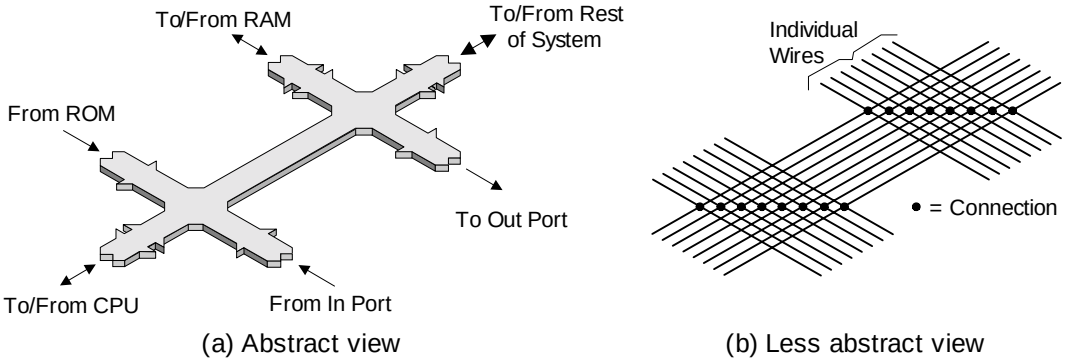


Figure 1-3. The *DIY Calculator's* data bus

The more abstract view of the data bus shows arrows indicating the directions in which signals can travel between the various components attached to the bus. The less abstract view omits these arrows to emphasize the fact that the bus is physically composed of simple wires. In reality, the only thing that determines which way signals are traveling on the bus at any particular

time is the CPU (see also the discussions on the *control bus* later in this chapter. Due to the fact that signals can travel in either direction on the data bus, this bus is said to be *bidirectional*.

The address bus

The totality of memory locations that can be addressed by a computer are referred to as its *address space*. The *DIY Calculator's* address bus is 16-bits wide, which allows it to address 2^{16} = 65,536 memory locations numbered from 0 to 65,535 (or $0000 to $FFFF in hexadecimal, where dollar "$" characters are used to indicate hexadecimal values) (Figure 1-4).

Each location in the memory is referred to as a *word*, and each word has the same width as the data bus. Thus, as the *DIY Calculator's* data bus is 8-bits wide, each word in the memory must also be 8-bits wide. Each bit in a memory word can be used to store a logic 0 or a logic 1, and all of the bits forming a word are typically written to or read from simultaneously.

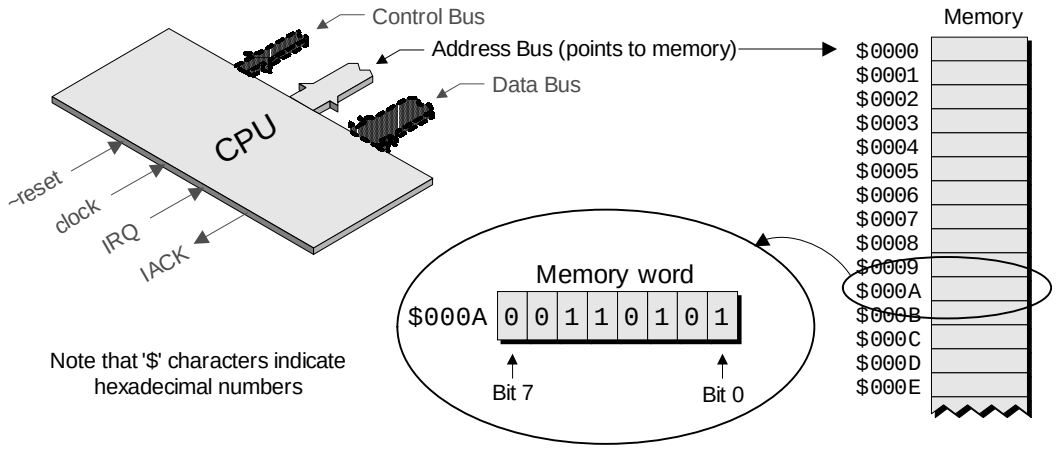


Figure 1-4. The *DIY Calculator's* address bus

The control bus

Paradoxically – although its name would appear to indicate something rather special – in some respects the *control bus* is the simplest bus of all, because it consists of only two wires named `~read` and `~write` (Figure 1-5).

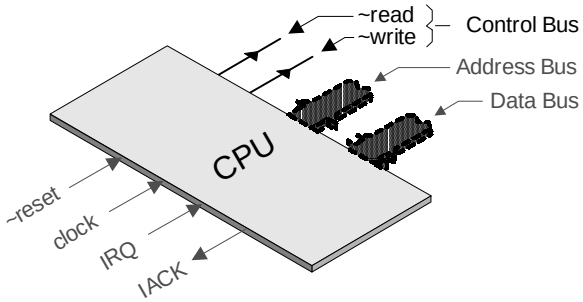


Figure 1-5. The *DIY Calculator's* control bus

In many systems these two signals are combined on a single wire, in which case a logic 1 is used to indicate a read operation and a logic 0 is used to indicate a write (or vice versa). To be perfectly honest, the `clock`, `~reset`, `IRQ`, and `IACK` signals are also considered to form part of the control bus, but these signals are often treated separately as is illustrated in our diagrams.

Both the `~read` and `~write` signals are active when they carry logic 0 values, as is indicated by the tilde "~" characters in their names. The CPU uses its `~read` signal to indicate when it wishes to receive (read) some data from whichever memory location it is currently pointing to with its address bus. The location selected by the address bus passes the required data to the CPU by means of the data bus. Similarly, the CPU uses its `~write` signal to indicate when it wishes to send (write) some data to whichever memory location it is currently pointing to with its address bus. The CPU passes the data to the targeted location by driving it out onto the data bus.

kB, KB, kb, Kb, etc.

The use of the metric system of measurement – which was developed during the French Revolution – was legalized in America in 1866. The *International System of Units (SI)* is a modernized version of the original metric system. Four of the SI prefixes that are of interest to us here are as shown in Table 1-1.

Name (SI)	Symbol	Factor	Name (USA)	Comment
tera	T	10^{12}	trillion	One million million
giga[1]	G	10^{9}	billion[4]	One thousand million[4]
mega[2]	M	10^{6}	million	
kilo[3]	k	10^{3}	thousand	

[1]The term *giga* comes from the Latin *gigas*, meaning "giant."

[2]The term *mega* comes from the Greek *mega* meaning "great" (hence the fact that Alexander the Great was known as *Alexandros Megos* to his contemporaries).

[3]The term *kilo* comes from the Greek *khiloi*, meaning "thousand" (strangely enough, this is the only prefix with an actual numerical meaning).

[4]In Britain, the term *"billion"* traditionally used to mean *"a million million"* (10^{12}). However, for reasons unknown, the Americans decided that *"billion"* should denote *"a thousand million"* (10^{9}). In order to avoid the confusion that would otherwise ensue, most countries (including Britain) have decided to go along with the Americans on this one.

Table 1-1. Four SI units of interest

This leads to an interesting quirk when referencing the size of memory and storage devices in a computer. In SI units, the qualifier **k** (kilo) represents one thousand (1,000), but computers are based on the binary number system and the closest power of two to one thousand is 2^{10}, which equals 1,024. Therefore a *1 kilobit* (1 kb or 1 Kb) memory actually refers to a device containing 1,024 bits, while a *1 kilobyte* (1 kB or 1 KB) memory refers to a component containing 1,024 bytes.

Similarly, the qualifier **M** (mega) is generally taken to represent one million (1,000,000), but the closest power of two to one million is 2^{20}, which equals 1,048,576. Thus, a *1 megabit* (1 Mb) memory actually refers to a device containing 1,048,576 bits, while a *1 megabyte* (1 MB) memory refers to a component containing 1,048,576 bytes.

Meanwhile, in the case of the qualifier **G** (giga), which is commonly understood to refer to one thousand million (1,000,000,000), the closest power of two is 2^{30}, which equals 1,073,741,824. This means that a *1 gigabit* (1 Gb) memory actually refers to a device containing 1,073,741,824 bits, while a *1 gigabyte* (1 GB) memory refers to a component containing 1,073,741,824 bytes.

Note the use of 'b' and 'B' to represent *bit* and *byte*, respectively. Also note that 'M' (mega) and 'G' (giga) are always uppercase, but people may use 'k' or 'K' to represent "kilo" (the lowercase 'k' is technically more correct, because this more closely matches the SI standard, but folks often use the uppercase 'K' because this maintains consistency with 'M' and 'G'.) Last, but not least, a useful rule to remember is that no space is used between a numerical quantity and any qualifying symbol for single letter qualifiers (for example, 5V, meaning "5 volts"), but spaces *are* used for multi-letter qualifiers (for example, 4 KB, meaning "4 kilobytes").

The memory (RAM and ROM)

For the purposes of these discussions, all of the memory devices in the *DIY Calculator* are assumed to be 4 KB in size (that is, they each contain 4,096 byte-sized words of data). Although their internal construction is quite different, ROM and RAM devices are very similar in external appearance (Figure 1-6).

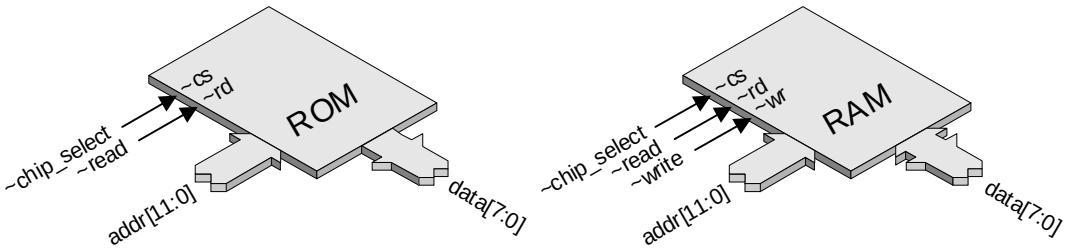


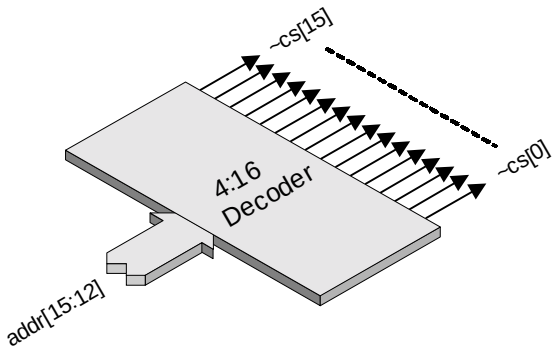
Figure 1-6. ROM and RAM devices

Both ROMs and RAMs have a `~cs` input, which is used to inform them when their services are required (where "cs" is a common abbreviation for "chip select"). Similarly, both devices have an `~rd` (*"read"*) input, which informs them when the CPU wishes to perform a read operation. However, only the RAM has a `~wr` (*"write"*) input, which is used to inform it when the CPU wishes to perform a write operation (the ROM does not have this input, because you can't write new data into a read-only memory). As the tilde "~" characters in their names would suggest, the active states for the `~cs`, `~rd`, and `~wr` inputs are logic 0s, which is commonly the case for control signals.

Memory address decoding

Remember that all of the ROM and RAM devices in the *DIY Calculator* are assumed to have a depth of 4 KB. Now let's assume that we've got a bucket of these 4 KB memory devices and we want to connect them together in such a way that – to the CPU – they appear to be a contiguous 64 KB memory. (You will recall that 1 KB actually equals 1,024 bytes, so 64 KB equates to the 2^{16} = 65,536 byte-sized words that can be addressed by the *DIY Calculator's* 16-bit address bus).

The way this works is that all of the devices are going to be connected to the least-significant twelve bits of the address bus: `addr[11:0]` (where these twelve bits can be used to address the 2^{12} = 4,096 locations contained in each of the 4 KB memory devices). The remaining four address bus bits, `addr[15:12]`, are going to be used to select between the individual memory devices. Based on the fact that four bits can be used to represent sixteen different combinations of 0s and 1s, what we're going to do is to use a 4:16 decoder (Figure 1-7).



addr[15:12]	~cs[15:0]
0000	1111111111111110
0001	1111111111111101
0010	1111111111111011
0011	1111111111110111
0100	1111111111101111
0101	1111111111011111
0110	1111111110111111
0111	1111111101111111
1000	1111111011111111
1001	1111110111111111
1010	1111101111111111
1011	1111011111111111
1100	1110111111111111
1101	1101111111111111
1110	1011111111111111
1111	0111111111111111

Figure 1-7. The 4:16 decoder

This device has four inputs connected to `addr[15:12]`, and sixteen outputs connected to a set of wires called `~cs[15:0]` (as usual, the tilde "~" characters in their names indicate that the active states of these outputs are logic 0s). Each pattern of 0s and 1s on the inputs causes an individual output to be driven to its active state, and each of these outputs is used to select a particular memory device. The only thing that remains to be done is to connect the decoder to sixteen of our 4 KB memory devices in such a way that the CPU is fooled into thinking that it is looking at a single 64 KB chunk of memory (16 × 4 KB = 64 KB) (Figure 1-8).

As we previously noted, the four most-significant bits of the address bus, `addr[15:12]`, are fed into our 4:16 decoder, and each of the outputs from the decoder can be used to select a 4 KB ROM or RAM device. (In fact, we're actually only going to use the `~cs[14:0]` outputs to select memory devices, because we're saving `~cs[15]` to select the input and output ports.) The remaining address bits, `addr[11:0]`, which are connected to all of the memory devices, are used to point to a particular word within whichever device is selected by the decoder. Due to lack of space, Figure 1-8 only shows three ROM devices – the remaining ROMs and RAMs would be connected in a similar fashion. As we noted earlier, the `~read` signal from the CPU is connected to all of the ROMs and RAMs, but the `~write` signal would only be connected to the RAMs (because, by definition, we can't write data into a "read-only" memory).

The main point to note is that the CPU doesn't know anything about the tricks we're playing with multiple memory devices and our decoder. As far as *it's* concerned, the address bus appears to be pointing to 64 Kbytes of contiguous memory, which means that the CPU is as happy as a clam.

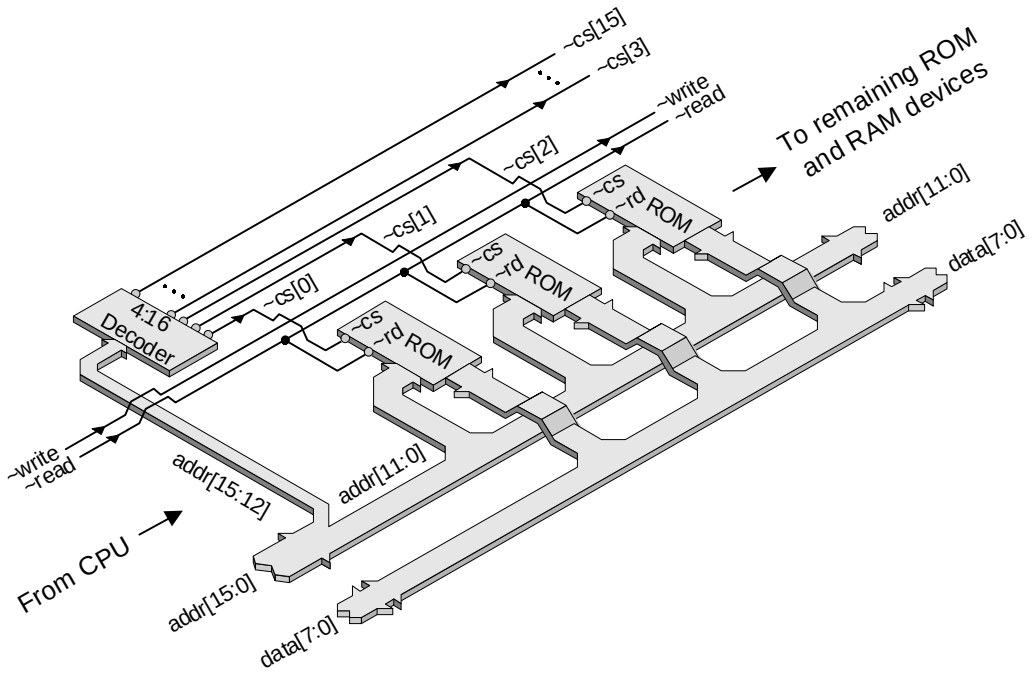


Figure 1-8. Memory address decoding

The memory map

A common method for representing the way in which a computer's memory is organized is by means of a diagram called a *memory map* (Figure 1-9).

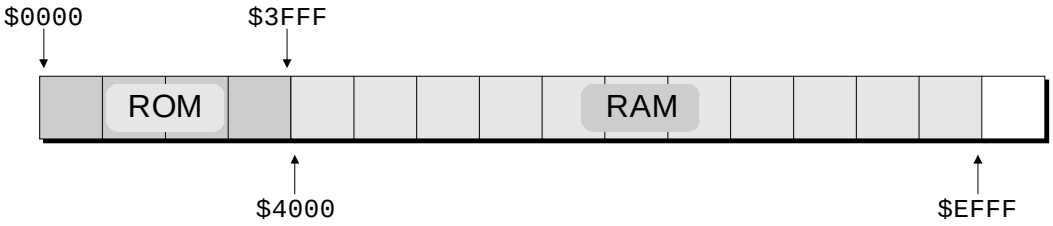


Figure 1-9. A simplified version of the *DIY Calculator's* memory map

From this memory map, we see that the *DIY Calculator* has 16 Kbytes of ROM (in four 4 KB chunks) starting at address $0000. Following the ROM there are 44 Kbytes of RAM (in eleven 4 KB chunks) starting at address $4000.

Observe that the last 4 KB chunk of the map doesn't actually contain any memory. How can this be? Well if you rewind your brain to the beginning of this chapter (Figure 1-1), you will recall our mentioning the *input ports* and *output ports* that the system uses to communicate with the outside world. There are a variety of ways in which the system can be configured to "talk" to its *input/output (I/O)* ports, but one of the most common is known as *memory-mapped I/O*, in which we fool the CPU into believing that these ports are actually standard memory locations. Theoretically, we could use all 4,096 locations at the top of the memory map as input and output ports, but this would be somewhat excessive to say the least.

In fact, addresses $F000 through $F01F are occupied by a set of thirty-two input ports (of which the *DIY Calculator's* front panel uses only one port at address $F011), while addresses $F020 through $F03F are occupied by a set of thirty-two output ports (of which the front panel employs only two ports at addresses $F031 and $F032) (Figure 1-10).

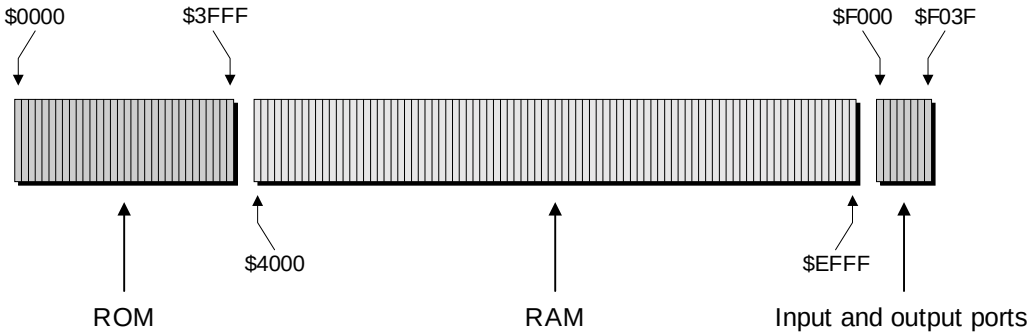


Figure 1-10. A more sophisticated representation of the memory map

Once again, it's important to note that the CPU regards these I/O port addresses as being standard memory locations and doesn't realize we're using them for our own cunning purposes. Finally, addresses $F040 through $FFFF are unused in this implementation (Table 1-2).

Addresses	Function
$0000 to $3FFF	ROM
$4000 to $EFFF	RAM
$F000 to $F01F	Input ports
$F020 to $F03F	Output ports
$F040 to $FFFF	Unused

Table 1-2. Summary of memory allocation

Input ports

As their name suggests, *input ports* allow the system to access information from the outside world. First consider a simple input port (Figure 1-11).

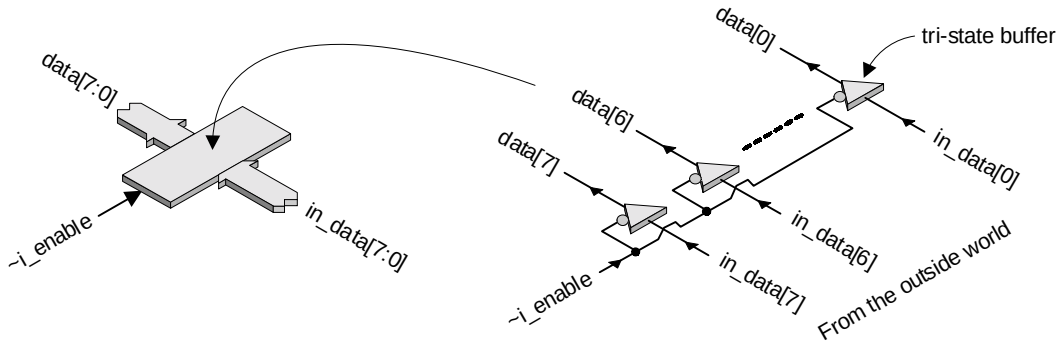


Figure 1-11. A simple 8-bit input port

Note that our input ports are 8-bits wide to match the *DIY Calculator's* data bus. The input signals driving the port, `in_data[7:0]`, would be connected to something in the outside world; for example, switches connected in such a way that they can represent either logic 0 or logic 1 values. The outputs from the port, `data[7:0]`, are connected to the system's data bus. Inside the port are logic gates known as *tri-state buffers*, one for each of the data signals. These buffers act in such a way that when their control inputs are in their active state (which is a logic 0 in this case), they pass whatever is on their data inputs through to their data outputs. By comparison, when their control inputs are in their inactive state, the buffers effectively disconnect themselves from the data bus and start to drive high-impedance (Z) values. (The use of tri-state buffers is what allows different ports and devices to push data on to a common data bus.)

When the CPU wishes to read the values being presented to this port's inputs, it must somehow cause a logic 0 to be applied to the port's `~i_enable` signal. In the case of our example system, there are thirty-two such input ports located at addresses $F000 to $F01F. In order for the CPU to individually address these ports, it must be able to generate a unique `~i_enable` signal for each port.

There are numerous ways of generating such signals. One technique would be to use the most significant output from the 4:16 decoder, `~cs[15]`, to indicate that we wish to address a location greater than or equal to $F000. We can then combine this signal with some additional logic that decodes the least-significant address bus bits to give us thirty-two signals corresponding to addresses $F000 to $F01F. Finally, we could combine each of these signals with the CPU's `~read` signal to give us thirty-two unique `~i_enable` control signals, one for each input port.

Output ports

In the case of an output port, the signals driving the port, `data[7:0]`, are connected to the system's data bus, while the port's outputs, `out_data[7:0]`, would be used to drive something in the outside world; for example, lights connected in such a way that a logic 0 would turn them OFF and a logic 1 would turn them ON (or vice versa) (Figure 1-12).

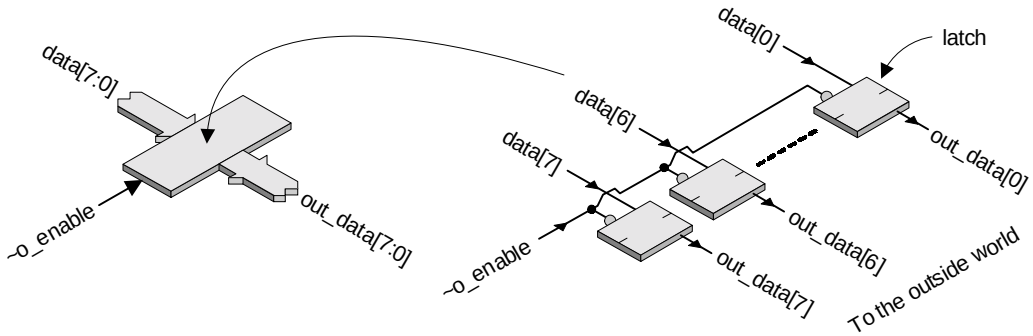


Figure 1-12. A simple 8-bit output port

Inside the port are logic functions known as *latches*, [1] one for each of the data signals. These latches act in such a way that, when their control inputs are in their active state (which is a logic

[1] Actually there are a variety of different types of output ports — we're concentrating on those based on latches because they're relatively common and easy to understand.

0 in this case), they pass whatever is on their data inputs through to their data outputs. By comparison, when their control inputs are in their inactive state, the latches remember the last values they were driving and continue to drive them to the outside world.

When the CPU wishes to send a value to a port's outputs, it must somehow cause a logic 0 to be applied to that port's `~o_enable` signal. In the case of our example system, there are thirty-two such input ports located at addresses $F020 to $F03F. In order for the CPU to individually address these ports, it must be able to generate a unique `~o_enable` signal for each port.

Once again, there are numerous ways of generating such signals. As before, we can use the most significant output from the 4:16 decoder, `~cs[15]`, to indicate that we wish to address a location greater than or equal to $F000. We can then combine this signal with some additional logic that decodes the least-significant address bus bits to give us thirty-two signals corresponding to addresses $F020 to $F03F. Finally, we could combine each of these signals with the CPU's `~write` signal to give us thirty-two unique `~o_enable` control signals, one for each output port.

Chapter 2

The *DIY Calculator's* CPU

Rampaging around the CPU

Everything we've discussed thus-far has merely been limbering-up exercises to prepare our mental muscles for the ordeals to come. Our real journey starts here and now as we quiver at the brink, poised to hurl ourselves into the bowels of the *DIY Calculator's* central processing unit (CPU) (Figure 2-1).

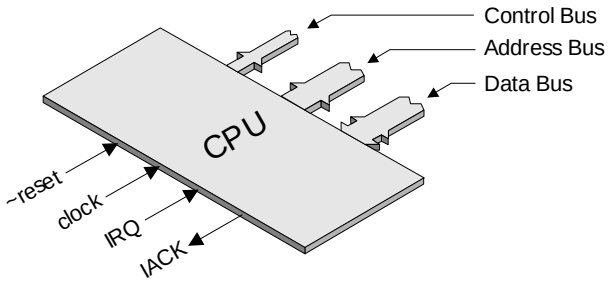


Figure 2-1. External view of the CPU

Our path will carry us on a rollicking roller-coaster ride (thrill-seekers only need apply); this will commence at a slow crawl to lull us into a false sense of security, followed by a screaming plunge into the nether regions where we'll disappear for a while ..... until we rocket out of the other side, clutching our stomachs and gasping for more.

Note: The discussions in this chapter assume familiarity with the *DIY Calculator's* addressing modes and instruction set, both of which are described in excruciating detail in *Appendix A*.

The accumulator (ACC) and status register (SR)

Before we commence our plunge into the unknown, it's worth noting that the CPU contains two very important registers called the *accumulator (ACC)* and the *status register (SR)* (Figure 2-2).

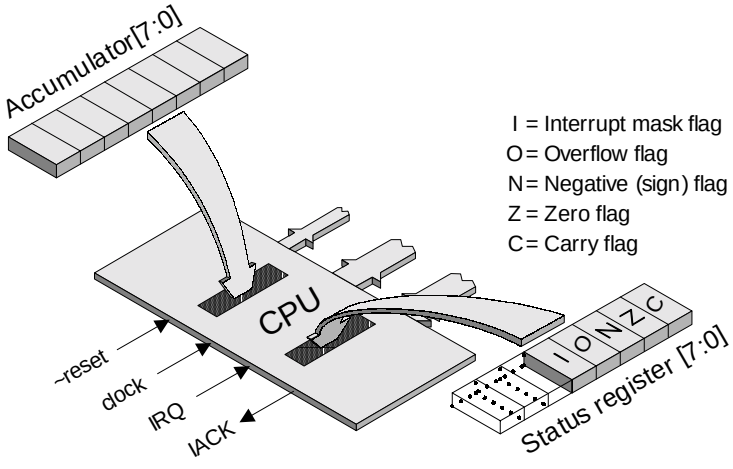


Figure 2-2. The accumulator (ACC) and status register (SR)

As its name implies, the accumulator – which is 8 bits wide in our system – is where the CPU gathers, or *accumulates*, intermediate results. The ways in which the contents of the

accumulator can be modified and used will become apparent as we progress. Suffice it to say that the CPU can be instructed to load a byte of data from any location in the *DIY Calculator's* memory into the accumulator (this involves taking a *copy* of the data in the memory; the contents of the memory at that location remain undisturbed). The CPU can also be instructed to perform a variety of arithmetic and logical operations on the data in the accumulator. Last but not least, the CPU can be instructed to store (copy) the contents of the accumulator into a memory location (this overwrites any existing contents in that memory location, but leaves the contents of the accumulator undisturbed).

In the case of the status register, each bit forming the register is called a *status bit*. These bits are also commonly referred to as *status flags* or *condition codes* because they serve to signal that certain conditions have occurred. Due to the fact that we may require to load the status register from the memory (or store its contents to the memory), it is usual to regard this register as being the same width as the data bus (8 bits in the case of the *DIY Calculator*). However, our CPU only actually employs five status flags, which occupy the five least-significant bits of the status register. This means that the three most-significant bits of the register exist only in our imaginations, so their non-existent contents are, by definition, undefined.

The negative (N) flag: This flag, which is also called the *sign flag*, typically contains a copy of the most significant bit *in* the accumulator following an operation *on* the accumulator. This flag is called "the negative flag" because – assuming the value in the accumulator is being regarded as a signed, twos complement number – a logic 1 in the most-significant bit of the accumulator indicates a negative value.

The zero (Z) flag: This flag is predominantly used to indicate whether or not the accumulator contains a value of zero following an operation on the accumulator. However, this flag is also used to indicate the presence or absence of a zero condition in the index register following the `INCX` ("*increment index register*") or `DECX` ("*decrement index register*") instructions. The zero flag is also used to indicate whether or not two numbers are equal following a `CMPA` (*"compare accumulator"*) instruction.

The carry (C) flag: As its name implies, the carry flag is predominantly used to store any carry-out (or borrow-out) from arithmetic operations performed on unsigned binary numbers. However, the carry flag is also used to store "the bit that falls off the end" during a shift or rotate instruction. This flag is also used to indicate if the value in the accumulator is greater than another value following a `CMPA` (*"compare accumulator"*) instruction.

The overflow (O) flag: This flag is used to indicate when the result of an arithmetic operation on a pair of signed (twos complement) binary numbers produces a result that cannot be correctly represented because it is two large to fit in the 8-bit accumulator.

The interrupt mask (I) flag: When the *DIY Calculator* is powered up, this flag is initialized to contain a logic 0. It can subsequently be loaded with a logic 1 by means of a `SETIM` (*"set interrupt mask"*) instruction; similarly, it can be re-loaded with a logic 0 by means of a `CLRIM` (*"clear interrupt mask"*) instruction. When the interrupt mask flag contains a logic 1, the CPU will respond to an external interrupt request (the concept of interrupts and the operation of the interrupt mask flag are discussed in more detail in *Chapter 3*).

Last but not least, we need to be aware of a certain convention as follows:

- A status flag is said to be "set" if it contains a logic 1, which is used to indicate a TRUE condition. For example, if the zero flag is set (contains a logic 1), this indicates that: *"It's TRUE to say that the current value stored in the accumulator is zero"* (that is, all of the bits in the accumulator contain logic 0s).
- A status flag is said to be "cleared" if it contains a logic 0, which is used to indicate a FALSE condition. For example, if the zero flag is cleared (contains a *l*ogic 0), this indicates that: *"It's FALSE to say that the current value stored in the accumulator is zero"* (that is, one or more of the bits in the accumulator contain a logic 1).

Unfortunately, this convention can appear somewhat counterintuitive – especially in the case of the zero flag – and it takes a little effort to wrap your brain around it the first time you see it, but you'll find that it does make sense once you get into the swing of things.

The CPU exposed

The CPU contains three main blocks of logic: the *arithmetic logic unit (ALU)*, the control logic that decodes and executes instructions, and the addressing logic that is use to point to the various memory locations. The CPU also contains four 8-bit registers: the *accumulator (ACC)*, a *temporary register (TMP)*, the *instruction register (IR)* and the *status register (SR)* (Figure 2-3).

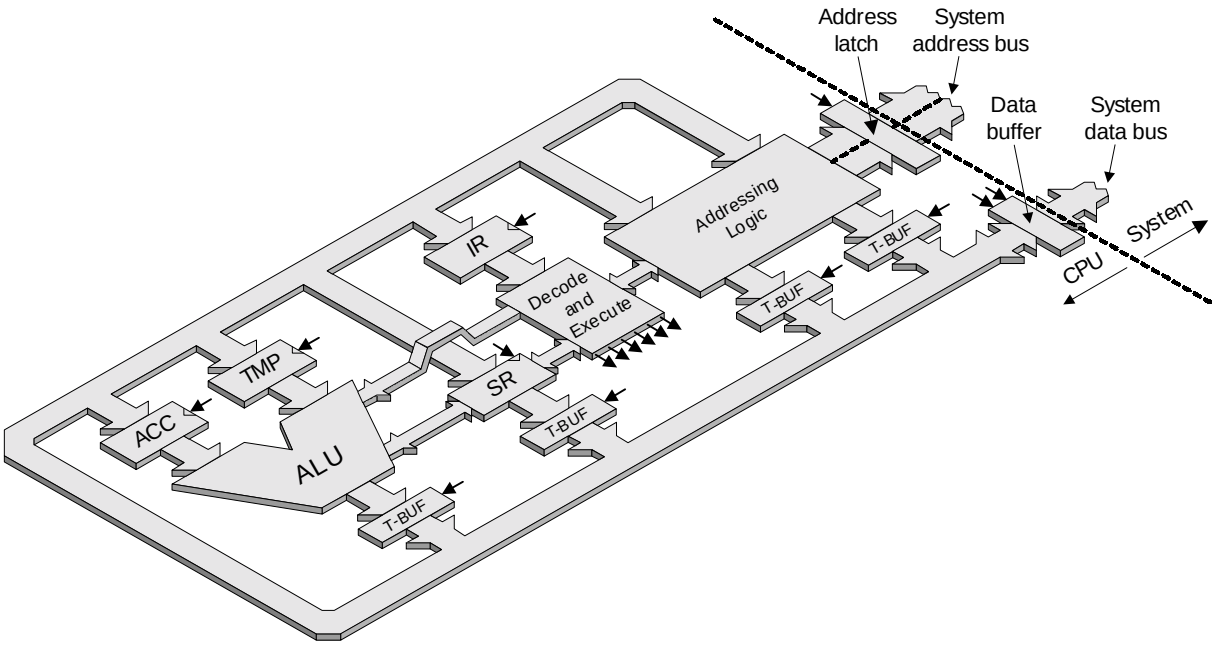


Figure 2-3. Internal view of the CPU

As we previously noted, the status register actually contains only 5 bits, but we typically treat it as being an 8-bit entity. In fact, there's some additional logic associated with the status register that isn't shown in this high-level view.

Each of the blocks marked *T-BUF* represents a group of eight tri-state buffers (one for each data bus bit). All of the control signals (the black arrows) feeding the various registers and tri-state buffers are generated by the instruction decoder and executer (control) logic.

The arithmetic-logic unit (ALU)

The heart (or perhaps the guts) of the CPU is the ALU, because this is where of the nitty-gritty number crunching and data manipulation takes place. As the *DIY Calculator's* data bus is 8 bits wide, our ALU works with 8-bit chunks of data (Figure 2-4).

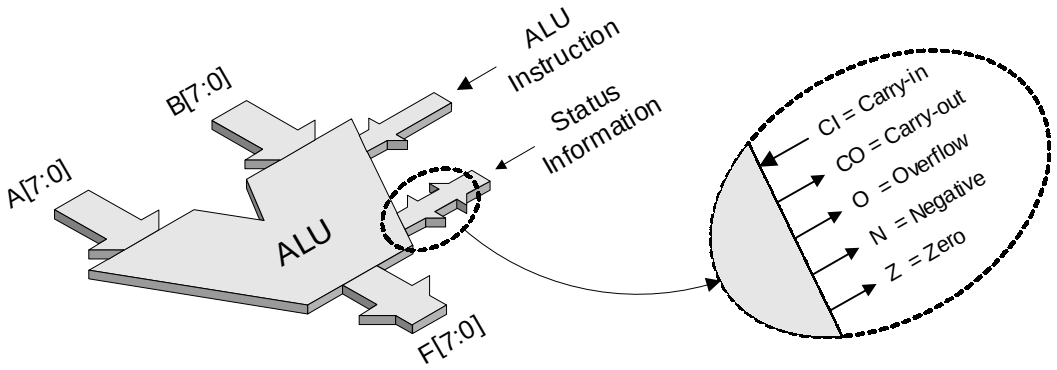


Figure 2-4. The ALU is where the "number crunching" takes place

The ALU accepts two 8-bit chunks of data as input (we'll refer to these two inputs as `A[7:0]` and `B[7:0]`); it "scrunches" them together using some arithmetic or logical operation; and it and outputs an 8-bit result which we'll call `F[7:0]`. Whatever operation is performed on the data is dictated by a pattern of logic 0s and logic 1s called the ALU instruction. For example, one pattern may instruct the ALU to add `A[7:0]` and `B[7:0]` together, while another may request the ALU logically AND each bit of `A[7:0]` with the corresponding bit in `B[7:0]`.

Note that the ALU is completely *asynchronous*, which means it is not directly controlled by – or synchronized to – the main system clock. As soon as any changes are presented to the ALU's data, instruction, or carry-in inputs, these changes will immediately start to ripple through its logic gates and will eventually appear at the `F[7:0]` data outputs and the status outputs.

The "core" ALU

We can start by visualizing an inner "core" ALU that performs only five simple functions as illustrated in Table 2-1.

Function	Outputs F[7:0]	Flags Modified
Logical AND	A[7:0] & B[7:0]	N, Z
Logical OR	A[7:0] | B[7:0]	N, Z
Logical XOR	A[7:0] ^ B[7:0]	N, Z
Addition (ADD)	A[7:0] + B[7:0] + CI	CO, O, N, Z
Compare (CMP)	A[7:0] ≡ B[7:0]	CO, Z

Table 2-1. The ALU's five core functions

The three logical functions, AND, OR, and XOR are said to operate in a *bitwise* fashion, which means that they operate on each bit independently. For example, in the case of the AND, `A[0]` is AND-ed with `B[0]` to generate `F[0]`, `A[1]` is AND-ed with `B[1]` to generate `F[1]`, and so forth. The ADD function adds `A[7:0]`, `B[7:0]`, and the `CI` ("*carry-in*") signal together, while the CMP ("*compare*") function compares `A[7:0]` to `B[7:0]` to see which is the larger (this function assumes that both `A[7:0]` and `B[7:0]` represent unsigned integers).

One way in which we could implement this core ALU would be to create individual blocks for each of the functions and then "glue" these blocks together using a multiplexer (Figure 2-5).

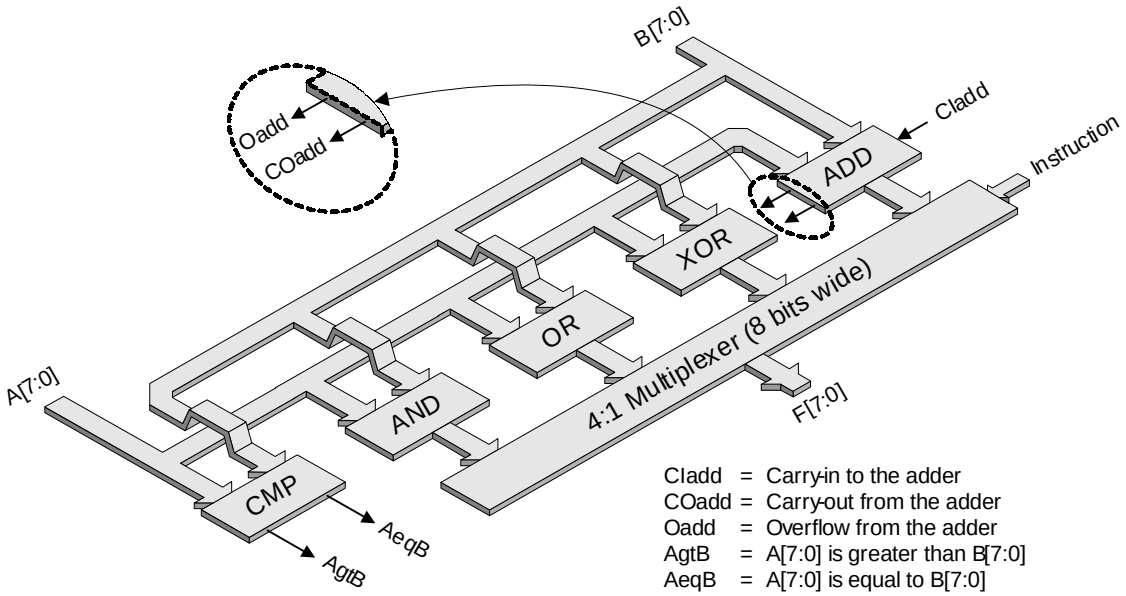


Figure 2-5. One way to implement the five core ALU functions (excluding any status logic)

In this scenario, we are using two instruction bits (which can represent four patterns of 0s and 1s) to control a 4:1 (four-to-one) multiplexer so as to select one of four input data channels, each of which is 8 bits wide. The `A[7:0]` and `B[7:0]` signals are presented to all of the function blocks, but the multiplexer selects the outputs only from the function in which we are interested. The reason we require only a 4:1 multiplexer (as opposed to a 5:1 version) is that the fifth function – the CMP – outputs only status information; that is, it doesn't generate any data that we need to feed through to the `F[7:0]` data outputs.

Note: The advantage of the multiplexer-based approach described here is that it's easy to understand, but we would be unlikely to use this technique in a real-world implementation. This is because we're interested in being able to perform only a single function at any particular time; thus, we would examine the functions to find areas of commonality allowing us to share gates between them. To put this another way, instead of having multiple distinct functions feeding a multiplexer, we'd probably lose the multiplexer and "scrunch" all of the functions together into one "super function", thereby allowing us to reduce the ALU's total gate count and increase its speed.

So now we know how to implement the data-processing portion of our core ALU, but we've yet to decide how we're going to use the AgtB (*"A greater than B"*), AeqB (*"A equal to B"*), Oadd (*"overflow from the adder"*), and COadd (*"carry-out from the adder"*) signals to generate the CO (*"carry-out"*), O (*"overflow"*), N (*"negative"*), and Z (*"zero"*) status signals (Figure 2-6).

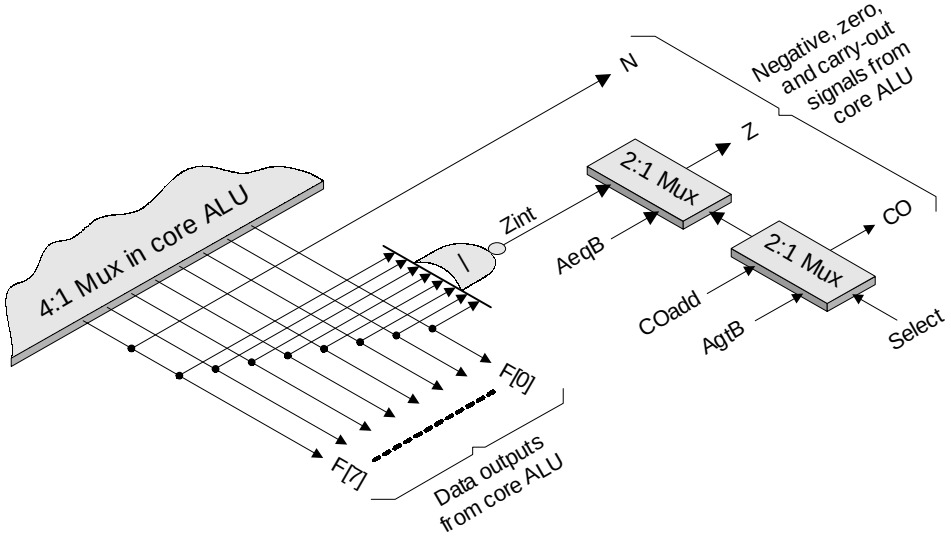


Figure 2-6. Generating status signals from the five core functions

The N ("*negative*") signal is the easiest of all, because it's simply a copy of the most-significant bit of the data outputs (that is, F[7]). Things get a little more complicated when we come to the Z ("*zero*") signal, because this depends on the type of operation the ALU is performing. In the case of the AND, OR, XOR, and ADD functions, the zero signal is set to logic 1 if the result from the operation is all 0s. We can create an internal signal called Zint to implement this by simply feeding all of the F[7:0] data outputs into an 8-bit NOR gate (the output from a NOR gate is logic 1 if all of the inputs are logic 0; conversely, the output is logic 0 if any of the inputs are logic 1). However, in the case of the CMP function, we wish the Z signal to be set to logic 1 if the two data values A[7:0] and B[7:0] are equal (this is represented by the AeqB signal coming out of the CMP block).

The bottom line is that we've got a single signal, Z, which we wish to reflect the state of one of two signals (Zint and AeqB) depending on the function being performed. We can achieve this by feeding Zint and AeqB into a 2:1 multiplexer, whose select input is controlled by a third instruction bit driving the core ALU. Similarly, we usually want the CO (*"carry-out"*) status signal to reflect the carry-out from the ADD function on its COadd signal; however, if we're performing a CMP instruction, then we want the CO signal to be set to logic 1 if the unsigned binary value on the A[7:0] inputs is greater than that on the B[7:0] inputs. Once again, we can achieve this by feeding both the COadd and AgtB signals into a 2:1 multiplexer controlled by our third instruction bit.

Note: Primitive logic gates like NOT, AND, OR, XOR, NAND, NOR, and XNOR – along with more complex logic functions like multiplexers, latches, and registers – are introduced in the book *Bebop to the Boolean Boogie (An Unconventional Guide to Electronics), Second Edition*, ISBN: 0750675438, by Clive "Max" Maxfield (the only electronics book in the world to feature a Seafood Gumbo recipe!),

Last but not least, the Oadd ("*overflow from the adder*") signal is used to directly drive the ALU's O ("*overflow*") status signal. (The overflow for an 8-bit adder can be generated by XOR-ing the carry-in and carry-out associated with the most-significant bit of the result, which would be bit [7] in this case.)

Extending the core ALU to perform subtractions and suchlike

Thus far, we've designed a core ALU that can perform five simple functions, but our CPU will require more of us in order to implement the instruction set described in *Appendix A*. For example, our core ALU has an ADD function that can add two 8-bit numbers together (along with a carry-in signal), but we wish to be able to perform both addition and subtraction in the form of the ADD ("*add without carry*") and SUB ("*subtract without carry*") instructions, respectively. Furthermore, in order to perform multi-byte additions and subtractions, we require the use of the ADDC ("*add with carry*") and SUBC ("*subtract with carry*") instructions.

The point is that the ADD, ADDC, SUB, and SUBC instructions are all going to employ the ADD function in the core ALU. And while we're pondering this poser, we might also decide to consider the INCA and DECA instructions, which increment or decrement the contents of the accumulator, respectively.

Note: At this point, it's probably worth taking a few moments to remind ourselves as to how a computer performs subtractions in the first place. Let's assume we have two 8-bit binary numbers called AA[7:0] and BB[7:0] and that we wish to subtract the latter value from the former. We can represent this operation as follows (where F[7:0] represents the 8-bit result):

```
F[7:0] = AA[7:0] – BB[7:0]
```

From the discussions in *Chapter 4* of our book *How Computers Do Math*, we know that we can also represent this operation as shown below:

```
F[7:0] = AA[7:0] + (–BB[7:0])
```

In turn, we can obtain –BB[7:0] by taking the twos complement of BB[7:0], and we know that the twos complement of a binary number is equal to its ones complement plus 1. Last but not least, we know that the ones complement of a binary number can be generated by logically negating it, which means converting all of its 0s to 1s, and vice versa. The end result is that we can re-write our original equation as:

```
F[7:0] = AA[7:0] + !BB[7:0] + 1
```

Where !BB[7:0] represents the ones complement of BB[7:0].

In the fullness of time, we're going to connect our 8-bit accumulator directly to the A[7:0] inputs feeding the core ALU. By comparison, the core ALU's B[7:0] inputs are going to be fed from a special block of logic that we'll call the *complementor* (Figure 2-7).

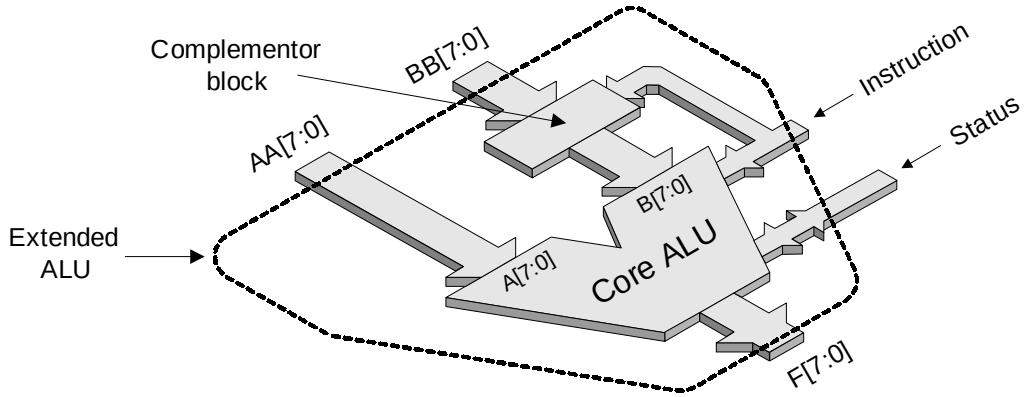


Figure 2-7. Extending the core ALU to perform subtractions and suchlike

In the case of instructions such as AND, OR, XOR, ADD, and ADDC, we want our new complementor block to pass whatever value is on the BB[7:0] inputs directly through to its outputs without any modification. In the case of the SUB and SUBC instructions, however, our new block must first logically negate the value on the BB[7:0] inputs (that is, swap all of the 0s for 1s, and vice versa) before passing it on to the core ALU. Finally, in the case of instructions such as INCA and DECA, we want our complementor block to generate the appropriate value to be added to, or subtracted from, the accumulator (which, as we previously stated, is going to be connected to the AA[7:0] inputs). One way in which we might implement this complementor block is shown in Figure 2-8.

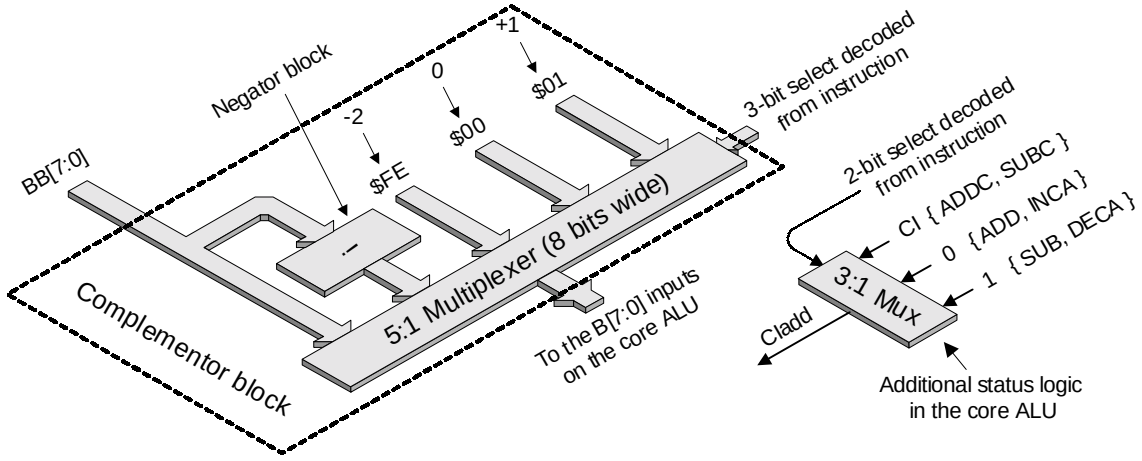
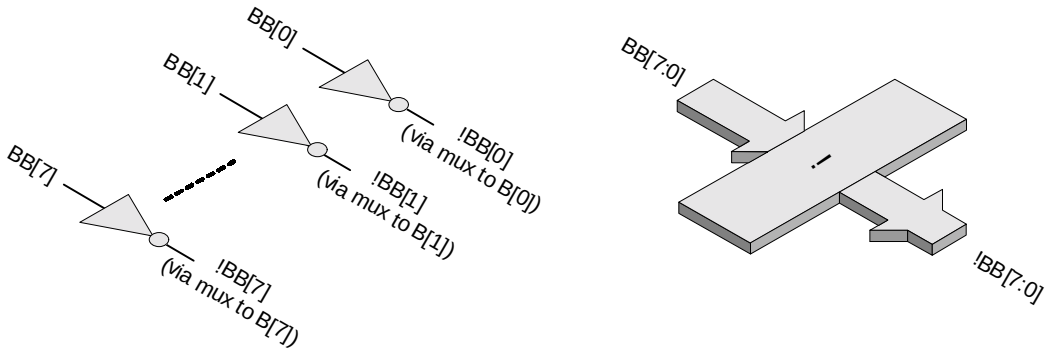


Figure 2-8. The contents of the complementor block

The way this works is that, if the pattern on the instruction bits represents an operation such as AND, OR, XOR, ADD, or ADDC, then we'll decode them in such a way that they cause the 5:1 multiplexer in the complementor block to select the value on BB[7:0]. By comparison, a SUB or SUBC instruction will cause the multiplexer to select the outputs from the negator block, whose value is the logical inverse of that found on BB[7:0]. The negator block will be formed from eight NOT gates, one for each data bit as shown in Figure 2-9.



(a) Gate-level view (b) Abstract view

Figure 2-9. The contents of a NOT-based negator block

Observe that an exclamation mark "!" is used to indicate a NOT function. Thus, `!BB[0]` represents the logical negation of `BB[0]`, `!BB[1]` represents the logical negation of `BB[1]`, and so forth. Furthermore, as the ones complement of a binary number is generated by inverting all of its bits (swapping the 0s for 1s, and vice versa), `!BB[7:0]` is the ones complement of `BB[7:0]`.

In the case of a `DECA` instruction, the 5:1 multiplexer in Figure 2-8 will select a hard-wired value of $FE, which – assuming we're working with signed binary numbers – equates to -2_{10} in decimal (we'll explain the reasoning behind this value in a moment). By comparison, an `INCA` instruction will cause the multiplexer to select a hardwired value of $01 (which is of course $+1_{10}$ in decimal). The final input to the multiplexer is a hard-wired value of $00 (zero), whose purpose will be revealed in the fullness of time.

Now observe the 3:1 multiplexer that we've added to the core ALU as illustrated in Figure 2-8. This allows us to force the `CIadd` ("*carry-in to the adder*") signal to a logic 0 for the `ADD` and `INCA` instructions, and to force it to a logic 1 for the `SUB` and `DECA` instructions.

The way in which the `ADD`, `ADDC`, `INCA`, `SUB`, `SUBC`, and `DECA` instructions employ the ADD function in the core ALU may be summarized as shown in Table 2-2.

Instruction	ALU B[7:0]	CIadd	Operation performed
ADD	BB[7:0]	0	F[7:0] = AA[7:0] + BB[7:0] + 0
ADDC	BB[7:0]	CI	F[7:0] = AA[7:0] + BB[7:0] + CI
INCA	$01	0	F[7:0] = AA[7:0] + $01 + 0
SUB	!BB[7:0]	1	F[7:0] = AA[7:0] + !BB[7:0] + 1
SUBC	!BB[7:0]	CI	F[7:0] = AA[7:0] + !BB[7:0] + CI
DECA	$FE	1	F[7:0] = AA[7:0] + $FE + 1

Table 2-2. Summary of operations for the initial implementation

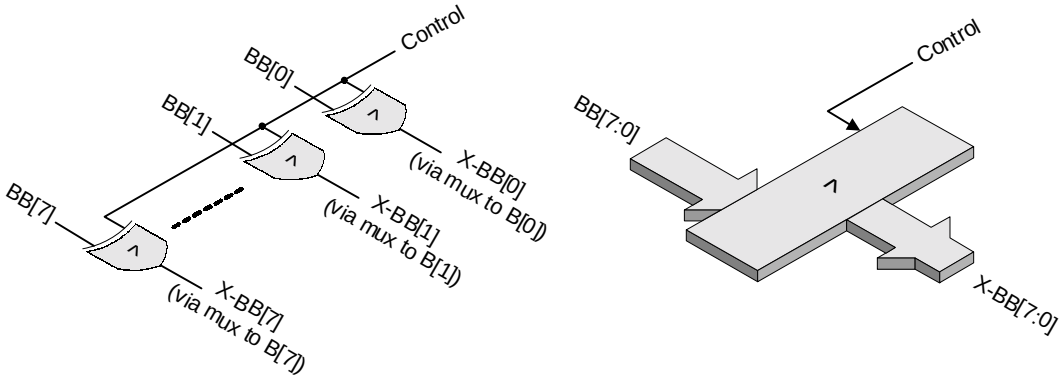
It now becomes apparent why we force our multiplexer to select the hard-wired $FE value ($-2_{10}$ in decimal) for the `DECA` instruction; in this case, the logic 1 on the `CIadd` signal means that the

operation actually performed by the extended ALU is A[7:0] + (–2) + 1. This is equivalent to A[7:0] + (–1), which is equivalent to A[7:0] –1, which is – of course – what we wanted our DECA instruction to do in the first place (phew!).

An alternative complementor block implementation

It's important to understand that Figures 2-8 and 2-9 reflect just one of many implementation alternatives. In fact, if the truth be told, our first pass implementation was not a very good one; we just presented this version to get our creative juices flowing and to provide a basis for comparison.

For example, our original complementor block comprised two functions: a 5:1 multiplexer and a negator, where the negator simply comprises eight NOT gates (one for each signal in the data path). We chose to use this form of negator because it made it easy to visualize what we were trying to do, but we probably wouldn't use this technique to create a physical realization. Instead, a slightly better implementation would be to replace the negator's eight NOT gates with eight 2-input XOR gates, each of which could have one of its inputs connected to a common control signal (Figure 2-10).



(a) Gate-level view (b) Abstract view

Figure 2-10. The contents of an XOR-based negator block

In this case, a logic 0 on the control signal would pass the values on BB[7:0] through the XOR gates unmodified (that is, signals X-BB[7:0] = BB[7:0]), while a logic 1 would cause these values to be inverted (that is, X-BB[7:0] = !BB[7:0]). This would allow us to replace our 5:1 multiplexer with a 4:1 version that would require substantially fewer gates.

Furthermore, we could replace the three hard-wired data inputs ($FE, $01, and $00) in our original implementation with just two hard-wired values ($FF and $00). This would allow us to replace our new 4:1 multiplexer with a 3:1 version requiring even fewer gates (Figure 2-11).

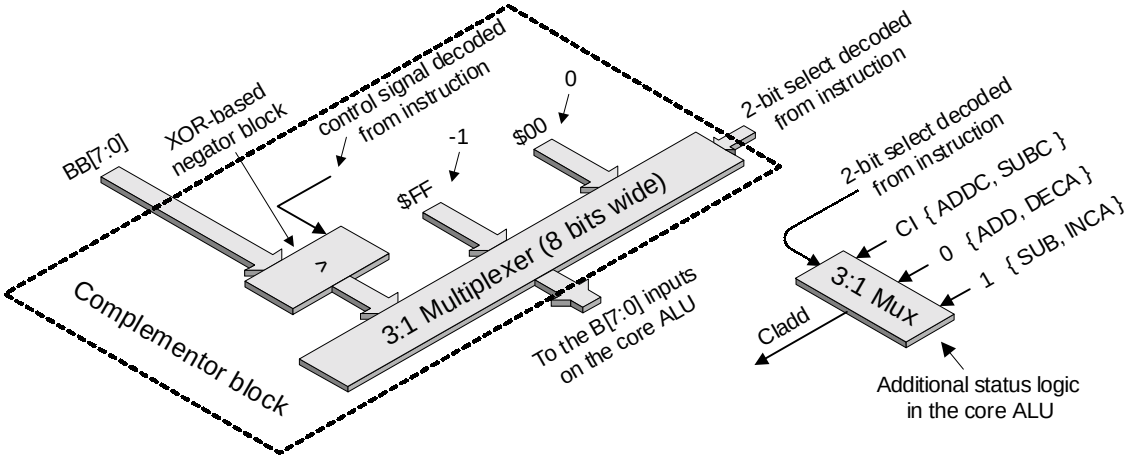


Figure 2-11. The contents of the alternative complementor block implementation

In this case, we'd also modify the 3:1 status logic multiplexer such that the `ADDC` and `SUBC` instructions cause the `CIadd` (*"carry-in to the adder"*) signal to be driven by the current contents of the carry flag (`CI`); the `ADD` and `DECA` instructions force `CIadd` to be driven by a logic 0; and the `SUB` and `INCA` instructions force `CIadd` to be driven by a logic 1. The internal machinations of this alternative implementation may be summarized as shown in Table 2-3:

Instruction	ALU B[7:0]	Cladd	Operation performed
ADD	BB[7:0]	0	F[7:0] = AA[7:0] + BB[7:0] + 0
ADDC	BB[7:0]	CI	F[7:0] = AA[7:0] + BB[7:0] + CI
INCA	$00	1	F[7:0] = AA[7:0] + $00 + 1
SUB	!BB[7:0]	1	F[7:0] = AA[7:0] + !BB[7:0] + 1
SUBC	!BB[7:0]	CI	F[7:0] = AA[7:0] + !BB[7:0] + CI
DECA	$FF	0	F[7:0] = AA[7:0] + $FF + 0

Table 2-3. Summary of operations for the alternative complementor implementation

Extending the core ALU to perform shifts and rotates

Cheer up, because this part is going to be a doddle. We want to be able to shift and rotate the contents of the accumulator using the following instructions: `SHL` ("*shift left*"), `SHR` ("*shift right*"), `ROLC` ("*rotate left through the carry bit*"), and `RORC` ("*rotate right through the carry bit*"). One way in which we could achieve this is to further extend our core ALU by "gluing" a new shifter/rotator block onto it as shown in Figure 2-12.

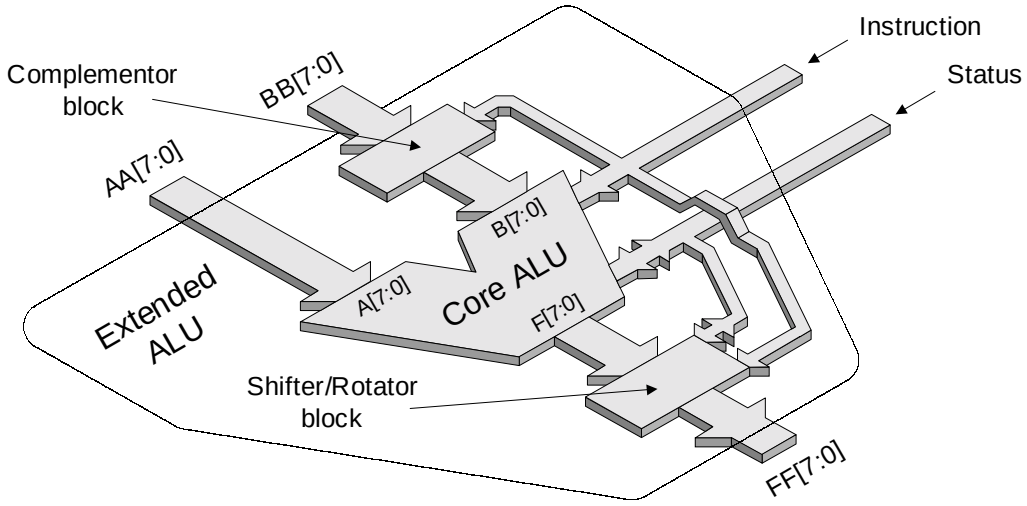


Figure 2-12. Extending the core ALU to perform shifts and rotates

Note that, although we've decided to attach this new block to the outputs of the core ALU, we could stick in a number of other places if we so desired (for example, it could go between the accumulator and the core ALU). However, the fact that we have attached this block to the ALU's outputs means that the logic we use to generate the Z ("*zero*") and N ("*negative*") status flags must be removed from the core ALU's `F[7:0]` outputs and reattached to the extended ALU's `FF[7:0]` outputs (the logic for the Z and N status flags was illustrated in Figure 2-6). Before leaping headfirst into this new block, there's one point we should discuss. As you may recall, the multiplexer in our complementor block had one set of inputs connected to a hard-wired value of $00 (see Figure 2-8), but we never got around to explaining why. Well, the accumulator is going to be connected to the extended ALU's `AA[7:0]` inputs. If we decide to perform a shift or rotate operation, then we want the values on the `AA[7:0]` inputs to be passed through the core ALU and fed directly into the shifter/rotator block without modification. But we know that the core ALU is always going to try to perform some sort of function on this data; after all, that's what it's there for! As usual, there are a number of different ruses we could employ to address this problem, but the technique we decided to use is as follows:

- Assume that the extended ALU is presented with a `SHL`, `SHR`, `ROLC`, or `RORC` instruction.
- This instruction is decoded in such a way that the multiplexer in the complementor block selects the hard-wired $00 value; the core ALU is instructed to perform an ADD operation; and the `CIadd` (*"carry-in to the adder"*) signal is presented with a logic 0.
- Thus, the core ALU simply adds $00 to the value on the `AA[7:0]` inputs, **which has no effect whatsoever!** The unmodified `AA[7:0]` value is then handed on to the shifter/rotator block for it to perform the real operation (pretty cunning, huh?).

Now let's dive into the shifter/rotator block itself. In the case of instructions such as `AND`, `OR`, `ADD`, and `SUB`, we want our new block to simply pass whatever comes out of the core ALU straight through without modification. It is only in the case of the shift or rotate instructions that this new block comes into play (Figure 2-13).

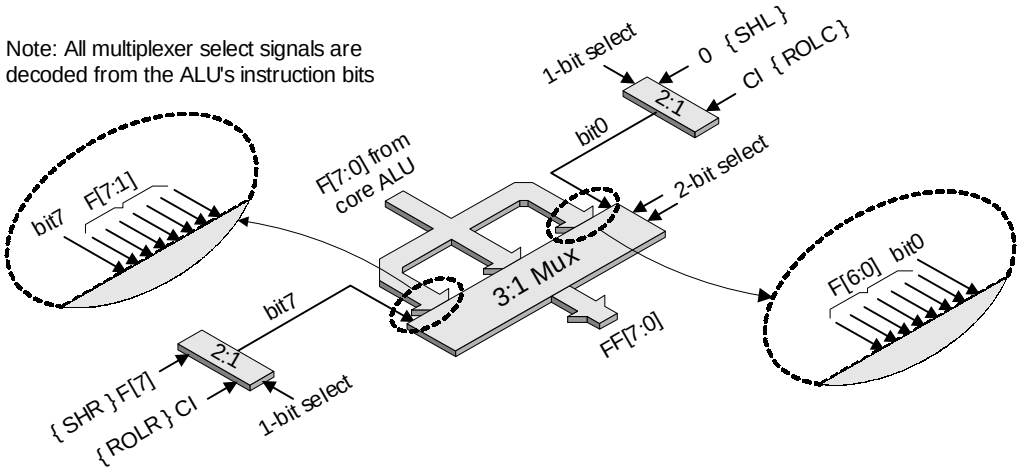


Figure 2-13. The contents of the shifter/rotator block

The mainstay of the shifter/rotator is a 3:1 multiplexer, in which each input channel is 8 bits wide. In the case of instructions like `AND`, `OR`, `ADD`, and `SUB`, we decode our instruction bits such that they cause this multiplexer to choose the `F[7:0]` outputs from the core ALU and pass them straight through to its `FF[7:0]` outputs. That is, the value on `F[7]` appears on `FF[7]`, the value on `F[6]` appears on `FF[6]`, and so on (this is represented to the multiplexer selecting the central set of inputs in Figure 2-13).

When we turn our attention to the `SHR` ("*shift right*") and `RORC` ("*rotate right through the carry bit*") instructions, a little thought reveals that we want both of them to shift whatever is coming out of the core ALU one bit to the right. That is, we want the value on `F[7]` to appear on `FF[6]`, the value on `F[6]` to appear on `FF[5]`, and so on (this is represented by the main multiplexer selecting the left-hand set of inputs in Figure 2-13).

In fact the only difference between these instructions is the value that comes out of `FF[7]`, which needs to be a copy of whatever was on `F[7]` for an `SHR`, or a copy of whatever is on the ALU's main `CI` ("*carry-in*") input for an `RORC`. In order to achieve all of this, we use a simple 2:1 multiplexer to generate a signal called `bit7`; an `SHR` causes this multiplexer to select the input connected to `F[7]`, while an `RORC` causes it to select the input being driven by the `CI` signal. Now consider the magnified view of the left-hand inputs to the main multiplexer shown in Figure 2-13. As we see, the most-significant input is connected to the `bit7` signal, while the remaining seven inputs are connected to `F[7:1]`. Thus, when the main multiplexer selects these inputs and passes them through to its outputs, the effect is to shift the bits coming out of the core ALU one bit to the right, and to insert whatever value is on the `bit7` signal into the most-significant bit.

Similarly, we want both the `SHL` ("*shift left*") and `ROLC` ("*rotate left through the carry bit*") instructions to shift whatever is coming out of the core ALU one bit to the left. That is, we want the value on `F[0]` to appear on `FF[1]`, the value on `F[1]` to appear on `FF[2]`, and so on (this is represented by the main multiplexer selecting the right-hand set of inputs in Figure 2-13). In this case, the only difference between these instructions is the value that comes out of `FF[0]`, which needs to be a logic 0 for an `SHL`, or a copy of whatever is on the ALU's main `CI` ("*carry in*") input for an `ROLC`.

Once again, we use a simple 2:1 multiplexer to generate a signal called `bit0`; an `SHL` causes this multiplexer to select the input connected to a logic 0, while an `ROLC` causes it to select the input being driven by the `CI` signal. Now consider the magnified view of the right-hand inputs to the main multiplexer. As we see, the most-significant inputs are connected to the `F[6:0]` signals, while the least-significant input is connected to `bit0`. Thus, when the main multiplexer selects these inputs and passes them through to its outputs, the effect is to shift the bits coming out of the core ALU one bit to the left, and to insert whatever value is on the `bit0` signal into the least-significant bit.

The only remaining task required to complete our shifter/rotator is to modify the logic used to drive the `CO` ("*carry-out*") signal generated by the ALU. In our earlier discussions, we used a 2:1 multiplexer to select between the `COadd` signal from the ADD function and the `AgtB` signal from the CMP function. To satisfy the requirements of our shifter/rotator, we now need to replace that 2:1 multiplexer with a 4:1 version (Figure 2-14).

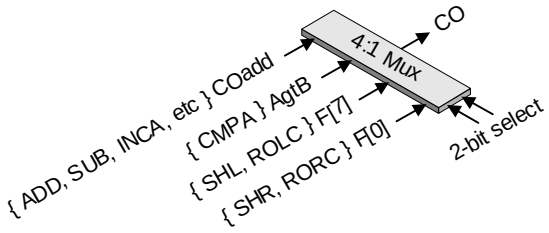


Figure 2-14. Modified carry-out logic to accommodate shift and rotate instructions

As usual, the select inputs controlling this multiplexer are decoded from the instruction bits driving the ALU. The `ADD`, `OR`, `XOR`, `ADD`, `ADDC`, `SUB`, `SUBC`, `INCA`, and `DECA` instructions all cause the multiplexer to select the `COadd` signal as before, while the `CMPA` instruction causes it to select the `AgtB` signal. In the case of the `SHL` or `ROLC` instructions, the multiplexer selects the input connected to the `F[7]` signal coming out of the core ALU, where `F[7]` is the bit that conceptually "drops off the end" when we shift everything to the left. Similarly, the `SHR` or `RORC` instructions cause the multiplexer to select the `[F0]` signal coming out of the core ALU, which is the bit that "drops off the end" when we shift everything to the right.

And that's all there is to it with regards to the ALU. The combination of the core ALU with the complementor and the shifter/rotator blocks provides us with everything we need to satisfy all of the *DIY Calculator's* ALU-related instructions.

Connecting the accumulator and data bus to the ALU

Once we have a functional ALU, we need to connect it to our trusty old accumulator and a few of its closest friends (Figure 2-15).

We commence by connecting the outputs from our 8-bit accumulator (ACC) to the extended ALU's `AA[7:0]` inputs. Similarly, we connect an 8-bit temporary register (TMP) to the ALU's `BB[7:0]` inputs. The inputs to both the accumulator and the temporary register are driven by an internal 8-bit data bus, which is, in turn, linked to the outside world by means of an 8-bit bi-directional data buffer.

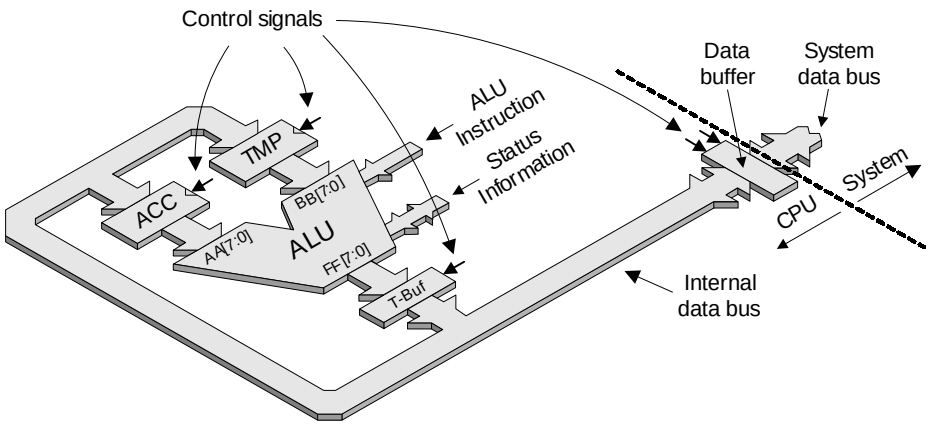


Figure 2-15. Connecting the accumulator and data bus to the extended ALU

Both the accumulator and the temporary register have control signals in the form of clocks. The fact that these signals are clocks is indicated by the chevrons ('v' shapes) on the symbols at the point where the signals enter them. Note that these clocks are not the same as the CPU's main clock input, although they are derived from it. Also note that these clocks can be activated (or not activated) individually as required, thereby allowing us to load one register or the other (or neither of them).

Now consider the bi-directional data buffer linking the CPU's internal data bus to the main system's data bus. This buffer also has control signals that dictate whether it will (a) allow data from the outside world to pass into the CPU, (b) allow data from the CPU to pass to the outside world, or (c) completely disconnect the internal data bus from the outside world.

Last but not least, we have an 8-bit tri-state buffer called *T-Buf*, which links the ALU's `FF[7:0]` outputs to the internal data bus. Depending on the state of the *T-Buf's* control signal, it either propagates the outputs from the ALU onto the data bus or it isolates the ALU from the bus. All of the control signals for the registers, latches, and buffers are generated by the instruction decoder and executer (control) logic that we haven't gotten around to worrying about yet.

One cunning point occurs when we wish to take the contents of the accumulator, use the ALU to perform some operation on these contents, and then feed the outputs from the ALU through its tri-state buffers onto the internal data bus and store the result back in the accumulator. But as soon as we load the accumulator with its new contents, these contents will be presented to the ALU's inputs, which will modify the results coming out of the ALU's outputs, and so forth.

The reason this isn't a problem is that the registers forming the accumulator and the gates forming the ALU all have delays. This means that when we load the result from the ALU into the accumulator, it takes a fraction of a second before the accumulator's outputs begin to respond. Similarly, when the outputs from the accumulator *do* respond, it takes some finite amount of time for the effect to ripple through the ALU and work its way back to the accumulator's inputs. Thus, by the time these unwanted signals present themselves to the accumulator, the result we were interested in is already safely stored away inside it.

Connecting the status register to the ALU and data bus

The next step is to connect the ALU's status signals to the *status register (SR)*, and also to link the status register to the CPU's internal data bus (Figure 2-16).

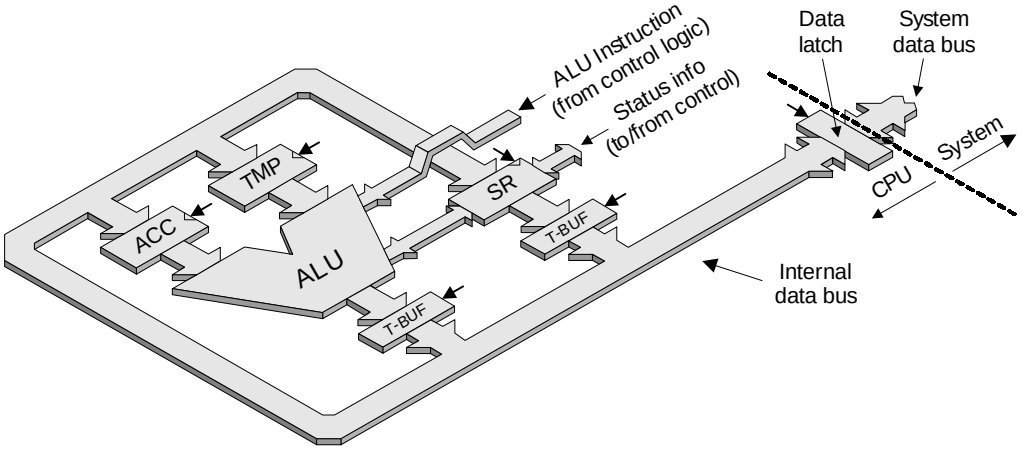


Figure 2-16. Connecting the status register

In this simplified diagram, the status register is only shown as having a single clock. In reality, each of the five register bits forming our status register (I, O, N, Z, and C) would have individual clocks; alternatively they might share a common clock, but each would be equipped with an individual clock-enable. Once again, all of these control signals will be generated by the instruction decoder and executer (control) logic that we haven't looked at yet.

This leads us to a related consideration, in that the ALU is *always* outputting values on its `O`, `N`, `Z`, and `CO` status signals, but this doesn't mean that we're obliged to save all of them in the status register, because one or more of these values may not be relevant in the context of the instruction that's being executed. (Note that the interrupt mask (I) status bit is handled by a separate mechanism as discussed in *Chapter 3*.) Depending on the particular operation that we're trying to perform, the control logic in the instruction decoder and executer will only cause the appropriate status register bits to be loaded.

When we power-up or reset the CPU, all of the status bits are cleared to *logic 0s*, and any subsequent values in the status bits depend on the results generated by whatever instructions have most recently been performed. Also, we occasionally require the ability to directly read values from, and write values to, the status register, which explains why we connected this register to the internal data bus. To better comprehend some of status register's more subtle details requires us to plunge a little deeper into the logic surrounding this little rapscallion (Figure 2-17).

As we see, the inputs to our status register would actually be driven by a set of five 2:1 multiplexers, which are used to choose between the signals on the internal data bus and the outputs from the ALU. These multiplexers all share a common select control signal generated by our furtive control logic. Similarly, in Figure 2-13, we appeared to have two sets of outputs from the status register, where the first set was connected to our control logic and the second was used to drive the tri-state buffer. However, the status register really has only a single set of

outputs that are used to drive both the control logic *and* the tri-state buffers (the tri-state buffers also share a common enable signal generated by our control logic).

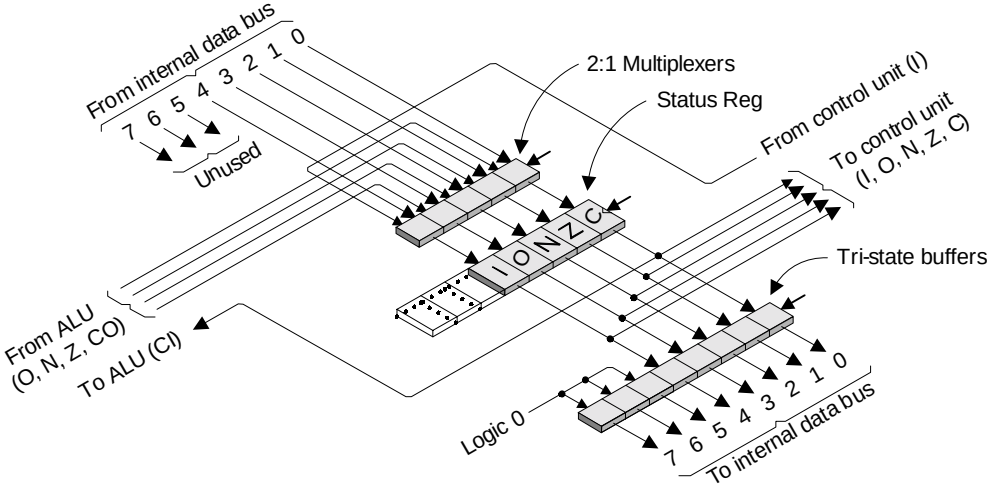


Figure 2-17. A closer look at the status register logic

Although some of the wires are shown as crossing over each other in this illustration, the only points at which they are electrically connected are those indicated by small black dots. Also remember that although we've only shown a single clock driving the status register, each bit forming the register would have an individual clock (or a common clock and an individual clock-enable).

As we previously discussed, our internal data bus is 8 bits wide, but our status register contains only 5 bits. This isn't a problem in the case of writing a value to status register, because all we have to do is connect bits 0 to 4 of the internal data bus to the multiplexer inputs and forget about bits 5 through 7. However, things are a little trickier when we wish to read a value from the status register, because we only have 5 register bits available to drive our 8-bit bus. The solution to this problem is quite simple, because although the status register itself contains only 5 bits, we can make the tri-state buffer 8 bits wide, and connect its three most-significant inputs to a logic value of our choice (we'll assume that they're connected to logic 0).

Figure 2-17 also allows us to understand how the C (carry) flag can be driven by the ALU's carry-out (`CO`) output signal and, at the same time, can be used to drive the ALU's carry-in (`CI`) input signal. For example, in the case of one of our rotate instructions (`ROLC` and `RORC`), the bit that's shifted into the accumulator comes *from* the carry flag, while the bit that "*drops off the end*" from the accumulator is copied (stored) *into* the carry flag. This is possible due to the delays in the circuit, which mean that the original contents of the carry flag have already been utilized by the time the new contents overwrite them.

Finally, note once again that the interrupt mask flag is not connected to the ALU, but is instead driven directly by the control logic. (The concept of interrupts is discussed in detail in *Chapter 3*).

Adding the instruction register (IR) and control logic

Now we come to consider the control logic that's been lurking furtively in the background for so long. In fact, there are two key pieces to the control logic: the *instruction register (IR)* and the *instruction decoder and executor* (we'll refer to the latter as the *control logic* for short) (Figure 2-18).

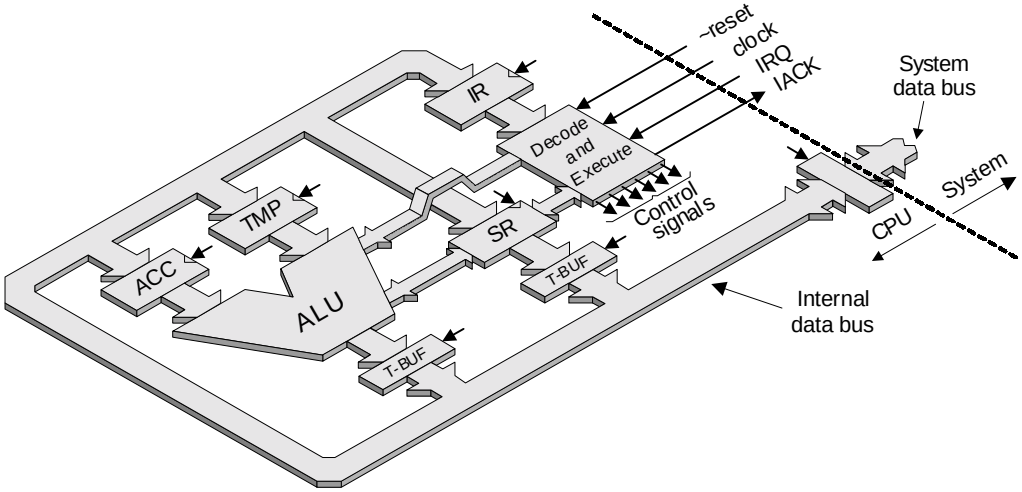


Figure 2-18. The instruction register (IR) and control logic

The control logic is the real "brain" of the CPU, because it supplies all of the timing and control signals to the other units. This block has three main inputs from the outside world: the system clock (`clock`), the system reset (`~reset`), and the interrupt request (`IRQ`). When the `~reset` signal is activated (either explicitly or when power is applied to the system), the control logic initializes the CPU by clearing the accumulator, status register, and instruction register (along with whatever else needs to be done).

Following initialization, the control logic causes the CPU to read an instruction from the main system's memory and store it in the instruction register (IR). The control logic then generates whatever sequence of internal control signals are required to execute this particular instruction. All of the control logic's actions are synchronized to the main system clock, and each instruction may require a number of clock cycles.

After a particular instruction has been executed, the control logic causes the next instruction to be loaded from the main system's memory, and so it goes. Finally, remember that the control logic can use the values stored in the status register to make decisions, along the lines of: "*If the zero flag is set I'll do one thing, otherwise I'll do something else*."

Adding the addressing logic

The last major block in the CPU is the addressing logic, which the control logic uses to point to locations in the main system's memory and to the input and output ports. The complexity (or lack thereof) of its addressing logic dictates the sophistication (or lack thereof) of the addressing modes that can be supported by the CPU. As fate would have it, the *DIY Calculator's* CPU offers a reasonably varied potpourri of commonly used addressing modes, with the result that the addressing logic is quite "hairy."

Remember that the *DIY Calculator* has a 16-bit address bus, which is why we show this bus as being twice the width of our 8-bit data bus (Figure 2-19).

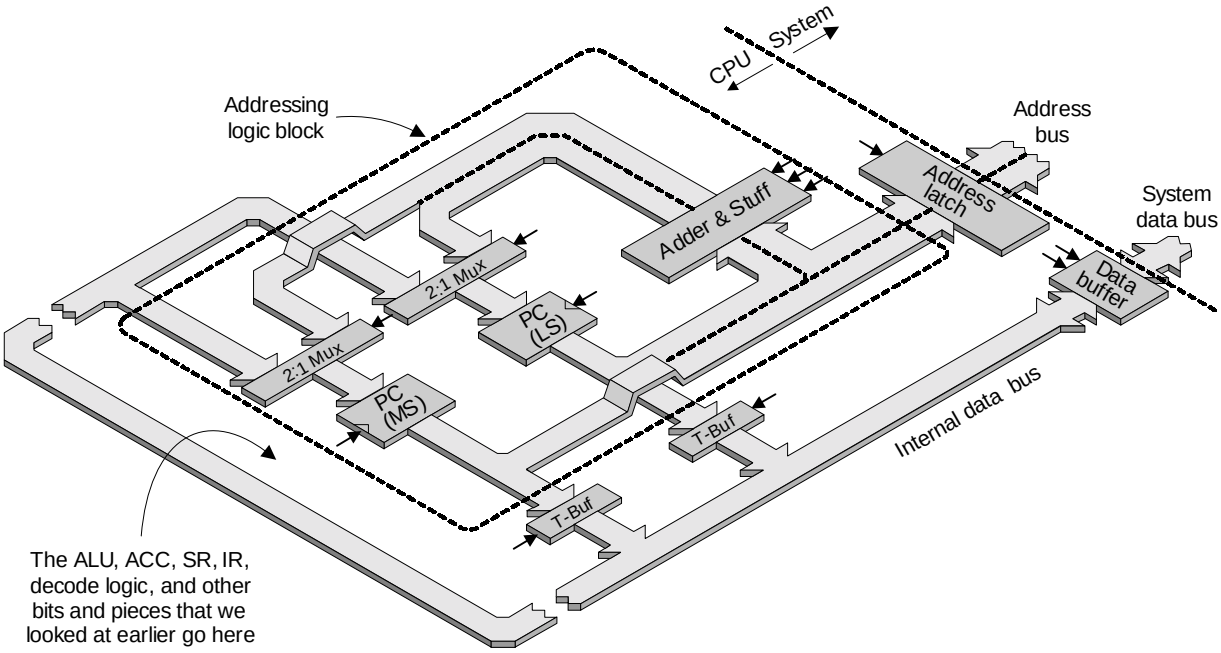


Figure 2-19. Inside the addressing logic block (simplified view)

Inside the addressing logic block are a number of 16-bit registers, each of which may be loaded into the address latch and used to drive the address bus. For the sake of simplicity, Figure 2-19 shows only the main 16-bit register, which is referred to as the *program counter (PC)*. The program counter is actually formed from two 8-bit registers, which we've called PC-MS and PC-LS for the most- and least-significant bytes, respectively. The two halves of the program counter can be clocked independently or together (all of the control signals are driven by the control logic that we introduced in the previous section). Thus, by means of the 2:1 multiplexers driving their inputs, we can individually load each half of the program counter with a value from the internal data bus. Similarly, the *T-Buf* tri-state buffers allow us to read the contents of PC-MS or PC-LS back onto the internal data bus.

In order to point to a location in memory, the control logic loads the outputs from both halves of the program counter into a 16-bit address latch, which, in turn, drives these signals onto the main system's address bus. Once the current value in the program counter is safely stored in the address latch, the control logic may wish to modify the program counter's contents. For example, the control logic might decide to increment the program counter to point to the next location in memory. In this case the control logic would use the 16-bit adder block to add $0001 to the current value in the program counter, and then store the result back into the program counter via the 2:1 multiplexers.

Now, before we proceed any further, we need to take a slightly closer look at the contents of the addressing logic's adder block (Figure 2-20).

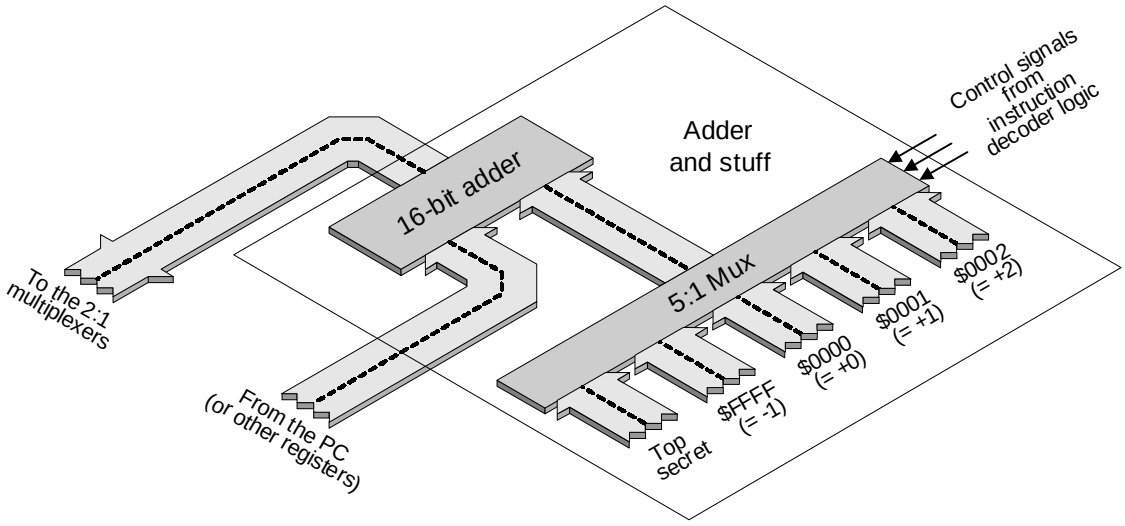


Figure 2-20. The addressing logic's adder block

At the heart of this block is a 16-bit adder, which (not unnaturally) adds two 16-bit values together and generates a 16-bit result. One set of the adder's inputs come from the program counter (or another register), while the other set is generated internally using a 5:1 multiplexer under the direction of the main control logic. In the case of the *DIY Calculator's* CPU, we can chose to modify the value from the program counter (or another register) by adding it to \$FFFF, \$0000, \$0001, or \$0002, which equate to the decimal values -1_{10}, $+0_{10}$, $+1_{10}$, and $+2_{10}$, respectively (we'll consider the fifth, "top secret" option in a moment). Of course, adding –1 to the existing contents of a register is exactly the same as subtracting +1 from it, thereby giving us the ability to decrement the contents of the program counter (or another register) should we so desire.

The addressing logic's registers

In addition to the *program counter (PC)*, the addressing logic contains five more 16-bit registers: two temporary program counters called *TPCA* and *TPCB*, the *stack pointer (SP)*, the *interrupt vector (IV)*, and the *index register (X)* (Figure 2-21).

Earlier we noted that each of these 16-bit registers is actually composed from two 8-bit chunks. For the purposes of these discussions, however, we can simplify things by returning to consider these registers as 16-bit entities, so long as we remember that each half of each register can be controlled independently.

When the CPU is reset, the control logic automatically initializes the program counter to cause it to point to address \$0000, from whence the CPU will retrieve its first instruction.[1] In order to do this, the control logic causes he outputs from the program counter to be loaded into the 16-bit address latch, which, in turn, drives these signals onto the main system's address bus. Once the current value in the program counter has been safety squirreled away in the address latch, the control logic uses our 16-bit adder block to increment the program counter such that it's ready to point to the next memory location.

[1] In the case of the *DIY Calculator*, address \$0000 points to the first location in the ROM. For the purposes of these discussions (and the discussions in the book *How Computers Do Math*), we may assume that the instruction at address \$0000 is an unconditional jump to the first location in the RAM at location \$4000.

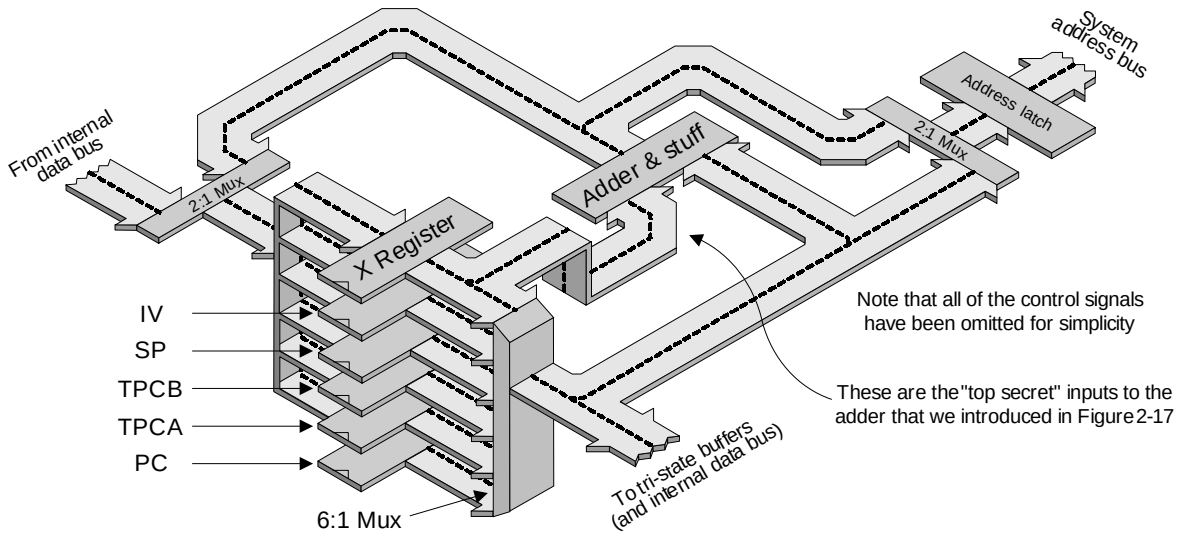


Figure 2-21. The addressing logic's six 16-bit registers

In fact, we can save time by performing both tasks concurrently; that is, at the same time as we're loading the value coming out of the program counter into the address latch, we can also be incrementing this value using the adder block and feeding the result back into the program counter via the 2:1 multiplexers. As usual, the reason this would work is because all of the logical elements forming the loop have internal delays (albeit small ones), which means that the original value in the program counter will have been safely stored in the address latch by the time the new value comes out (see also the discussions in *Appendix C* for more details on the way all of this works).

If the only register in our addressing logic were the program counter, then the only forms of addressing we could support would be the *implied* and *immediate* modes. In order to implement the more interesting addressing modes offered by the *DIY Calculator* (as detailed in *Appendix A*), the other registers are employed as follows:

- The temporary program counter TPCA is used to implement *absolute addressing* (including unconditional and conditional jumps).
- Both TPCA and TPCB are required to implement *indirect addressing*.
- The stack pointer (SP) is used to control stack operations, including the `PUSH` and `POP` instructions and subroutine and interrupt calls and returns.
- The interrupt vector (IV) is used to point to the interrupt service routine (see *Chapter 3* for more details).
- The index register (X) and TPCA are used to implement *indexed addressing*. Also, X, TPCA, and TPCB are used to implement *pre-indexed indirect addressing* and *indirect post-indexed addressing*.

Note especially that the SP, IV, and X registers are not automatically initialized by a reset. Instead, these registers must be loaded explicitly using the `BLDSP` (*"big load stack pointer"*), `BLDIV` (*"big load interrupt vector"*), and `BLDX` (*"big load index register"*) instructions, respectively.

As is illustrated in Figure 2-21, the output from the index register (X) is connected to a 6:1 multiplexer, which feeds it into the adder block in the same way as the other registers (this allows us to increment or decrement its contents using the `INCX` and `DECX` instructions, respectively). However, the index register is somewhat different to the other registers, in that we never actually load its contents directly into the address latch, but instead we use it to modify the values coming out of the other registers.

In order to do this, we also take the outputs from the index register and feed them directly into a second set of inputs to the adder block. This second set of inputs form the "*Top Secret*" signals that we introduced in Figure 2-20. Thus, we can use the 6:1 multiplexer to select the outputs from TPCA (or one of the other registers) and feed them to the adder block, where we now have the ability to add them to the contents of the index register if we so desire.

Furthermore, as shown in Figure 2-21, we've added a new 2:1 multiplexer to feed the address latch. This multiplexer can either select the signals from the 6:1 output multiplexer as before, or it can directly select the outputs from the adder block when we wish to load the sum of the index register and one of the other registers into the address latch.

Last but not least, the interrupt vector (IV) is also somewhat different to the other registers. This is because, once we've loaded it with a value using a `BLDIV` (*"big load interrupt vector"*) instruction, it simply hangs around waiting for an interrupt to occur (this is discussed in detail in *Chapter 3*).

Note: In Figure 2-20, we show the adder as having four sets of hard-wired inputs: $FFFF, $0000, $0001, or $0002, which equate to the decimal values -1_{10}, $+0_{10}$, $+1_{10}$, and $+2_{10}$, respectively. Now it makes sense that we require the ability to add -1_{10} so as to decrement the contents of a register, and $+1_{10}$ to increment the contents of a registers, and $+0_{10}$ to leave the contents of the register unchanged (for reasons that will be made clear in *Appendix C*), but what about the $+2_{10}$ value?

Well, in the cause of conditional jump instructions such as `JZ` (*"jump if zero"*), these have an opcode followed by two operand bytes. If the conditional test fails, we will need to jump over these two operand bytes in order to reach the next instruction (again, the way this actually works is discussed in toe-curling detail in *Appendix C*).

[THIS PAGE IS INTENTIONALLY LEFT BLANK FOR PRINTING PAGINATION]

Chapter 3

The *DIY Calculator's* Interrupt Structure

Interrupts and interrupt handling

Generally speaking, we wish to create programs that can concentrate on the task for which they are intended without being obliged to constantly check to see what's happening in the outside world. In some cases, however, we want the computer's response to be fast and furious when an external situation meriting action does arise. In order to facilitate this sort of thing, CPUs are equipped with a special *interrupt request* (IRQ) input and an *interrupt acknowledge* (IACK) output (Figure 3-1).

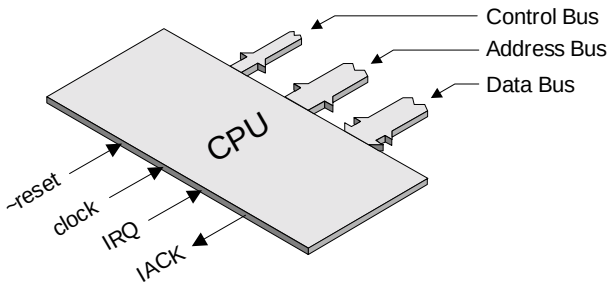


Figure 3-1. The CPU's IRQ and IACK signals

The active state of the *DIY Calculator's* IRQ signal is a logic 0. When the IRQ enters its active state, this fact is stored in a special latching circuit (called the *interrupt latch*) inside the CPU, thereby circumventing the problem of the IRQ going inactive before the CPU gets around to checking it.

The CPU also contains a special status flag called the *interrupt mask (I)*, which is used to enable or disable interrupts, and which can be set or cleared under program control (the status register and its interrupt mask bit were introduced in *Chapter 2*). The CPU supports two instructions, SETIM and CLRIM, which set or clear its interrupt mask, respectively.

By default, the CPU powers up with the interrupt mask in its inactive state, which is a logic 0. Thus, in order for the CPU to be able to "see" an interrupt request, the programmer has to use a SETIM ("*set interrupt mask*") instruction to place the mask in its active state (this instruction also clears the interrupt latch). Similarly, if the programmer subsequently wishes to prevent the CPU from responding to interrupt requests, then he or she can use a CLRIM ("*clear interrupt mask*") instruction to return the mask to its inactive state.

Note: Some CPUs use logic 0 as the interrupt mask's active state, while others (like the *DIY Calculator's* CPU) use logic 1. There aren't any hard-and-fast rules about this: "*You pays your money and you takes your choice*," as the old saying goes.

The CPU checks the state of the interrupt mask every time it completes a machine-code level instruction. If the mask is inactive the CPU simply proceeds to the next instruction, but if the mask is active the CPU takes a peek inside the interrupt latch to determine whether or not an interrupt has been requested (Figure 3-2).

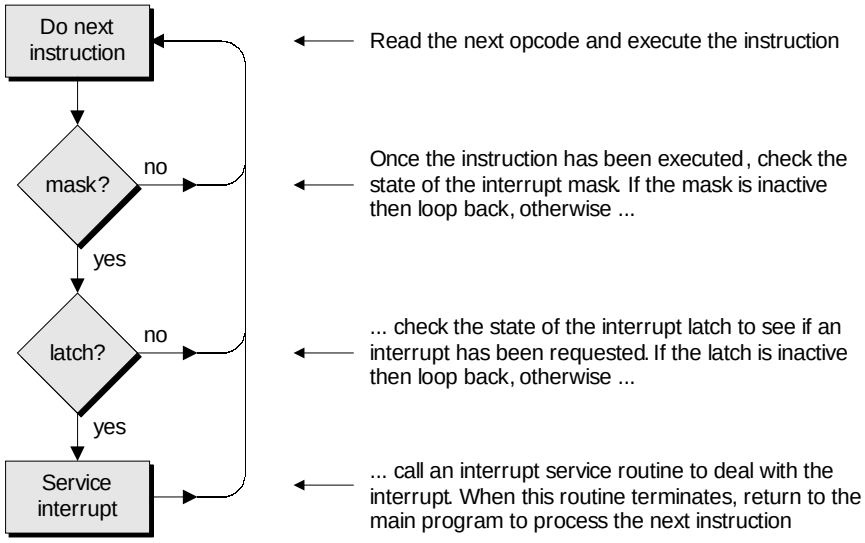


Figure 3-2. High-level flowchart for interrupt handling

If an interrupt *has* been requested, the CPU jumps to a special subroutine called an *interrupt service routine*. Some CPUs contain a hardwired interrupt address that points to another address stored in the main memory. This second address, which is called the *interrupt vector (IV)*, is used to point to the first instruction in the interrupt service routine. By comparison, the *DIY Calculator's* CPU has slightly simpler interrupt structure, in that it contains a 16-bit *interrupt vector (IV)* register that has to be loaded under program control using a `BLDIV` (*"big load interrupt vector"*) instruction. Having the interrupt vector inside the CPU isn't a particularly common technique, but there's no rule book that says a CPU cannot be implemented this way.

An example interrupt sequence

Before we start, it's worth reminding ourselves that when the CPU powers-up or is reset, its interrupt mask is loaded with a logic 0, thereby disabling the CPU's ability to see an interrupt. This means that if we want the *DIY Calculator* to respond to an interrupt, we first have to use a `SETIM` ("*set interrupt mask*") instruction, which loads the interrupt mask with a logic 1.

Also, every time an active (low-going or 1-0-1) pulse is presented to the CPU's `IRQ` (interrupt request) input, this event is stored in the interrupt latch inside the CPU. Bearing all of this in mind, consider the following simple program:

```
## Program to test the DIY Calculator's interrupt capability
MAINDISP: .EQU    $F031       # Output port for main display
SIXLEDS:  .EQU    $F032       # Output port for six LEDs

## Here's the start of the program itself
          .ORG    $4000       # Specify the program's origin

## Initialize a temporary value
INITTEMP:  LDA    %01010101   # Load a simple binary pattern
           STA    [TEMP]      #   and store to temp location
```

```
## Initialize everything else
GETREADY:  LDA    $09         # Load the accumulator
           BLDSP  $4FFF       # Load the stack pointer
           BLDIV  SERVICE     # Load the interrupt vector
           SETIM              # Enable interrupts

## Display 9876543210 over and over and over again
MAINLOOP:  STA    [MAINDISP]  # Store ACC to main display
           DECA               # Decrement the accumulator
           JNN    [MAINLOOP]  # Jump if ACC = $09 thru $00
           LDA    $09         #   else load ACC with $09
           JMP    [MAINLOOP]  #   and do it all again

## Here's the interrupt service routine
SERVICE:   PUSHA              # Push accumulator onto the
stack
           LDA    [TEMP]      # Load ACC with TEMP value
           STA    [SIXLEDS]   #   and write it to the LEDs
           XOR    %11111111   # Invert all bits in ACC
           STA    [TEMP]      #   and store the next value
           POPA               # Get original value from stack
           RTI                # Return to the main program

## Reserve a 1-byte temporary location
TEMP:     .BYTE

## That's all folks
          .END                # End of the program
```

The main body of the test program

As we see, this program is really quite simple. First we define a couple of constant labels called `MAINDISP` and `SIXLEDs`, to which we assign to the addresses of the output ports driving the *DIY Calculator's* main display and six LEDs, respectively.

After defining the origin of the program to be $4000, we arrive at label `INITTEMP` where we load the accumulator with a binary value of %01010101, then we store this value in a temporary location called `TEMP`.

Next, at label `INITSTUF`, we load the accumulator with an initial value of $09; we load the stack pointer with $4FFF; and we initialize the interrupt vector by assigning the `SERVICE` label to it (the assembler will automatically substitute this label for the start address of our interrupt service routine, which is presented later in the program). Once everything is ready to rock-and-roll, the final step in the initialization is to use a `SETIM` instruction to load the CPU's interrupt mask with a logic 1, thereby permitting the CPU to see an interrupt (the `SETIM` also clears the interrupt latch, so as to ensure that the CPU only responds to any *new* interrupt requests).

The core of the main program commences at the `MAINLOOP` label. This is where we start to loop around copying the contents of the accumulator to the main display, decrementing the accumulator, and checking to see if the value in the accumulator is still greater-than-or-equal-to 0. If it is, we jump back to grapple with the next value; if not, we reload the accumulator with $09 and start the loop all over again.

The interrupt service routine

When we eventually run this program, the main loop will keep running forever, unless (a) the *DIY Calculator* is powered-down, (b) the CPU is reset, or (c) an interrupt occurs.

Remember that the CPU checks the state of the interrupt mask after every instruction and, as we've now loaded the mask with a logic 1 (using our `SETIM` instruction), it will subsequently proceed to check the state of the interrupt latch. Whenever the `IRQ` input enters its active state, this will be stored in the CPU's interrupt latch, and the CPU will see the interrupt as soon as it finishes whatever instruction it's currently working on. At this point, the CPU will push the current contents of the program counter onto the stack, followed by the current contents of the status register. The CPU also automatically loads the interrupt mask with a logic 0 to prevent future interrupts from having any effect. The CPU then copies the contents of the interrupt vector (IV) into the program counter (PC), which now points to the first instruction in the interrupt service routine.

Some CPUs also push copies of one or more of the other internal registers onto the stack, such as the accumulator and the index register, because there's a good chance that the act of servicing the interrupt will modify the contents of these registers. Simpler CPUs like the one powering the *DIY Calculator* don't do this automatically, so it's up to us to save the contents of any registers we deem to be important as soon as the interrupt service routine is entered. This explains why the first thing we do upon entering our interrupt service routine at the `SERVICE` label is to use a `PUSHA` instruction to push a copy of the current contents of the accumulator onto the stack. We do this because we know that our example routine is going to modify the accumulator, which would interfere with the main body of the program when we eventually return to it.

Purely for the sake of this example, the only tasks performed by our interrupt service routine are to load the accumulator with the bit pattern stored in our temporary location called `TEMP` (this will be our original %01010101 value the first time the routine is called); copy this value to the port driving the six LEDs; XOR the value with all ones (%11111111), which will invert all of the bits leaving the accumulator containing %10101010; and store this new value back into the temporary location. (The next time we invoke this routine, the value will end up being inverted from %10101010 back to %01010101, and so forth each time the routine is called.)

Once these actions have been completed, we use a `POPA` instruction to restore the contents of the accumulator to their original value when we entered the routine (you will recall that our first action was to push the accumulator into the stack). Finally, we use an `RTI` (*"return from interrupt"*) instruction to terminate the interrupt service routine and return us to the main body of the program. The `RTI` causes the CPU to pop the original value of the status register back off the stack and to then restore the program counter from the stack. Note that the act of popping the status register off the stack will return the interrupt mask to a logic 1 (which was its value when the status register was originally pushed onto the stack), thereby re-enabling the CPU's ability to see any future interrupts (as usual, this will automatically reset the interrupt latch). Meanwhile, the other status flags will be returned to whatever states they were in at the point when the interrupt service routine was first activated.

Testing our example interrupt service routine

Use the assembler to create the simple program presented above and save it into a file called *test-interrupts.asm*. Assemble this program, use the **On/Off** button to power up the *DIY*

Calculator, use the **Memory > Load RAM** command to load the *test-interrupts.ram* machine code file into the calculator's memory, and click the **Run** button to start executing the program .

The main display will immediately start filling up with the character sequence 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 repeated over and over again. Also, observe that all of the LEDs are lit up, thereby indicating that they are currently in an uninitialized state. If everything is happening too quickly and the main display is an eye-watering blur of activity, use the **Setup > System Clock** command to access the appropriate dialog and slow the calculator's clock down to a comfortable level.

Next, click the **Interrupt Request** icon on the main tool bar (the red circle with a diagonal bar). This causes the main program to pause while our interrupt service routine is called. As we discussed above, the first time this routine is called, it will write %01010101 to the port driving the six LEDs (only the right-most six bits will have any effect). The routine will then return control to the main program, which will continue writing to the main display. If we decide to activate the **Interrupt Request** a second time, the interrupt service routine will write a value of %10101010 to the port driving the six LEDs before returning control to the main program.

A few final points to ponder

The interrupt acknowledge output

In the real world, as soon as the CPU begins to respond to an interrupt request, it will automatically place its `IACK` ("*interrupt acknowledge*") output into its active state. The `IACK` signal can therefore be used to inform whichever unit called the interrupt in the first place that the CPU has heard it's plea for help and is in the process of servicing the interrupt. Similarly, as soon as the interrupt service routine is terminated via an `RTI` instruction, the CPU automatically returns its `IACK` output into its inactive state.

Nested interrupts

As we noted earlier, as soon as the CPU sees an interrupt, it will push the current contents of the program counter onto the stack, followed by the current contents of the status register. Next, the CPU copies the contents of the interrupt vector (IV) into the program counter (PC), which now points to the first instruction in the interrupt service routine.

While it is doing all of this, the CPU also automatically loads the interrupt mask with a logic 0 to prevent future interrupts from having any effect. By default, the interrupt mask will continue to contain a logic 0 until this interrupt service routine has completed its mission. Having said this, it is certainly possible to have nested interrupts. In this case, we would need to reload the interrupt vector with the address of a new interrupt service routine and then use a SETIM instruction to load the interrupt mask with a logic 1, thereby enabling future interrupts.

The HALT instruction

It sometimes occurs that the only thing we want the CPU to do is to pause a program and wait for an interrupt to occur. The solution is to use a `HALT` instruction, which occupies only one byte of memory. When the CPU sees a `HALT`, it stops executing the program and commences to generate internal `NOP` (*"no operation"*) instructions. The only way out if the `HALT` is to call an interrupt (or to reset the CPU). Note that, when the interrupt is called, the return address placed on the stack will be for the instruction *following* the `HALT` (pretty cunning, huh?).

Appendix A

Addressing Modes and Instruction Set

Addressing modes

The term *addressing modes* refers to the way in which the CPU determines or resolves the addresses of any data to be used in the execution of its instructions. Different computers can support a wide variety of addressing modes, where the selection of such modes depends both on the computer's architecture and the whims of the designer. The seven addressing modes supported by the *DIY Calculator's* CPU are as follows:

a) *Implied*
b) *Immediate*
c) *Absolute*
d) *Indexed* (or *absolute-indexed*)
e) *Indirect*
f) *Pre-indexed indirect*
g) *Indirect post-indexed*

Implied addressing (imp)

The *implied addressing* mode refers to instructions that comprise only an opcode without an operand; for example, the `INCA` ("*increment accumulator*") instruction. In this case, any data required by the instruction and the destination of any result from the instruction are implied by the instruction itself (Figure A-1).

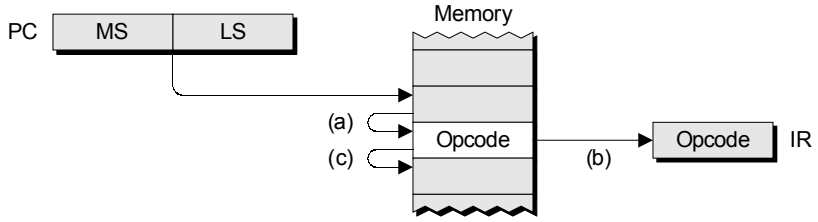


Figure A-1. Implied addressing (using `INCA` as an example)

An implied sequence commences when the PC reaches the opcode for an implied instruction (a), loads that opcode into the IR (b), and increments the PC (c). Recognizing that this is an implied instruction, the CPU executes it and continues on to the next instruction.

Instructions that support implied addressing are: `CLRIM`, `DECA`, `DECX`, `HALT`, `INCA`, `INCX`, `NOP`, `POPA`, `POPSR`, `PUSHA`, `PUSHSR`, `ROLC`, `RORC`, `RTI`, `RTS`, `SETIM`, `SHL`, and `SHR`.

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

Standard immediate addressing (imm)

An instruction using *standard immediate addressing* has one data operand byte following the opcode; for example, `ADD $03` ("*add $03 to the contents of the accumulator*) (Figure A-2).

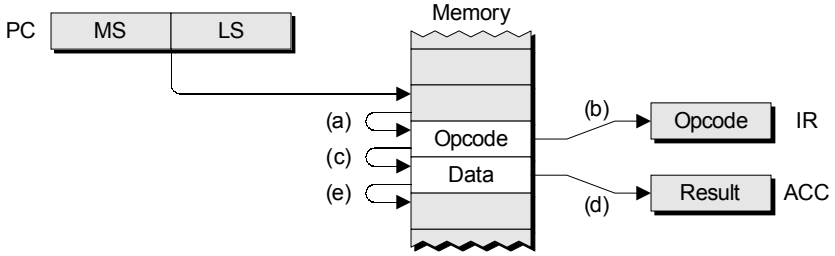


Figure A-2. Standard immediate addressing (using `ADD` as an example)

The sequence commences when the PC reaches the opcode for an immediate instruction (a), loads that opcode into the IR (b), and increments the PC (c). Recognizing that this is an immediate instruction, the CPU reads the data byte pointed to by the PC, executes the instruction using this data, stores the result in the accumulator (d), and increments the PC to look for the next instruction (e).

Instructions that support standard immediate addressing are: `ADD`, `ADDC`, `AND`, `CMPA`, `LDA`, `OR`, `SUB`, `SUBC`, and `XOR`.

Big immediate addressing (imm)

The big immediate addressing mode is very similar to the standard mode, but it refers to instructions that are used to load the 16-bit X, SP, and IV registers. An instruction using big immediate addressing has two data operand bytes following the opcode; for example, `BLDSP $01C4` ("*load $01C4 into the stack pointer*") (Figure A-3).

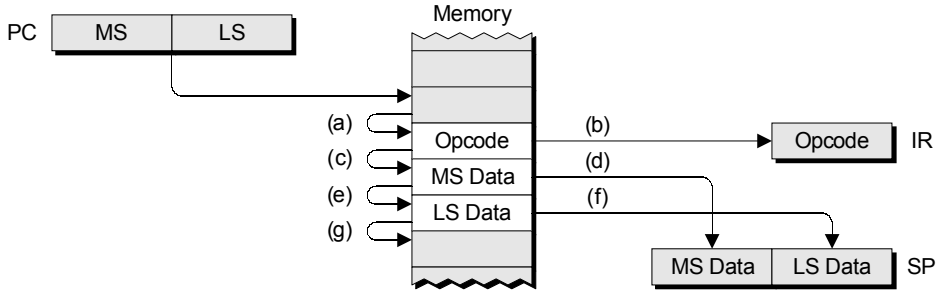


Figure A-3. Big immediate addressing (using `BLDSP` as an example)

The sequence commences when the PC reaches the opcode for an immediate instruction (a), loads that opcode into the IR (b), and increments the PC (c). Recognizing that this is a big immediate instruction, the CPU reads the MS data byte from memory, stores it in the MS byte of the target register (d), and increments the PC (e). The CPU then reads the LS data byte from memory, stores it in the LS byte of the target register (f), and increments the PC to look for the next instruction (g).

Instructions that support big immediate addressing are: `BLDSP`, `BLDX`, and `BLDIV`.

Standard absolute addressing (abs)

An instruction using *standard absolute addressing* has two address operand bytes following its opcode, where these two bytes are used to point to a byte of data (or to a byte in which to store data); for example, `ADD [$4B06]` ("*add the data stored in location $4B06 to the contents of the accumulator*) (Figure A-4).

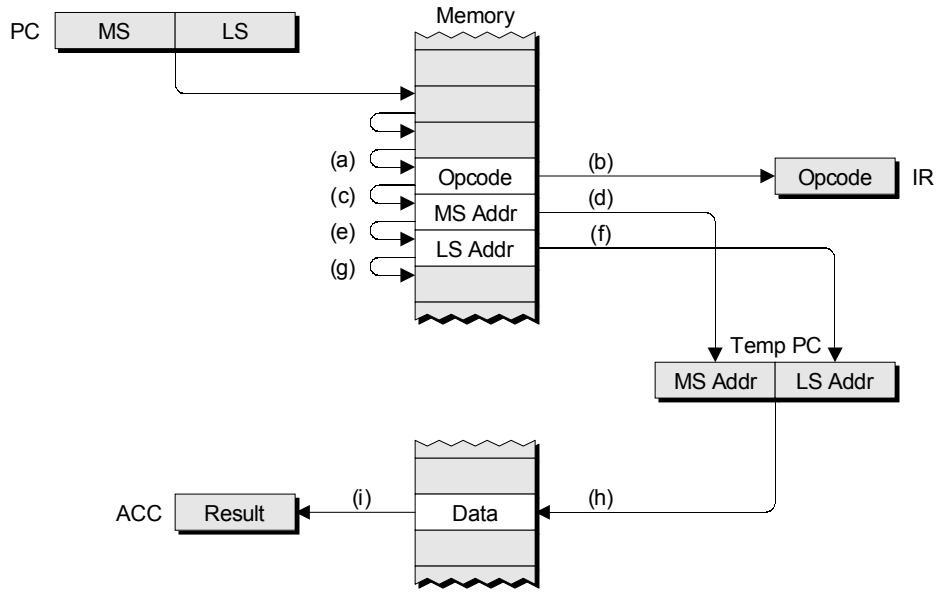


Figure A-4. Standard absolute addressing (using `ADD` as an example)

The sequence commences when the PC reaches the opcode for an absolute instruction (a), loads that opcode into the IR (b), and increments the PC (c). Recognizing that this is a standard absolute instruction, the CPU reads the MS address byte from memory, stores it in the MS byte of one of the temporary PCs (d), and increments the main PC (e). The CPU then reads the LS address byte from memory, stores it in the LS byte of the temporary PC (f), and increments the main PC (g).

The main PC is now "put on hold" while the CPU uses the temporary PC to point to the target address containing the data (h). The CPU executes the original instruction using this data, stores the result into the accumulator (i), and returns control to the main PC to look for the next instruction.

Instructions that support standard absolute addressing are: `ADD`, `ADDC`, `AND`, `CMPA`, `LDA`, `OR`, `STA`, `SUB`, `SUBC`, and `XOR`.

Note: In the case of a `STA` ("*store accumulator*"), the contents of the accumulator would be copied (stored) *into* the data byte in memory.

Note: The jump instructions `JMP`, `JC`, `JNC`, `JN`, `JNN`, `JO`, `JNO`, `JZ`, `JNZ`, and `JSR` can use absolute addressing. However, in this case, the address operand bytes point to the target address which will be loaded into the main PC.

Big absolute addressing (abs)

The *big absolute addressing* mode is very similar to its standard counterpart, but it refers to instructions that affect our 16-bit X, SP, and IV registers. An instruction using big absolute addressing has two address operand bytes following the opcode, and these two bytes are used to point to a *pair* of bytes from which to load or store data; for example, `BLDSP [$4B06]` ("*load the two bytes of data starting at location $4B06 into the stack pointer*) (Figure A-5).

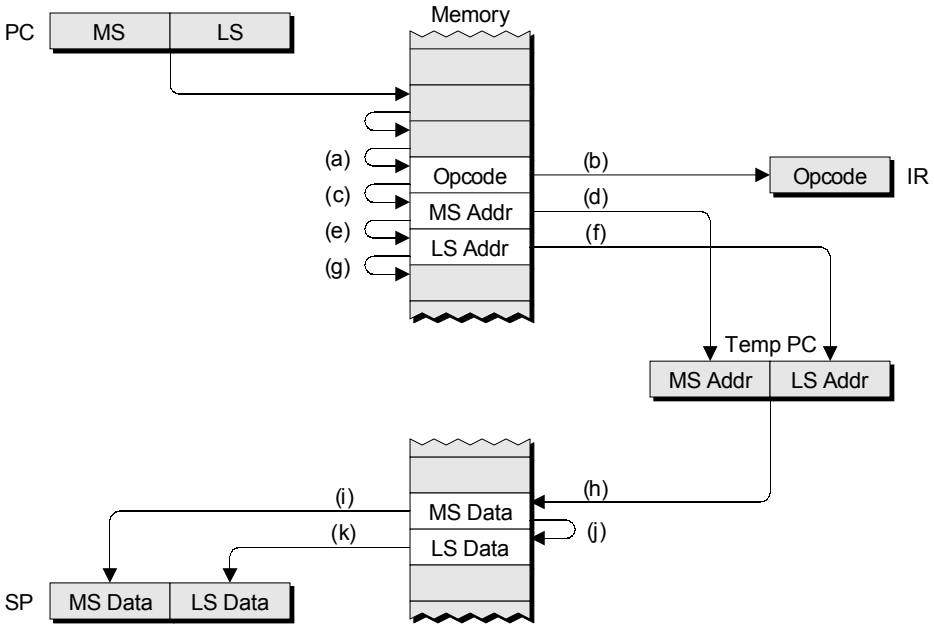


Figure A-5. Big absolute addressing (using `BLDSP` as an example)

The sequence commences when the PC reaches the opcode for an absolute instruction (a), loads that opcode into the IR (b), and increments the PC (c). Recognizing that this is a big absolute instruction, the CPU reads the MS address byte from memory, stores it in the MS byte of one of our temporary PCs (d), and increments the main PC (e). The CPU then reads the LS address byte from memory, stores it in the LS byte of the temporary PC (f), and increments the main PC (g).

The main PC is now "put on hold" while the CPU uses the temporary PC to point to the target address containing the MS data byte (h) and store it in the MS byte of our target register (i). The CPU then increments the temporary PC so as to point to the LS data byte (j) and store it in the LS byte of our target register (k). The CPU now returns control to the main PC to look for the next instruction.

Remember that the above sequence described a "big load" of one of our 16 bit registers (the stack pointer in this example). In the case of a "big store", the contents of the 16-bit register in question would be copied (stored) *into* the two data bytes in memory.

Instructions that support big absolute addressing are: `BLDSP`, `BLDX`, `BLDIV`, `BSTSP`, and `BSTX`.

Indexed addressing (abs-x)

An instruction using the *indexed addressing* mode is very similar to its absolute counterpart, in that it has two address operand bytes following the opcode. However, these two bytes are added to the contents of the *index register (X)*, and the result is used to point to a byte of data (or to a byte in which to store data); for example, `ADD [$4B06,X]` ("*add the data stored in location ($4B06 + X) to the contents of the accumulator*) (Figure A-6).

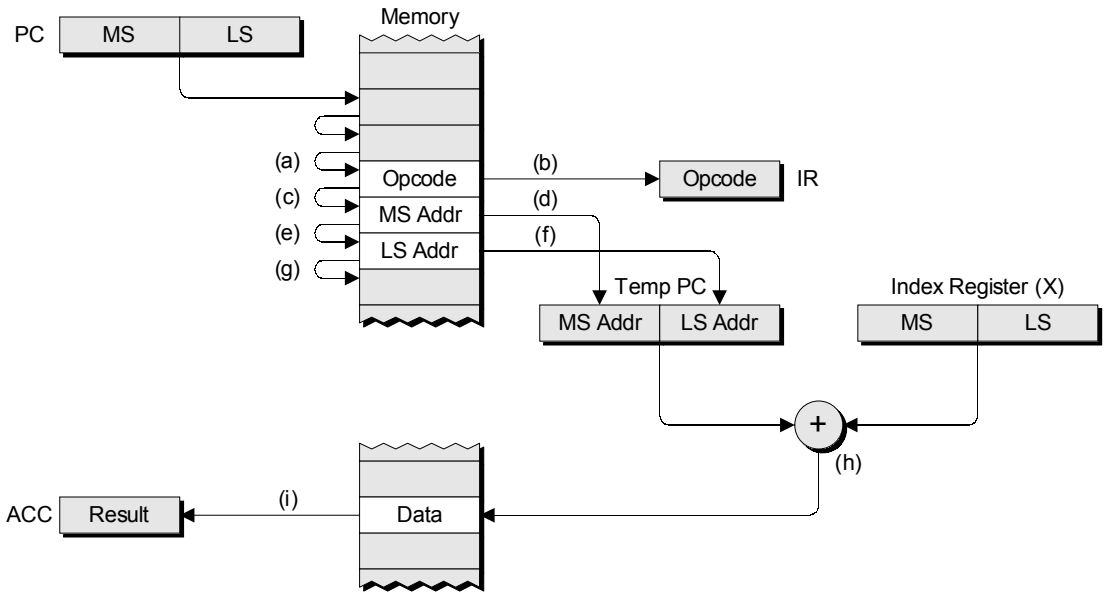


Figure A-6. Indexed addressing (using `ADD` as an example)

The sequence commences when the PC reaches the opcode for an indexed instruction (a), loads that opcode into the IR (b), and increments the PC (c). Recognizing that this is an indexed instruction, the CPU reads the MS address byte from memory, stores it in the MS byte of one of our temporary PCs (d), and increments the main PC (e). The CPU then reads the LS address byte from memory, stores it in the LS byte of the temporary PC (f), and increments the main PC (g).

The main PC is now "put on hold" while the CPU adds the contents of the temporary PC to the contents of the index register and uses the result to point to the target address containing the data (h). The CPU now executes the original instruction using this data and stores the result into the accumulator (i). Finally, the CPU returns control to the main PC to look for the next instruction. (Note that the act of adding the temporary PC to the index register does not affect the contents of the index register. Also note that the index register must have been loaded with a valid value prior to the first indexed instruction).

Instructions supporting this mode are: `ADD`, `ADDC`, `AND`, `CMPA`, `LDA`, `OR`, `STA`, `SUB`, `SUBC`, and `XOR`.

Note: In the case of a `STA` ("*store accumulator*"), the contents of the accumulator would be copied (stored) *into* the data byte in memory. Also, the jump instructions `JMP` and `JSR` can use indexed addressing; in this case, however, the result of adding the contents of the temporary PC to the index register forms the target jump address, which is loaded into the main PC.

Indirect addressing (ind)

As for an absolute instruction, an indirect instruction has two address operand bytes following the opcode. However, these two bytes do not point to the target data themselves, but instead point to the first byte of another pair of address bytes, and it is *these* address bytes that point to the data (or to a byte in which to store data). Thus, an indirect instruction is so-named because it employs a level of indirection. For example, consider an `LDA [[$4B06]]` ("*load the accumulator with the data stored in the location pointed to by the address whose first byte occupies location $4B06*) (Figure A-7).

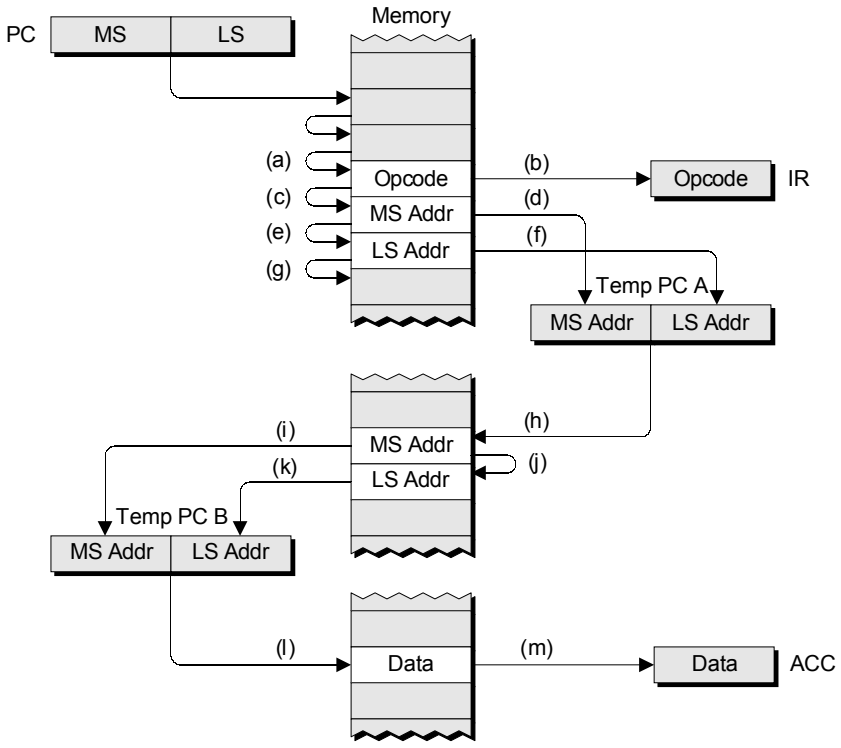


Figure A-7. Indirect addressing (using `LDA` as an example)

When the PC reaches an indirect opcode (a), the CPU loads that opcode into the IR (b), and increments the PC (c). Now the CPU reads the MS address byte from memory, stores it in the MS byte of temporary PC A (d), and increments the main PC (e). Next, the CPU reads the LS address byte from memory, stores it in the LS byte of temporary PC A (f), and increments the main PC (g).

The CPU now employs temporary PC A to read the MS byte of the second address (h), store it in the MS byte of temporary PC B (i), and increment temporary PC A (j). Next, the CPU reads the LS byte of the second address and stores it in the LS byte of temporary PC B (k). The CPU now uses temporary PC B to point to the target data (l) and loads this data into the accumulator (m). Finally, the CPU returns control to the main PC to look for the next instruction.

Instructions that support indirect addressing are `LDA` and `STA`. Also, the jump instructions `JMP` and `JSR` can use indirect addressing; in this case, however, the second address is the target jump address which is loaded into the main PC.

Pre-indexed indirect addressing (x-ind)

The *pre-indexed indirect addressing* mode is a combination of the indexed and indirect modes. This form of addressing is so-named because the address in the opcode bytes is first added to the contents of the index register, and the result points to the first byte of the second address. For example, consider an `LDA [[$4B06,X]]` (“*load the accumulator with the data stored in the location pointed to by the address whose first byte occupies location ($4B06 + X)*) (Figure A-8).

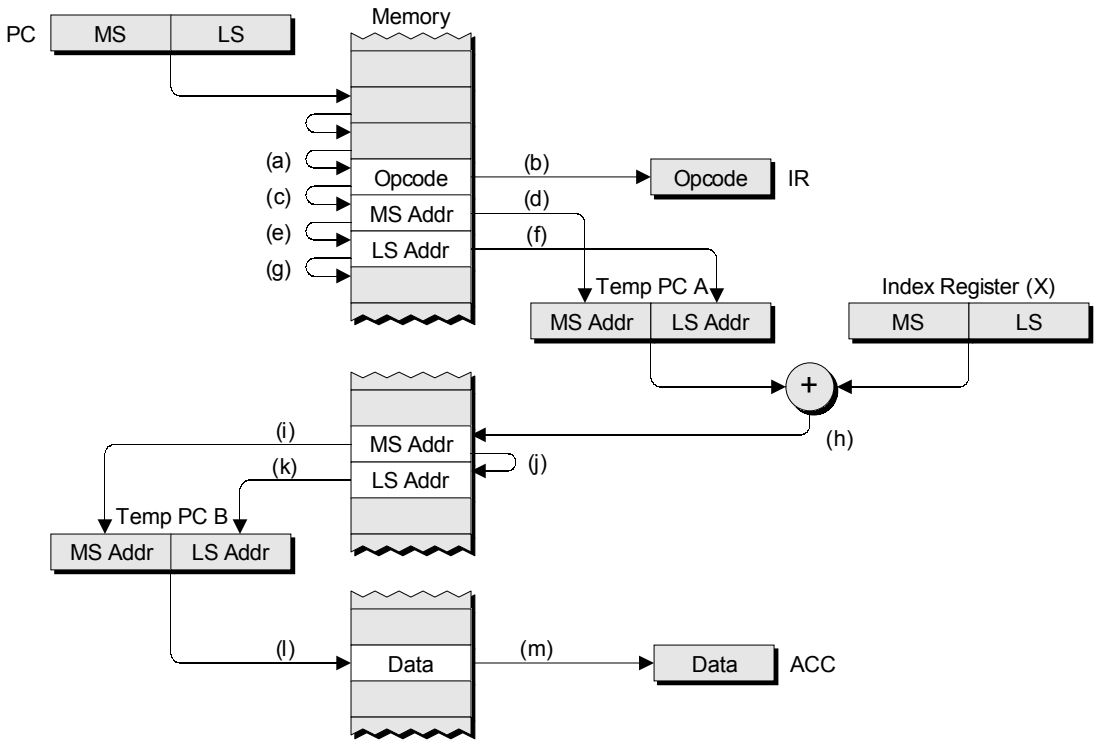


Figure A-8. Pre-indexed indirect addressing (using `LDA` as an example)

When the PC reaches a pre-indexed indirect opcode (a), the CPU loads that opcode into the IR (b), and increments the PC (c). Next, the CPU reads the MS address byte from memory, stores it in the MS byte of temporary PC A (d), and increments the main PC (e). Now the CPU reads the LS address byte from memory, stores it in the LS byte of temporary PC A (f), and increments the main PC (g).

The CPU now adds the contents of temporary PC A to the contents of the index register, uses the result to point to the MS byte of the second address (h), and stores this byte in the MS byte of temporary PC B (i). The CPU then points to the LS byte of the second address (j), stores it in the LS byte of temporary PC B (k), uses temporary PC B to point to the target data (l), and loads this data into the accumulator (m). Finally, the CPU returns control to the main PC to look for the next instruction.

Instructions that support pre-indexed indirect addressing are `LDA` and `STA`. Also, the jump instructions `JMP` and `JSR` can use this form of addressing; however, in this case, the address pointed to by the combination of temporary PC A and the index register is the target jump address which is loaded into the main PC.

Indirect post-indexed addressing (ind-x)

The *indirect post-indexed addressing* mode is similar in concept to pre-indexed indirect mode. In this case, however, the address in the opcode bytes points to a second address, and it is this second address that is added to the contents of the index register to generate the address of the target data. For example, consider an `LDA [[$4B06],X]` (Figure A-9).

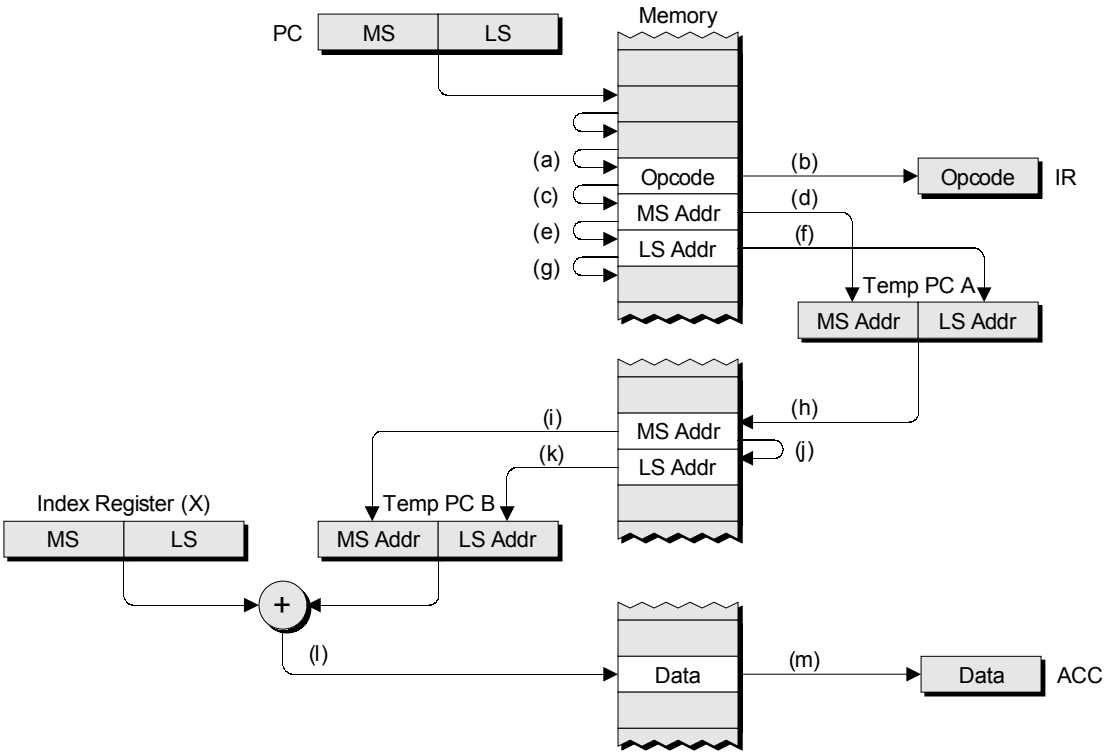


Figure A-9. Indirect post-indexed addressing (using `LDA` as an example)

When the PC reaches an indirect post-indexed opcode (a), the CPU loads that opcode into the IR (b), and increments the PC (c). Now the CPU reads the MS address byte from memory, stores it in the MS byte of temporary PC A (d), and increments the main PC (e). Next, the CPU reads the LS address byte from memory, stores it in the LS byte of temporary PC A (f), and increments the main PC (g).

The CPU uses the contents of temporary PC A to point to the MS byte of the second address (h), and stores this byte in the MS byte of temporary PC B (i). The CPU then increments temporary PC A to point to the LS byte of the second address (j), and stores this byte in the LS byte of temporary PC B (k). Now the CPU adds the contents of temporary PC B to the contents of the index register, uses the result to point to the target data (l), and loads this data into the accumulator (m). Finally, the CPU returns control to the main PC to look for the next instruction.

Instructions that support indirect post-indexed addressing are `LDA` and `STA`. Also, the jump instructions `JMP` and `JSR` can use this form of addressing; in this case, however, the address pointed to by the combination of temporary PC B and the index register is the target jump address which is loaded into the main PC.

Instruction set summary

Control	NOP	No-operation, CPU doesn't do anything.
	HALT	Generate internal NOPs until an interrupt occurs
	SETIM	Set the interrupt mask flag in the status register.
	CLRIM	Clear the interrupt mask flag in the status register
Arithmetic	ADD	Add data in memory to the accumulator.
	ADDC	Like an ADD, but include contents of the carry flag.
	SUB	Subtract data in memory from the accumulator.
	SUBC	Like a SUB, but include contents of the carry flag
Logical	AND	AND data in memory to the accumulator.
	OR	OR data in memory to the accumulator.
	XOR	XOR data in memory to the accumulator.
Comparison	CMPA	Compare data in memory to the accumulator
Shifts & Rotates	SHL	Shift the accumulator left 1 bit (arithmetic shift).
	SHR	Shift the accumulator right 1 bit (arithmetic shift).
	ROLC	Rotate the accumulator left 1 bit (through carry flag).
	RORC	Rotate the accumulator right 1 bit (through carry flag).
Increments & Decrements	INCA	Increment the accumulator.
	DECA	Decrement the accumulator.
	INCX	Increment the index register.
	DECX	Decrement the index register.
Loads & Stores	LDA	Load data in memory into the accumulator
	STA	Store data in the accumulator into memory.
	BLDX	Load data in memory into the index register
	BSTX	Store data in the index register into memory.
	BLDSP	Load data in memory into the stack pointer.
	BSTSP	Store data in the stack pointer into memory.
	BLDIV	Load data in memory into the interrupt vector.
Push & Pop	PUSHA	Push the accumulator onto the stack.
	POPA	Pop the accumulator from the stack.
	PUSHSR	Push the status register onto the stack.
	POPSR	Pop the status register from the stack.
Jumps	JMP	Jump to a new memory location
	JSR	Jump to a subroutine.
	JZ	Jump if the result was zero.
	JNZ	Jump if the result wasn't zero.
	JN	Jump if the result was negative.
	JNN	Jump if the result wasn't negative.
	JC	Jump if the result generated a carry.
	JNC	Jump if the result didn't generate a carry.
	JO	Jump if the result generated an overflow
	JNO	Jump if the result didn't generate an overflow.
Returns	RTS	Return from a subroutine.
	RTI	Return from an interrupt

Table A-1. The *DIY Calculator's* instructions by category

	imp		imm		abs		abs-x		ind		x-ind		ind-x		flags				
	op	#	op	#	op	#	op	#	op	#	op	#	op	#	I	O	N	Z	C
ADD			$10	2	$11	3	$12	3							-	O	N	Z	C
ADDC			$18	2	$19	3	$1A	3							-	O	N	Z	C
AND			$30	2	$31	3	$32	3							-	-	N	Z	-
BLDIV			$F0	3	$F1	3									-	-	-	-	-
BLDSP			$50	3	$51	3									-	-	-	-	-
BLDX			$A0	3	$A1	3									-	-	-	-	-
BSTSP					$59	3									-	-	-	-	-
BSTX					$A9	3									-	-	-	-	-
CLRIM	$09	1													0	-	-	-	-
CMPA			$60	2	$61	3	$62	3							-	-	-	≥	≥
DECA	$81	1													-	-	N	Z	-
DECX	$83	1													-	-	-	Z	-
HALT	$01	1													-	-	-	-	-
INCA	$80	1													-	-	N	Z	-
INCX	$82	1													-	-	-	Z	-
JC					$E1	3									-	-	-	-	-
JMP					$C1	3	$C2	3	$C3	3	$C4	3	$C5	3	-	-	-	-	-
JN					$D9	3									-	-	-	-	-
JNC					$E6	3									-	-	-	-	-
JNN					$DE	3									-	-	-	-	-
JNO					$EE	3									-	-	-	-	-
JNZ					$D6	3									-	-	-	-	-

Table A-2a. Instruction set summary (continued on next page)

Legend			**Flags**			**Addressing Modes**		
op	=	Opcode	Z	=	Zero	imp	=	Implied
$	=	Hexadecimal value	N	=	Negative	imm	=	Immediate
#	=	Number of bytes	C	=	Carry	abs	=	Absolute
-	=	No change	O	=	Overflow	abs-x	=	Indexed
≥	=	Magnitude comparison	I	=	Interrupt Mask	ind	=	Indirect
↔	=	Shift or rotate thru carry bit				x-ind	=	Pre-indexed indirect
Φ	=	Restored by popping status register				ind-x	=	Indirect post-indexed

	imp		imm		abs		abs-x		ind		x-ind		ind-x		flags				
	op	#	op	#	op	#	op	#	op	#	op	#	op	#	I	O	N	Z	C
JO					$E9	3									-	-	-	-	-
JSR					$C9	3	$CA	3	$CB	3	$CC	3	$CD	3	-	-	-	-	-
JZ					$D1	3									-	-	-	-	-
LDA			$90	2	$91	3	$92	3	$93	3	$94	3	$95	3	-	-	N	Z	-
NOP	$00	1													-	-	-	-	-
OR			$38	2	$39	3	$3A	3							-	-	N	Z	-
POPA	$B0	1													-	-	N	Z	-
POPSR	$B1	1													Φ	Φ	Φ	Φ	Φ
PUSHA	$B2	1													-	-	-	-	-
PUSHSR	$B3	1													-	-	-	-	-
ROLC	$78	1													-	-	N	Z	↔
RORC	$79	1													-	-	N	Z	↔
RTI	$C7	1													Φ	Φ	Φ	Φ	Φ
RTS	$CF	1													-	-	-	-	-
SETIM	$08	1													1	-	-	-	-
SHL	$70	1													-	-	N	Z	↔
SHR	$71	1													-	-	N	Z	↔
STA					$99	3	$9A	3	$9B	3	$9C	3	$9D	3	-	-	-	-	-
SUB			$20	2	$21	3	$22	3							-	O	N	Z	C
SUBC			$28	2	$29	3	$2A	3							-	O	N	Z	C
XOR			$40	2	$41	3	$42	3							-	-	N	Z	-

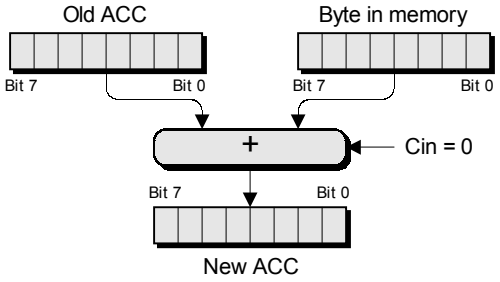
Table A-2b. Instruction set summary (continued from previous page)

Legend			**Flags**			**Addressing Modes**		
op	=	Opcode	Z	=	Zero	imp	=	Implied
$	=	Hexadecimal value	N	=	Negative	imm	=	Immediate
#	=	Number of bytes	C	=	Carry	abs	=	Absolute
-	=	No change	O	=	Overflow	abs-x	=	Indexed
≥	=	Magnitude comparison	I	=	Interrupt Mask	ind	=	Indirect
↔	=	Shift or rotate thru carry bit				x-ind	=	Pre-indexed indirect
Φ	=	Restored by popping status register				ind-x	=	Indirect post-indexed

ADD (Add without carry)

Description

This instruction adds the contents of a byte of data in memory to the current contents of the accumulator and stores the result in the accumulator (the contents of the memory are not affected). Note that the result is not affected by the contents of the carry flag, because the carry-in to the ALU is forced to logic 0. See also the corresponding `ADDC` instruction.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imm	2	$10	`ADD $03`	Add $03 to the ACC.
abs	3	$11	`ADD [$4C76]`	Add the contents of memory location $4C76 to the ACC
abs-x	3	$12	`ADD [$4C76,X]`	Add the contents of a memory location to the ACC, where the address of the memory location is $4C76 plus the contents of the X register

Flags affected

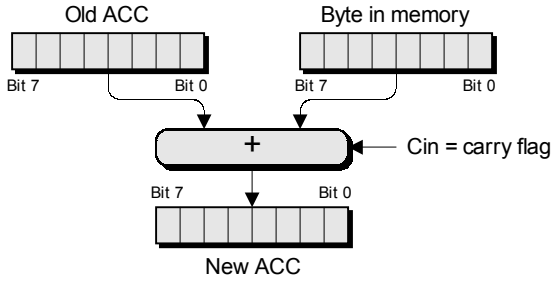
- O Set to 1 if the result overflows; otherwise cleared to 0
- N Set to 1 if the MS bit of the result is 1; otherwise cleared to 0
- Z Set to 1 of all of the bits in the result are 0; otherwise cleared to 0
- C Set to 1 if there is a carry out from the addition; otherwise cleared to 0

Registers			**Flags**			**Addressing Modes**			**Other**		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

ADDC (Add with carry)

Description

This instruction adds the contents of a byte of data in memory (along with the current contents of the carry flag) to the current contents of the accumulator and stores the result in the accumulator (the contents of the memory are not affected). See also the corresponding `ADD` instruction.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imm	2	$18	`ADDC $03`	Add $03 to the ACC
abs	3	$19	`ADDC [$4C76]`	Add the contents of memory location $4C76 to the ACC
abs-x	3	$1A	`ADDC [$4C76,X]`	Add the contents of a memory location to the ACC, where the address of the memory location is $4C76 plus the contents of the X register

Flags affected

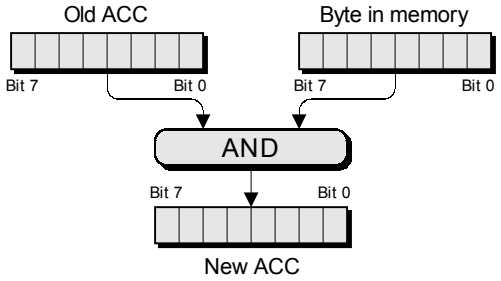
- O Set to 1 if the result overflows; otherwise cleared to 0
- N Set to 1 if the MS bit of the result is 1; otherwise cleared to 0
- Z Set to 1 of all of the bits in the result are 0; otherwise cleared to 0
- C Set to 1 if there is a carry out from the addition; otherwise cleared to 0

Registers			**Flags**			**Addressing Modes**			**Other**		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

AND (Logical operation)

Description

This instruction logically ANDs the contents of a byte of data in memory with the current contents of the accumulator and stores the result in the accumulator (the contents of the memory are not affected). Note that this is a bit-wise operation, which means that bit 0 of the old ACC is AND-ed with bit 0 of the memory to generate bit 0 of the new ACC. Similarly, bit 1 is AND-ed with bit 1, bit 2 with bit 2, and so forth. See also the `OR` and `XOR` instructions.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imm	2	$30	`AND $03`	Logically AND $03 with the ACC
abs	3	$31	`AND [$4C76]`	Logically AND the contents of memory location $4C76 with the ACC
abs-x	3	$32	`AND [$4C76,X]`	Logically AND the contents of a memory location with the ACC, where the address of the memory location is $4C76 plus the contents of the X register

Flags affected

N Set to 1 if the MS bit of the result is 1; otherwise cleared to 0

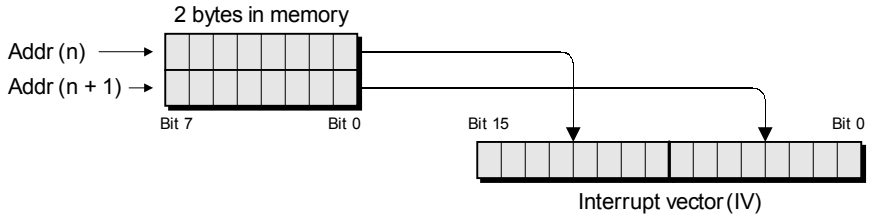
Z Set to 1 of all of the bits in the result are 0; otherwise cleared to 0

Registers			**Flags**			**Addressing Modes**			**Other**		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

BLDIV ("Big" load the interrupt vector)

Description

This instruction loads two contiguous bytes of memory into the 16-bit interrupt vector, MS byte first (the contents of the memory are not affected). Note that, unlike the `BLDSP` and `BLDX` instructions that have corresponding store instructions (`BSTSP` and `BSTX`), there is no `BSTIV`. This is because the contents of the IV register can only be changed explicitly using a `BLDIV`, which means that the programmer always knows what it contains (a programmer who doesn't is a nincompoop of no account).



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imm	3	$F0	`BLDIV $6100`	Load the IV with $6100
abs	3	$F1	`BLDIV [$4C76]`	Load the IV with two bytes from memory, where the first (MS) byte is located at address $4C76

Flags affected

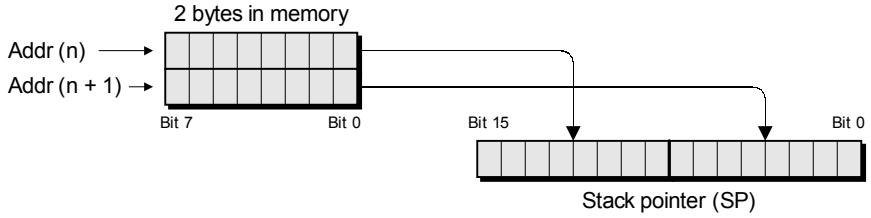
None

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

BLDSP ("Big" load the stack pointer)

Description

This instruction loads two contiguous bytes of memory into the 16-bit stack pointer, MS byte first (the contents of the memory are not affected).



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imm	3	$50	`BLDSP $4FFF`	Load the SP with $4FFF
abs	3	$51	`BLDSP [$4C76]`	Load the SP with two bytes from memory, where the first (MS) byte is located at address $4C76

Flags affected

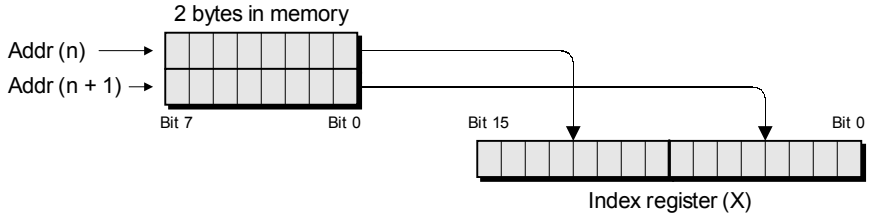
None

Registers			**Flags**			**Addressing Modes**			**Other**		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

BLDX (“Big” load the index register)

Description

This instruction loads two contiguous bytes of memory into the 16-bit index register, MS byte first (the contents of the memory are not affected).



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imm	3	$A0	`BLDX $0C64`	Load the X register with $0C64
abs	3	$A1	`BLDX [$4C76]`	Load the X register with two bytes from memory, where the first (MS) byte is located at address $4C76

Flags affected

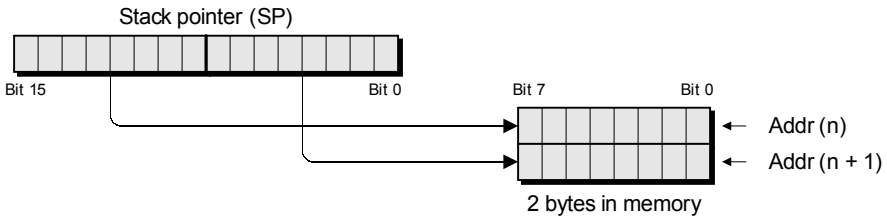
None

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

BSTSP ("Big" store the stack pointer)

Description

This instruction stores the current contents of the 16-bit stack pointer into two contiguous bytes of memory, MS byte first (the contents of the stack pointer are not affected).



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
abs	3	$59	`BSTSP [$4C76]`	Store the SP into two bytes of memory, where the first (MS) byte is located at address $4C76

Flags affected

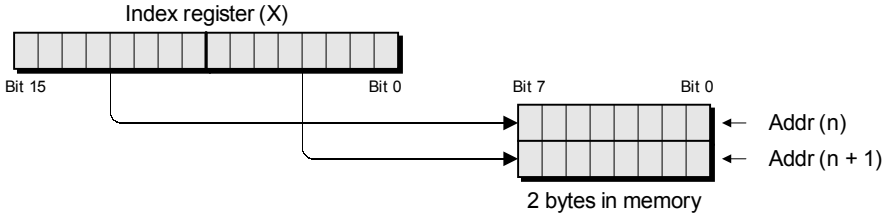
None

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

BSTX ("Big" store the index register)

Description

This instruction stores the current contents of the 16-bit index register into two contiguous bytes of memory, MS byte first (the contents of the index register are not affected).



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
abs	3	$A9	`BSTX [$4C76]`	Store the X register into two bytes of memory, where the first (MS) byte is located at address $4C76

Flags affected

None

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

CLRIM (Clear the interrupt mask)

Description

This instruction clears the interrupt mask bit in the status register to logic 0, thereby preventing the CPU from seeing any future interrupts. See also the `SETIM` instruction in this appendix and the discussions on interrupts in *Chapter 3*.

Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imp	1	$09	`CLRIM`	Load the interrupt mask bit with 0

Flags affected

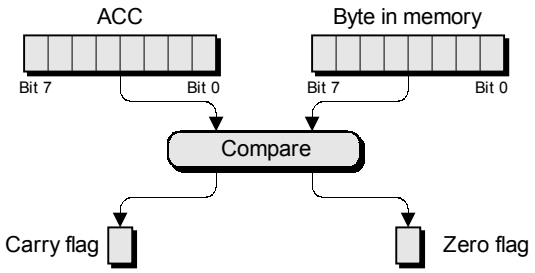
I Loaded with logic 0

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

CMPA (Compare accumulator to byte in memory)

Description

This instruction compares the contents of the accumulator to a byte in memory. The instruction assumes that both quantities represent unsigned binary values. Based on this assumption, the carry flag is set to logic 1 if the value in the accumulator is the greater (otherwise it's cleared to logic 0), while the zero flag is set to logic 1 if the values are equal (otherwise it's cleared to logic 0). The original values in the accumulator and memory are not modified in any way.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imm	2	$60	`CMPA $03`	Compare the contents of the ACC with $03
abs	3	$61	`CMPA [$4C76]`	Compare the contents of the ACC with the contents of memory location $4C76
abs-x	3	$62	`CMPA [$4C76,X]`	Compare the contents of the ACC with the contents of a memory location, where the address of the memory location is $4C76 plus the contents of the X register

Flags affected

Z Set to 1 if the two values are equal; otherwise cleared to 0

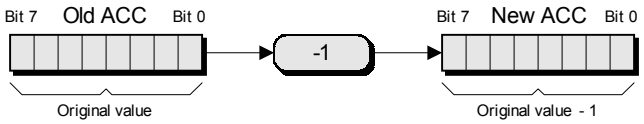
C Set to 1 if the value in the accumulator is the greater; otherwise cleared to 0

Registers			**Flags**			**Addressing Modes**			**Other**		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

DECA (Decrement the contents of the accumulator)

Description

This instruction decrements (subtracts 1 from) the existing contents of the accumulator. See also the counterpart to this instruction, `INCA`, and the somewhat similar instructions `DECX` and `INCX`.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imp	1	$81	`DECA`	Decrement (subtract 1 from) the contents of the ACC

Flags affected

N Set to 1 if the MS bit of the result is 1; otherwise cleared to 0

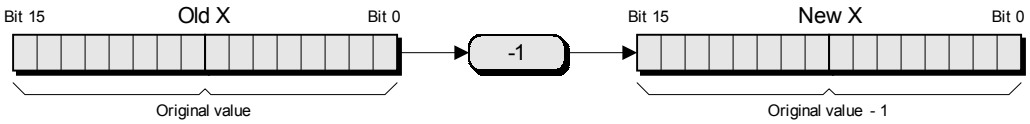
Z Set to 1 of all of the bits in the result are 0; otherwise cleared to 0

Registers			**Flags**			**Addressing Modes**			**Other**		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

DECX (Decrement the contents of the index register)

Description

This instruction decrements (subtracts 1 from) the contents of the 16-bit index register. See also the counterpart to this instruction, `INCX`, and the somewhat similar instructions `DECA` and `INCA`. Note that this instruction only modifies the Z flag (unlike `DECA` which modifies both the Z and N flags).



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imp	1	$83	`DECX`	Decrement (subtract 1 from) the contents of the X register

Flags affected

Z Set to 1 of all of the bits in the result (the new contents of the X register) are 0; otherwise cleared to 0.

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

HALT (Halt the CPU)

Description

This instruction instructs the CPU to cease processing instructions from the memory and to start performing *internal* `NOP` (no-operation) instructions. Left to its own devices, the CPU will continue to perform internal `NOPs` until the end of time, and the only way to override the `HALT` is for the CPU to receive an interrupt (or for it to be reset). See also the `NOP` instruction in this appendix and the discussions on interrupts in *Chapter 3*.

Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imp	1	$01	`HALT`	Halts the CPU and causes it to perform internal `NOP` instructions

Flags affected

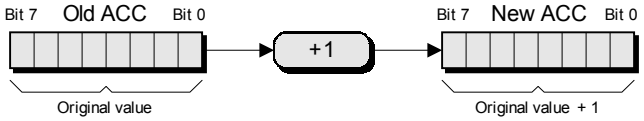
None

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

INCA (Increment the contents of the accumulator)

Description

This instruction increments (adds 1 to) the contents of the accumulator. See also the counterpart to this instruction, `DECA`, and the somewhat similar instructions `DECX` and `INCX`.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imp	1	$80	`INCA`	Increment (add 1 to) the contents of the ACC

Flags affected

N Set to 1 if the MS bit of the result is 1; otherwise cleared to 0.

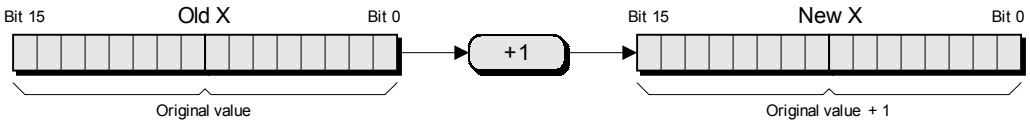
Z Set to 1 of all of the bits in the result are 0; otherwise cleared to 0. Note that the reason incrementing the ACC can result in it containing zero is if its original value were all 1s; that is, $FF in hexadecimal.

Registers			**Flags**			**Addressing Modes**			**Other**		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

INCX (Increment the contents of the index register)

Description

This instruction increments (adds 1 to) the contents of the 16-bit index register. See also the counterpart to this instruction, `DECX`, and the somewhat similar instructions `DECA` and `INCA`. Note that this instruction only modifies the Z flag (unlike `INCA` which modifies both the Z and N flags).



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imp	1	$82	`INCX`	Increment (add 1 to) the contents of the X register

Flags affected

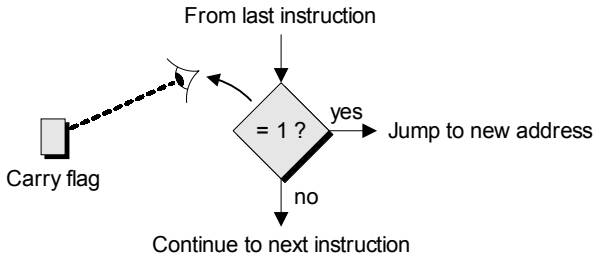
Z Set to 1 of all of the bits in the result (the new contents of the X register) are 0; otherwise cleared to 0. Note that the reason incrementing the X register can result in it containing zero is if its original value were all 1s; that is, $FFFF in hexadecimal.

Registers			**Flags**			**Addressing Modes**			**Other**		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

JC (Jump if carry)

Description

This instruction is used to change the “flow” of the program by causing the CPU to jump to a new address if the carry status flag is `TRUE` (contains a logic 1, thereby indicating that the previous instruction generated a carry), otherwise the CPU ignores the operand and continues to the next instruction. See also the corresponding `JNC` instruction.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
abs	3	$E1	`JC [$4C76]`	If the carry flag contains a logic 1 then jump to address $4C76; otherwise continue to the next instruction

Flags affected

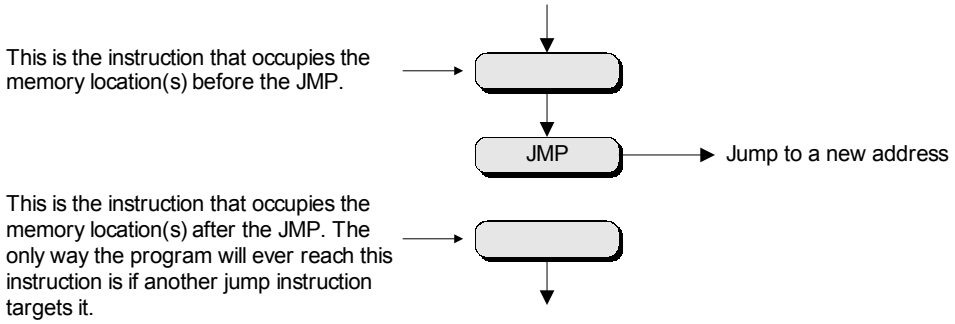
None

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

JMP (Jump unconditionally)

Description

This instruction is used to change the "flow" of the program by causing the CPU to unconditionally jump to a new address. See also the somewhat related `JSR` instruction.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
abs	3	$C1	`JMP [$4C76]`	Jump to address $4C76
abs-x	3	$C2	`JMP [$4C76,X]`	Add $4C76 to the contents of the X register to form the target address $xxxx and jump to this target address
ind	3	$C3	`JMP [[$4C76]]`	Read the target address $xxxx stored in the two bytes starting at address $4C76, and then jump to this target address
x-ind	3	$C4	`JMP [[$4C76,X]]`	Add $4C76 to the contents of the X register to form a new address $zzzz. Read the target address $xxxx stored in the two bytes starting at address $zzzz and jump to this target address
ind-x	3	$C5	`JMP [[$4C76],X]`	Read the address $zzzz stored in the two bytes starting at address $4C76, then add $zzzz to the contents of the X register to form the target address $xxxx and jump to this target address

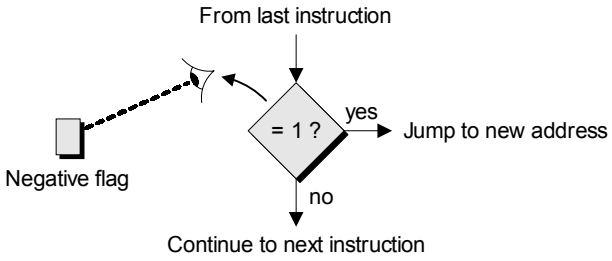
Flags affected

None

JN (Jump if negative)

Description

This instruction is used to change the “flow” of the program by causing the CPU to jump to a new address if the negative status flag is `TRUE` (contains a logic 1, thereby indicating that the result from the previous instruction was negative), otherwise the CPU ignores the operand and continues to the next instruction. See also the corresponding `JNN` instruction.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
abs	3	$D9	`JN [$4C76]`	If the negative flag contains a logic 1 then jump to address $4C76; otherwise continue to the next instruction

Flags affected

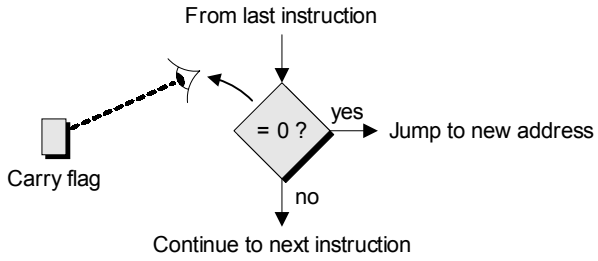
None

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

JNC (Jump if not carry)

Description

This instruction is used to change the "flow" of the program by causing the CPU to jump to a new address if the carry status flag is `FALSE` (contains a logic 0, thereby indicating that the previous instruction did not generate a carry), otherwise the CPU ignores the operand and continues to the next instruction. See also the corresponding `JC` instruction.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
abs	3	$E6	`JNC [$4C76]`	If the carry flag contains a logic 0 then jump to address $4C76; otherwise continue to the next instruction

Flags affected

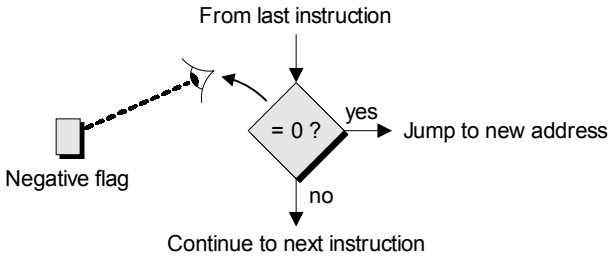
None

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

JNN (Jump if not negative)

Description

This instruction is used to change the “flow” of the program by causing the CPU to jump to a new address if the negative status flag is `FALSE` (contains a logic 0, thereby indicating that the result from the previous instruction was positive (not negative)), otherwise the CPU ignores the operand and continues to the next instruction. See also the corresponding `JN` instruction.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
abs	3	$DE	`JNN [$4C76]`	If the negative flag contains a logic 0 then jump to address $4C76; otherwise continue to the next instruction

Flags affected

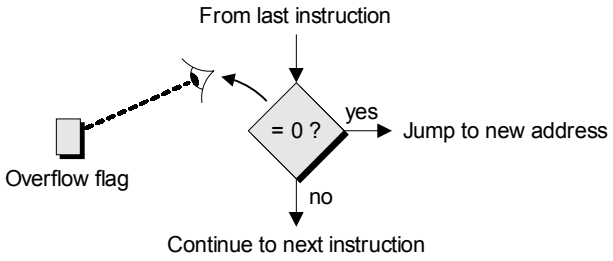
None

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

JNO (Jump if not overflow)

Description

This instruction is used to change the "flow" of the program by causing the CPU to jump to a new address if the overflow status flag is `FALSE` (contains a logic 0, thereby indicating that the previous instruction did not generate an overflow), otherwise the CPU ignores the operand and continues to the next instruction. See also the corresponding `JO` instruction.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
abs	3	$EE	`JNO [$4C76]`	If the overflow flag contains a logic 0 then jump to address $4C76; otherwise continue to the next instruction

Flags affected

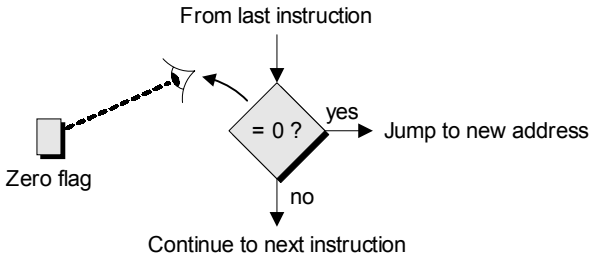
None

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

JNZ (Jump if not zero)

Description

This instruction is used to change the “flow” of the program by causing the CPU to jump to a new address if the zero status flag is `FALSE` (contains a logic 0, thereby indicating that result from the previous operation was non-zero), otherwise the CPU ignores the operand and continues to the next instruction. See also the corresponding `JZ` instruction.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
abs	3	$D6	`JNZ [$4C76]`	If the zero flag contains a logic 0 then jump to address $4C76; otherwise continue to the next instruction

Flags affected

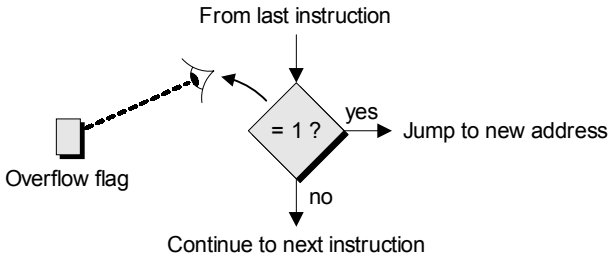
None

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

JO (Jump if overflow)

Description

This instruction is used to change the "flow" of the program by causing the CPU to jump to a new address if the overflow status flag is `TRUE` (contains a logic 1, thereby indicating that the previous instruction generated an overflow), otherwise the CPU ignores the operand and continues to the next instruction. See also the corresponding `JNO` instruction.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
abs	3	$E9	`JO [$4C76]`	If the overflow flag contains a logic 1 then jump to address $4C76; otherwise continue to the next instruction

Flags affected

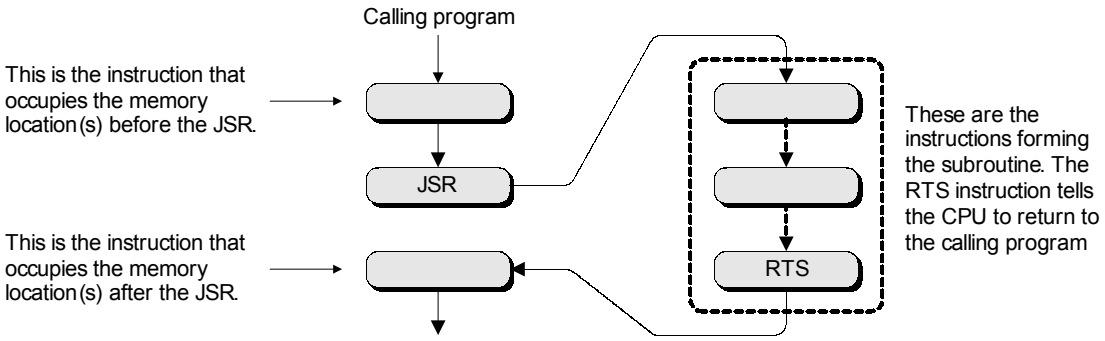
None

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

JSR (Jump to a subroutine)

Description

This instruction is used to change the "flow" of the program by causing the CPU to jump to a subroutine. Note that the CPU automatically places a 2-byte return address on the top of the stack before jumping to the subroutine. See also the related `RTS` instruction.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
abs	3	$C9	`JSR [$4C76]`	Jump to the subroutine at address $4C76
abs-x	3	$CA	`JSR [$4C76,X]`	Add $4C76 to the contents of the X reg to form the target address of the sub-routine ($xxxx) and jump to this address
ind	3	$CB	`JSR [[$4C76]]`	Read the target address of the subroutine ($xxxx) stored in the two bytes starting at address $4C76, then jump to this address
x-ind	3	$CC	`JSR [[$4C76,X]]`	Add $4C76 to the contents of the X register to form a new address $zzzz. Read the target address of the subroutine ($xxxx) stored in the two bytes starting at address $zzzz and jump to this address
ind-x	3	$CD	`JSR [[$4C76],X]`	Read the address $zzzz stored in the two bytes starting at address $4C76, then add $zzzz to the contents of the X register to form the target address of the subroutine ($xxxx) and jump to this target address

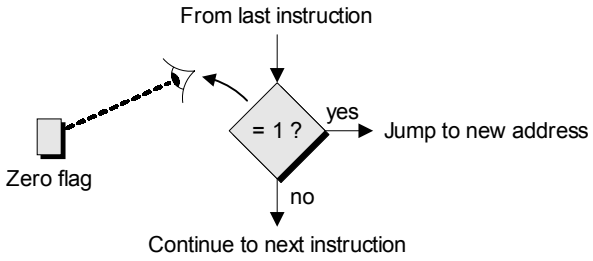
Flags affected

None

JZ (Jump if zero)

Description

This instruction is used to change the "flow" of the program by causing the CPU to jump to a new address if the zero status flag is `TRUE` (contains a logic 1, thereby indicating that the result from the previous instruction was zero), otherwise the CPU ignores the operand and continues to the next instruction. See also the corresponding `JNZ` instruction.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
abs	3	$D1	`JZ [$4C76]`	If the zero flag contains a logic 1 then jump to address $4C76; otherwise continue to the next instruction

Flags affected

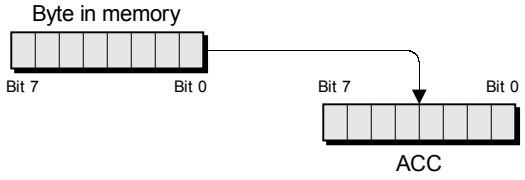
None

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

LDA (Load the accumulator)

Description

This instruction loads the contents of a byte of data in memory into the accumulator (the contents of the memory are not affected). See also the corresponding `STA` instruction.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imm	2	$90	`LDA $03`	Load the ACC with $03.
abs	3	$91	`LDA [$4C76]`	Load the ACC with the contents of address $4C76
abs-x	3	$92	`LDA [$4C76,X]`	Add $4C76 to the contents of the X register to form the target address $xxxx, then load the ACC with the contents of the target address
ind	3	$93	`LDA [[$4C76]]`	Read the target address $xxxx stored in the two bytes starting at address $4C76, then load the ACC with the contents of the target address
x-ind	3	$94	`LDA [[$4C76,X]]`	Add $4C76 to the contents of the X register to form a new address $zzzz. Read the target address $xxxx stored in the two bytes starting at address $zzzz, then load the ACC with the contents of the target address
ind-x	3	$95	`LDA [[$4C76],X]`	Read the address $zzzz stored in the two bytes starting at address $4C76, then add $zzzz to the contents of the X register to form the target address $xxxx, then load the ACC with the contents of the target address

Flags affected

N Set to 1 if the MS bit of the ACC is 1; otherwise cleared to 0
Z Set to 1 of all of the bits in the ACC are 0; otherwise cleared to 0

NOP (No operation)

Description

This instruction is a little strange in that it doesn't do anything at all, which may prompt the question: "*Why bother having it in the first place*?" The point is that executing a `NOP` does take a finite amount of time, which makes it useful for creating delay loops in a program. Also, it's sometimes useful to use a `NOP` instruction as the target for a breakpoint. See also the `HALT` instruction, which causes the CPU to generate internal `NOP`s.

Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imp	1	$00	`NOP`	Doesn't actually do anything

Flags affected

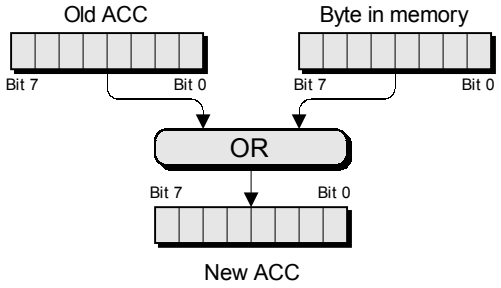
None

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

OR (Logical operation)

Description

This instruction logically ORs the contents of a byte of data in memory with the current contents of the accumulator and stores the result in the accumulator (the contents of the memory are not affected). Note that this is a bit-wise operation, which means that bit 0 of the old ACC is OR-ed with bit 0 of the memory to generate bit 0 of the new ACC. Similarly, bit 1 is OR-ed with bit 1, bit 2 with bit 2, and so forth. See also the `AND` and `XOR` instructions.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imm	2	$38	`OR $03`	Logically OR $03 with the ACC
abs	3	$39	`OR [$4C76]`	Logically OR the contents of memory location $4C76 with the ACC
abs-x	3	$3A	`OR [$4C76,X]`	Logically OR the contents of a memory location with the ACC, where the address of the memory location is $4C76 plus the contents of the X register

Flags affected

N Set to 1 if the MS bit of the result is 1; otherwise cleared to 0

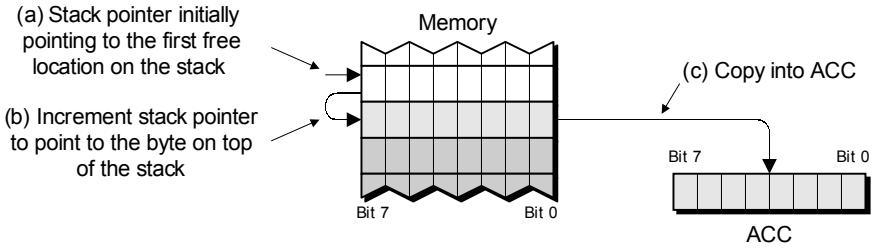
Z Set to 1 of all of the bits in the result are 0; otherwise cleared to 0

Registers			**Flags**			**Addressing Modes**			**Other**		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

POPA (Pop the accumulator off the top of the stack)

Description

This instruction first increments the stack pointer such that it points to the last byte placed onto the stack, then it copies this byte into the accumulator. See also the corresponding `PUSHA` instruction and the related `POPSR` and `PUSHSR` instructions.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imp	1	$B0	`POPA`	Pops the byte on top of the stack into the ACC

Flags affected

N Set to 1 if the MS bit of the ACC is 1; otherwise cleared to 0

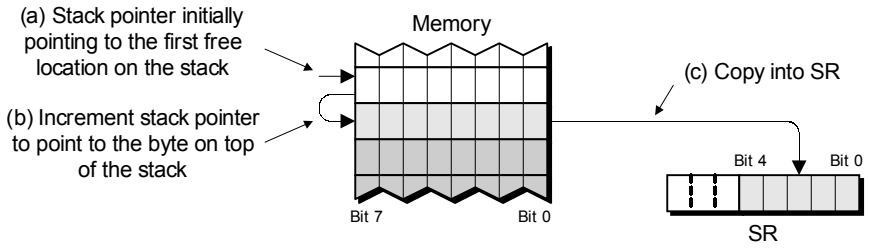
Z Set to 1 of all of the bits in the ACC are 0; otherwise cleared to 0

Registers			**Flags**			**Addressing Modes**			**Other**		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

POPSR (Pop the status register off the top of the stack)

Description

This instruction first increments the stack pointer such that it points to the last byte placed onto the stack, then it copies this byte into the status register. Note that, as the status register is only five bits wide, the three most-significant bits from the byte on top of the stack are discarded. See also the corresponding `PUSHSR` instruction and the related `POPA` and `PUSHA` instructions.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imp	1	$B1	`POPSR`	Pops the byte on top of the stack into the SR

Flags affected

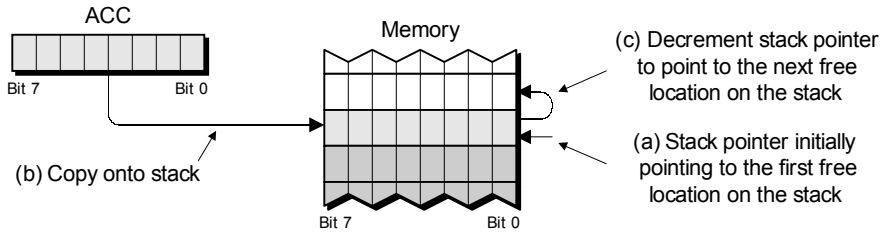
I	Loaded with whatever was in bit 4 of the byte on top of the stack
O	Loaded with whatever was in bit 3 of the byte on top of the stack
N	Loaded with whatever was in bit 2 of the byte on top of the stack
Z	Loaded with whatever was in bit 1 of the byte on top of the stack
C	Loaded with whatever was in bit 0 of the byte on top of the stack

Registers			**Flags**			**Addressing Modes**			**Other**		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

PUSHA (Push the accumulator onto the top of the stack)

Description

This instruction first copies the contents of the accumulator onto the top of the stack; it then decrements the stack pointer such that it points to the next free location (the contents of the accumulator are not affected). See also the corresponding `POPA` instruction and the related `POPSR` and `PUSHSR` instructions.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imp	1	$B2	`PUSHA`	Pushes the ACC onto the stack

Flags affected

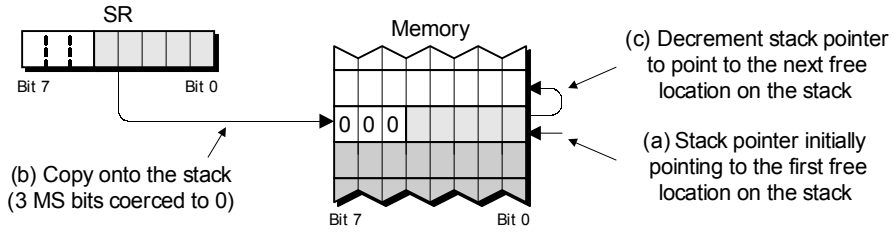
None

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

PUSHSR (Push the status register onto the stack)

Description

This instruction first copies the contents of the status register onto the top of the stack; it then decrements the stack pointer such that it points to the next free location (the contents of the status register are not affected). Note that, as the status register is only five bits wide, the three most-significant bits in the byte on top of the stack are coerced to logic 0*s*. See also the corresponding `POPSR` instruction and the related `POPA` and `PUSHA` instructions.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imp	1	$B3	`PUSHSR`	Pushes the SR onto the stack

Flags affected

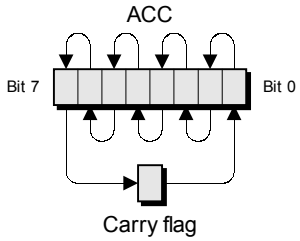
None

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

ROLC (Rotate accumulator left through the carry flag)

Description

This instruction rotates the contents of the accumulator 1 bit left and through the carry status flag. The original contents of the carry flag are loaded into bit 0 of the ACC, while the original contents of bit 7 of the ACC are loaded into the carry flag. See also the corresponding `RORC` instruction and the related `SHL` and `SHR` instructions.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imp	1	$78	`ROLC`	Rotates the ACC 1 bit left and through the carry flag

Flags affected

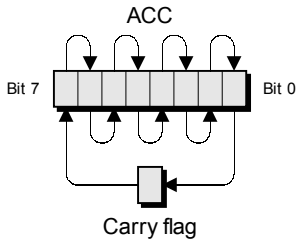
- N Set to 1 if the MS bit of the ACC is 1; otherwise cleared to 0
- Z Set to 1 of all of the bits in the ACC are 0; otherwise cleared to 0
- C Loaded with whatever was in bit 7 of the ACC

Registers			**Flags**			**Addressing Modes**			**Other**		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

RORC (Rotate accumulator right through the carry flag)

Description

This instruction rotates the contents of the accumulator 1 bit right and through the carry status flag. The original contents of the carry flag are loaded into bit 7 of the ACC, while the original contents of bit 0 of the ACC are loaded into the carry flag. See also the corresponding `ROLC` instruction and the related `SHL` and `SHR` instructions.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imp	1	$79	`RORC`	Rotates the ACC 1 bit right and through the carry flag

Flags affected

N Set to 1 if the MS bit of the ACC is 1; otherwise cleared to 0
Z Set to 1 of all of the bits in the ACC are 0; otherwise cleared to 0
C Loaded with whatever was in bit 0 of the ACC

Registers			**Flags**			**Addressing Modes**			**Other**		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

RTI (Return from an interrupt)

Description

This instruction is used to terminate an interrupt service routine and return control back to the main program. Remember that when an interrupt occurs, the CPU completes the instruction it's currently working on and then checks the state of the interrupt mask flag. If this flag is active (logic 1), the CPU automatically pushes a return address onto the stack followed by the current contents of the status register; it then jumps to the interrupt service routine located at the address contained in the interrupt vector (the CPU also clears the interrupt latch at this time).

By comparison, when the CPU sees an `RTI` instruction, it automatically pops the top-most byte off the stack into the status register (much like a `POPSR` instruction). The CPU then retrieves the 2-byte address from the top of the stack and uses this address as the entry point for its return to the main program. Note that an interrupt service routine can contain a number of `RTI` instructions. See also the somewhat related `RTS` instruction and the discussions on interrupts in *Chapter 3*.

Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imp	1	$C7	`RTI`	Exits the interrupt service routine and returns control to the main program

Flags affected

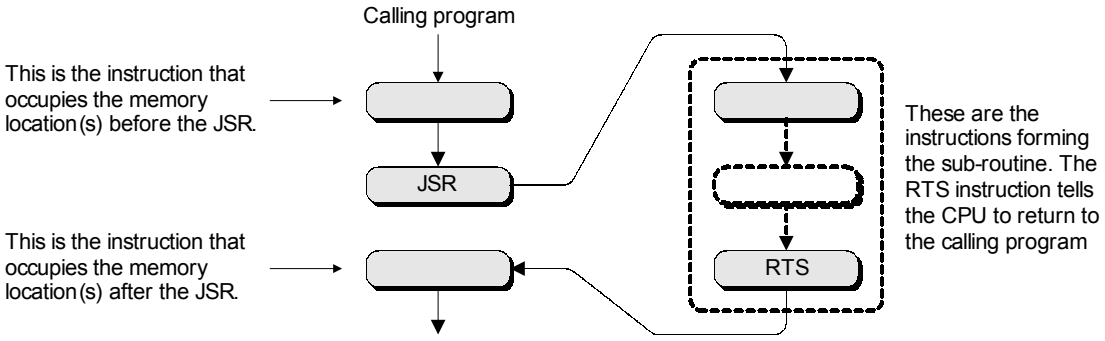
All The SR is reloaded with whatever its contents were when the interrupt caused it to be pushed onto the stack (assuming the programmer hasn't used the interrupt service routine to modify the copy of the SR on the stack)

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

RTS (Return from a subroutine)

Description

This instruction is used to terminate a subroutine and return control to the calling program. The CPU automatically retrieves a 2-byte address from the top of the stack and uses this address as the entry point for its return to the calling program. Note that a subroutine can contain a number of `RTS` instructions. See also the related `JSR` instruction and the somewhat related `RTI` instruction.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imp	1	$CF	`RTS`	Exits the subroutine and returns control to the calling program

Flags affected

None

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

SETIM (Set the interrupt mask)

Description

This instruction sets the interrupt mask bit in the status register to logic 1, thereby allowing the CPU to see any future interrupts. See also the `CLRIM` instruction in this appendix and the discussions on interrupts in *Chapter 3*.

Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imp	1	$08	`SETIM`	Load the interrupt mask with 1

Flags affected

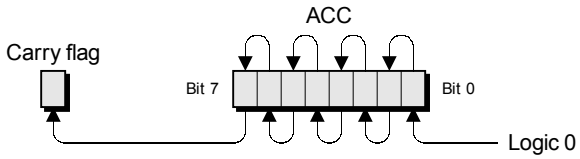
I Loaded with logic 1

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

SHL (Shift accumulator left)

Description

This instruction shifts the contents of the accumulator 1 bit to the left. A logic 0 is shifted in to the least-significant bit of the ACC, while the bit that "falls off the end" (bit 7 in this case) is stored in the carry status flag. See also the corresponding `SHR` instruction and the related `ROLC` and `RORC` instructions.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imp	1	$70	`SHL`	Shifts the accumulator 1 bit left

Flags affected

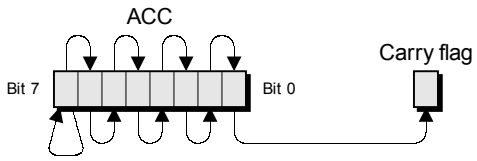
N Set to 1 if the MS bit of the ACC is 1; otherwise cleared to 0
Z Set to 1 of all of the bits in the ACC are 0; otherwise cleared to 0
C Loaded with whatever was in bit 7 of the ACC

Registers			**Flags**			**Addressing Modes**			**Other**		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

SHR (Shift accumulator right)

Description

This instruction shifts the contents of the accumulator 1 bit to the right. This is an *arithmetic shift right* (as opposed to a *logical shift right*, for which the CPU doesn't have an instruction), which means that the most-significant bit of the ACC is copied back into itself, while the bit that "falls off the end" (bit 0 in this case) is stored in the carry status flag. See also the corresponding `SHL` instruction and the related `ROLC` and `RORC` instructions.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imp	1	$71	`SHR`	Shifts the accumulator 1 bit right (using an arithmetic shift technique)

Flags affected

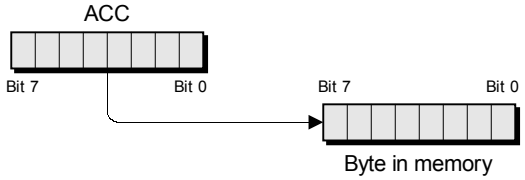
- N Set to 1 if the MS bit of the ACC is 1; otherwise cleared to 0
- Z Set to 1 of all of the bits in the ACC are 0; otherwise cleared to 0
- C Loaded with whatever was in bit 0 of the ACC

Registers			Flags			Addressing Modes			Other		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

STA (Store the accumulator)

Description

This instruction stores the contents of the accumulator to a byte in the memory (the contents of the accumulator are not affected). See also the corresponding `LDA` instruction.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
abs	3	$99	`STA [$4C76]`	Copy the contents of the ACC into address $4C76
abs-x	3	$9A	`STA [$4C76,X]`	Add $4C76 to the contents of the X register to form the target address $xxxx, then copy the contents of the ACC into the target address
ind	3	$9B	`STA [[$4C76]]`	Read the target address $xxxx stored in the two bytes starting at address $4C76, then copy the contents of the ACC into the target address
x-ind	3	$9C	`STA [[$4C76,X]]`	Add $4C76 to the contents of the X register to form a new address $zzzz. Read the target address $xxxx stored in the two bytes starting at address $zzzz, then copy the contents of the ACC into the target address
ind-x	3	$9D	`STA [[$4C76],X]`	Read the address $zzzz stored in the two bytes starting at address $4C76, then add $zzzz to the contents of the X register to form the target address $xxxx, then copy the contents of the ACC into the target address

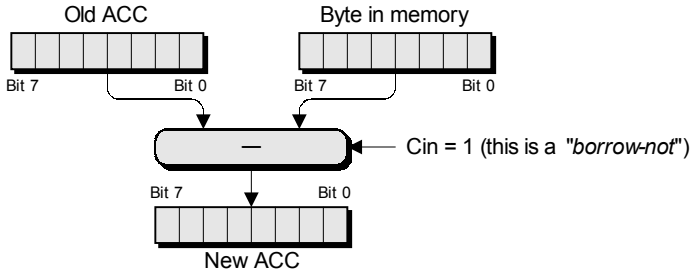
Flags affected

None

SUB (Subtract without carry)

Description

This instruction subtracts the contents of a byte of data in memory from the current contents of the accumulator and stores the result in the accumulator (the contents of the memory are not affected). Note that the result is not affected by the contents of the carry flag, because the carry-in to the ALU is forced to logic 1 (the carry-in is really a "*borrow-not*" in this case). See also the corresponding `SUBC` instruction.



Note: The diagram above is a stylized representation of the action of the `SUB` instruction. In reality, the CPU doesn't have a "subtractor block," so the actual operation that is performed to achieve the desired result is as follows (where the +1 on the far right-hand side of the equation comes from the `Cin` ("carry-in") signal):

```
new_ACC[7:0] = old_ACC[7:0] + NOT(byte_in_memory[7:0]) + 1
```

Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imm	2	$20	`SUB $03`	Subtract $03 from the ACC.
abs	3	$21	`SUB [$4C76]`	Subtract the contents of memory location $4C76 from the ACC
abs-x	3	$22	`SUB [$4C76,X]`	Subtract the contents of a memory location from the ACC, where the address of the memory location is $4C76 plus the contents of the X register

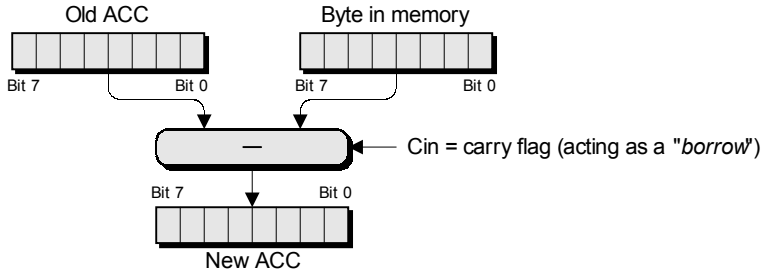
Flags affected

- O Set to 1 if the result overflows; otherwise cleared to 0
- N Set to 1 if the MS bit of the result is 1; otherwise cleared to 0
- Z Set to 1 of all of the bits in the result are 0; otherwise cleared to 0
- C Set to 1 if there is a carry out (really a "*borrow-not*") from the subtraction; otherwise cleared to 0 (which indicates a "*borrow*")

SUBC (Subtract with carry)

Description

This instruction subtracts the contents of a byte of data in memory (along with the current contents of the carry flag) from the current contents of the accumulator and stores the result in the accumulator (the contents of the memory are not affected). See also the corresponding `SUB` instruction.



Note: The diagram above is a stylized representation of the action of the `SUBC` instruction. In reality, the CPU doesn't have a "subtractor block," so the actual operation that is performed to achieve the desired result is as follows:

```
new_ACC[7:0] = old_ACC[7:0] + NOT(byte_in_memory[7:0]) + Cin
```

Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imm	2	$28	`SUBC $03`	Subtract $03 from the ACC
abs	3	$29	`SUBC [$4C76]`	Subtract the contents of memory location $4C76 from the ACC
abs-x	3	$2A	`SUBC [$4C76,X]`	Subtract the contents of a memory location from the ACC, where the address of the memory location is $4C76 plus the contents of the X register

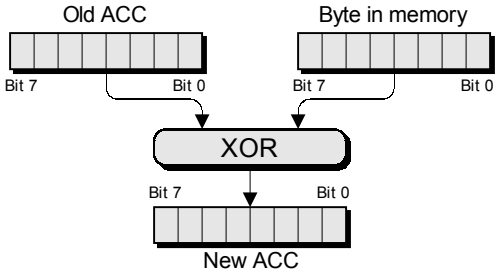
Flags affected

- O Set to 1 if the result overflows; otherwise cleared to 0
- N Set to 1 if the MS bit of the result is 1; otherwise cleared to 0
- Z Set to 1 of all of the bits in the result are 0; otherwise cleared to 0
- C Set to 1 if there is a carry out (really a "*borrow-not*") from the subtraction; otherwise cleared to 0 (which indicates a "*borrow*")

XOR (Logical operation)

Description

This instruction logically XORs the contents of a byte of data in memory with the current contents of the accumulator and stores the result in the accumulator (the contents of the memory are not affected). Note that this is a bit-wise operation, which means that bit 0 of the old ACC is XOR-ed with bit 0 of the memory to generate bit 0 of the new ACC. Similarly, bit 1 is XOR-ed with bit 1, bit 2 with bit 2, and so forth. See also the `AND` and `OR` instructions.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imm	2	$40	`XOR $03`	Logically XOR $03 with the ACC
abs	3	$41	`XOR [$4C76]`	Logically XOR the contents of memory location $4C76 with the ACC
abs-x	3	$42	`XOR [$4C76,X]`	Logically XOR the contents of a memory location with the ACC, where the address of the memory location is $4C76 plus the contents of the X register

Flags affected

N Set to 1 if the MS bit of the result is 1; otherwise cleared to 0

Z Set to 1 of all of the bits in the result are 0; otherwise cleared to 0

Registers			**Flags**			**Addressing Modes**			**Other**		
ACC	=	Accumulator	Z	=	Zero	imp	=	Implied	LS	=	Least-significant
PC	=	Program Counter	N	=	Negative	imm	=	Immediate	MS	=	Most-significant
IR	=	Instruction Register	C	=	Carry	abs	=	Absolute	Addr	=	Address
X	=	Index Register	O	=	Overflow	abs-x	=	Indexed			
SP	=	Stack Pointer	I	=	Interrupt Mask	ind	=	Indirect			
IV	=	Interrupt Vector				x-ind	=	Pre-indexed indirect			
						ind-x	=	Indirect post-indexed			

[THIS PAGE IS INTENTIONALLY LEFT BLANK FOR PRINTING PAGINATION]

Appendix B

Chip Packaging and Pin Descriptions

Chip packaging and pin descriptions

The *DIY Calculator's* CPU is usually supplied in a plastic quad flat-pack package with an 0.1-inch pin pitch intended for commercial applications (0^{O} C through 70^{O} C), but other options are available — contact your nearest supplier for more details (Figure B-1).

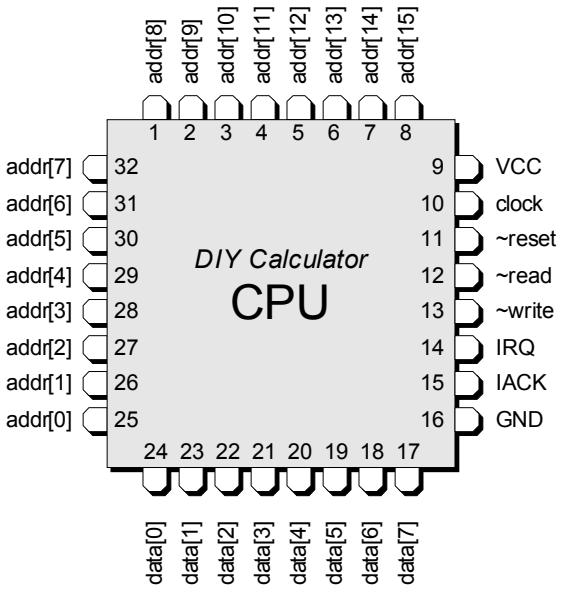


Figure B-1. The CPU in a quad flat-pack package ("birds-eye" view)

VCC and GND (power supply pins)

The CPU is usually powered with `GND` = 0V and `VCC` = +5V (other voltage options are available – contact your nearest supplier for more details).

clock (input)

The *DIY Calculator* employs a single phase clock with a 1:1 mark-space ratio. The frequency of the clock can range from 0 Hz to 10 MHz. (higher frequency variants are available – contact your nearest supplier for more details).

See also the timing diagrams in *Appendix C for* more details as to the relationships between the `clock` and other signals.

~reset (input, active-low)

When the `~reset` input is placed in its active state (logic 0 = 0V), it causes the CPU to load the program counter (PC) with address $0000. A reset condition also loads the status register (SR) with $00, thereby disabling the interrupt mask flag. The reset will also force the `IACK`, `~read`, and `~write` outputs into their inactive (logic 1) states. Once a reset sequence begins, the `~reset` input must remain in its active state active for a minimum of 10 clock cycles. Similarly, when power is first applied to the CPU, an external power-on reset circuit must force the `~reset` input into its active state for a minimum of 10 clock cycles.

Once the ~reset input returns to its inactive state, the CPU will automatically read its first instruction from address $0000. (In the case of the *DIY Calculator*, address $0000 is the first memory location in the ROM, and we may assume that this instruction is an unconditional jump to address $4000, which is the first location in the RAM.)

Note: Resetting the CPU (including a power-on reset) leaves the stack pointer (SP), index register (X), and interrupt vector (IV) in undefined states.

See also the timing diagrams in *Appendix C* for more details as to the relationships between the ~reset input and the other signals.

~read and ~write (outputs, active-low)

The CPU places its ~read or ~write outputs in their active (logic 0) states to inform other devices in the system when it wishes to read or write data, respectively. Resetting the CPU (or a power-on reset) will force these outputs into their inactive (logic 1) states.

See also the timing diagrams in *Appendix C* for more details as to the relationships between the ~read and ~write outputs and the other signals.

IRQ (input, active-low)

The interrupt request (IRQ) input is active low (logic 0). A low-going pulse on the IRQ will be automatically stored in the CPU's interrupt latch. This means that the IRQ does not have to be synchronized to the clock or remain active for any particular number of clock cycles. The IRQ does, however, have a minimum pulse width requirement of 25 nanoseconds.

As was noted earlier, a reset or a power-on reset will load the status register with $00, thereby disabling the interrupt mask flag. This means that the CPU will not respond to any activity on the IRQ input following a reset. In order for the CPU to see an event on the IRQ input, the programmer must use a SETIM instruction to set the interrupt mask in the status register to a logic 1. (Note that the SETIM instruction also automatically resets the interrupt latch.)

Once the interrupt mask has been enabled, a low-going pulse on the IRQ input will cause an interrupt to occur (see also *Chapter 3*). At this point, the CPU will push the current contents of the program counter (PC) onto the stack, followed by the current contents of the status register (SR). The CPU also loads the interrupt mask with a logic 0 to prevent future interrupts from having any effect. The CPU then copies the contents of the interrupt vector (IV) into the program counter, which now points to the first instruction in the interrupt service routine.

When the interrupt service routine is terminated by an RTI (*"return from interrupt"*) instruction, the CPU pops the original value of the status register back off the stack, followed by the original contents of the program counter. The act of popping the status register off the stack will return the interrupt mask to a logic 1 (which was its value when the status register was originally pushed onto the stack), thereby re-enabling the CPU's ability to see any future interrupts. (Note that this will automatically reset the interrupt latch.) Also, the other status flags will be returned to whatever states they were in at the point when the interrupt was first activated.

See also the timing diagrams in *Appendix C* for more details as to the relationships between the IRQ input and the other signals.

IACK (output, active low)

As soon as the CPU starts to service an interrupt, it places its interrupt acknowledge (`IACK`) output into its active state (logic 0). The `IACK` output will remain in this state until the CPU receives an `RTI` ("*return from interrupt*") instruction or a reset occurs.

See also the timing diagrams in *Appendix C* for more details as to the relationships between the `IACK` output and the other signals.

data[7:0] (bidirectional inputs/outputs)

The 8-bit data bus is bidirectional, and these pins may be used to load instructions and data into the CPU or write values from the CPU.

See also the timing diagrams in *Appendix C* for more details as to the relationships between the `data[7:0]` signals and the other signals.

addr[15:0] (outputs)

The CPU uses its 16-bit address bus to “point” to locations in the memory or to the various input and output ports (which the CPU “sees” as being locations in memory).

See also the timing diagrams in *Appendix C* for more details as to the relationships between the `addr[15:0]` signals and the other signals.

Appendix C

Signal Descriptions and Timing Diagrams

The CPU's single-phase clock

The *DIY Calculator's* CPU is driven by a single-phase clock, which is used to synchronize all of its internal and external actions (Figure C-1).

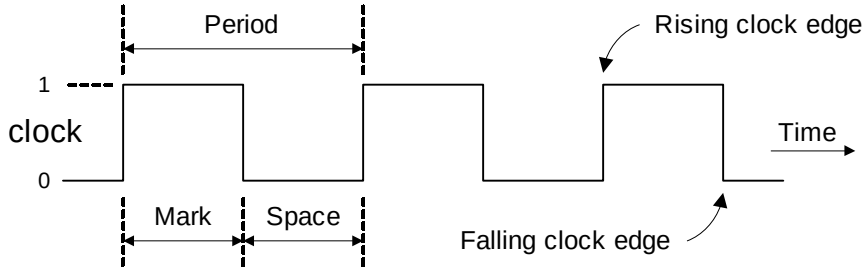


Figure C-1. The CPU's single-phase clock

A clock's *period* is the amount of time it takes to complete a full cycle (from 0 to 1 and back to 0 again). The *frequency* of a clock is the number of cycles it goes through per second, where frequency is measured in *Hertz* (or *Hz* for short), so 25 Hz means *"twenty-five cycles per second."* The frequency is the reciprocal of the period, and vice versa, so for a frequency of 25 Hz the period will be 1/25 seconds (that is 0.04 seconds). A clock cycle is split into two portions called the *mark* (which refers to the portion of the cycle when the clock is a logic 1) and the *space* (the portion when the clock is a logic 0).

The *DIY Calculator's* clock has a 1:1 mark-space ratio, which means that the mark and the space are of equal duration and each occupy 50% of the cycle. The frequency of the clock can range from 0 Hz to 10 MHz (ten million cycles-per-second).[(1)] This doesn't imply that the frequency is supposed to wander around, but rather that you can drive the clock at any frequency between these two limits.

Note: A clock frequency of 0 Hz may seem a little meaningless, because if the clock isn't "clocking" the CPU isn't doing anything, but specifying the clock in this manner indicates that the CPU won't forget any internal information if the clock is halted or paused for any reason. By comparison, certain microprocessors do require some minimum clock frequency to be maintained.

Note: The single-phase clock used by the *DIY Calculator* is the simplest of all schemes; many microprocessors use two or more clocks.

A generic read cycle

First, let's consider a generic read cycle (Figure C-2). Before we start this cycle (just prior to the rising clock edge we designate as #1), the address bus reflects some address associated with the previous operation. Similarly, the data bus is in its high-impedance state, which means that neither the CPU nor any other devices are currently driving any values onto this bus.

[1]Higher frequency variants are available -- contact your nearest supplier for more details.

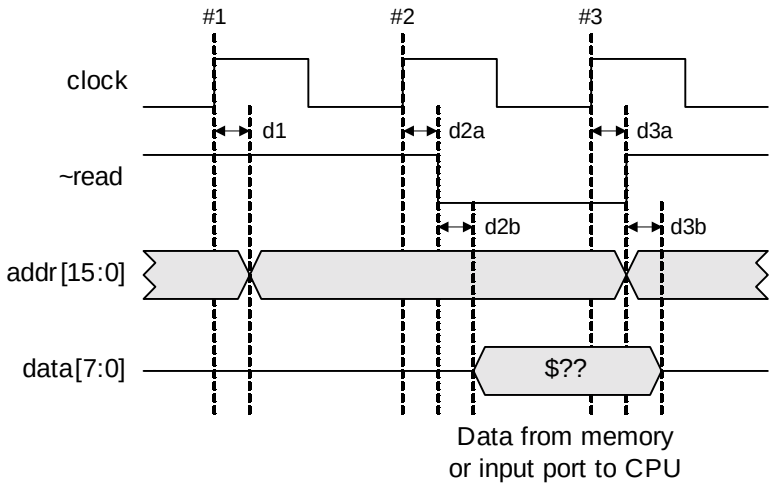


Figure C-2. Generic read cycle

The CPU uses the first rising edge on the clock (#1) to load its address latch with the contents of one of the addressing logic registers (such as the program counter or stack pointer). After a short delay (d1) caused by the internal logic gates and registers, this value appears on the address bus coming out of the CPU.

The next rising edge on the clock (#2) causes the CPU to place the `~read` control signal into its active state (logic 0).[(2)] Once again, there is a short delay (d2a) between the rising edge of the clock and the response on the `~read` signal due to internal gate delays. Similarly, there is another small delay (d2b) between the `~read` signal going active and the selected memory location or input port responding by placing some data onto the data bus (this data is shown as $?? to represent two hexadecimal characters whose values are of no concern at the moment).

The last clock in the read cycle (#3) loads the current value on the data bus into the appropriate register inside the CPU, and also causes the `~read` signal to return to its inactive state (logic 1) after a small delay (d3a).[(2)] The act of the `~read` signal going inactive causes the selected memory location or input port to cease driving a value onto the data bus, which therefore returns to a high-impedance value after another small delay (d3b). Using the same clock edge to load the data into the CPU *and* disable the device driving the data bus isn't a problem, because delays in the logic gates forming the system mean that the data is safely stored away before the data bus returns to its high-impedance state.

In addition to loading the data into the CPU and deactivating the `~read` signal, clock #3 also loads a new value into the CPU's address latch and this new value subsequently appears on the address bus. This new address is the one that will be used by the following read or write cycle.

A generic write cycle

Now let's consider a generic write cycle (Figure C-3). At the beginning of this cycle (just prior to the rising clock edge we designate as #1), the address bus reflects some address associated with the previous operation and the data bus is in its high-impedance state.

[2]See also the discussions on the CPU's data bus later in this appendix.

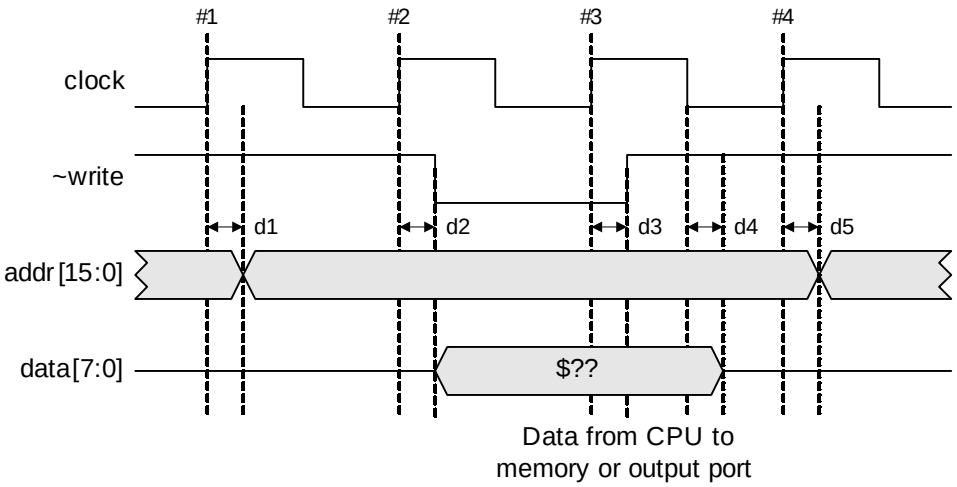


Figure C-3. Generic write cycle

As before, the CPU uses the first rising edge on the clock (#1) to load its address latch with the contents of one of the addressing logic registers. After a short delay (d1) caused by the internal logic gates and registers, this value appears on the address bus coming out of the CPU.

Note: If this generic write cycle were to follow the generic read cycle we discussed in the previous section, then clock #3 from the read cycle and clock #1 from the write cycle would be one and the same thing. Similarly, if our generic read cycle were to follow this generic write cycle, then clock #4 from the write cycle and clock #1 from the read cycle would actually be the same clock.

The next rising edge on the clock (#2) causes the CPU to place the `~write` control signal into its active state (logic 0). At the same time, the CPU also starts to drive a data value out onto the data bus(3) (once again, this data is shown as $?? to represent two hexadecimal characters whose values are of no concern at the moment). As usual, there is a short delay (d2) between the rising edge of the clock and the response on the `~write` signal and the data bus due to internal gate delays. (The `~write` signal and the data bus are shown as having identical delays for simplicity, but these delays are not directly related to each other in the real world.)

The period during which the `~write` signal is in its active state gives the destination memory location or output port the time to either start loading the data or to prepare to load the data. The next rising edge on the clock (#3) causes the CPU to return the `~write` signal to its inactive state, and it is this rising transition on the `~write` signal that ultimately loads the data into the destination location.

As an aside, consider a 3-bit binary field transitioning from a value of 001 to 100. In an ideal world, all of the bits would transition simultaneously, but in the real world the delay associated with a 0 to 1 transition will be different to its 1 to 0 counterpart. Thus, our hypothetical 3-bit field will actually exhibit a 001 > 000 > 100 sequence or a 001 > 101 > 100 sequence depending on which transition is the faster.

[3] See also the discussions on the CPU's data bus later in this appendix.

Similarly, if the value on the address bus were to change whilst the `~write` signal is in its active state, then the address bus could pass through a large number of intermediate addresses, each of which could become corrupted. Also, if the value on the data bus were to change while the `~write` signal was active or at the same time as `~write` returned to its inactive state, then a bad value could be written into the target location. This explains why the CPU continues to maintain the values on the address and data busses for some time after the `~write` signal has gone inactive. Thus, it's not until the *falling edge* of clock #3 that the CPU returns the data bus to its high-impedance state,(4) and the address latch isn't loaded with the address for the next operation until clock #4.

The CPU's data buffer

Way back in those mists of time that we fondly referred to as Chapter 2, we briefly introduced the CPU's data buffer, which is used to connect the CPU's internal data bus to the main system's data bus (Figure C-4).

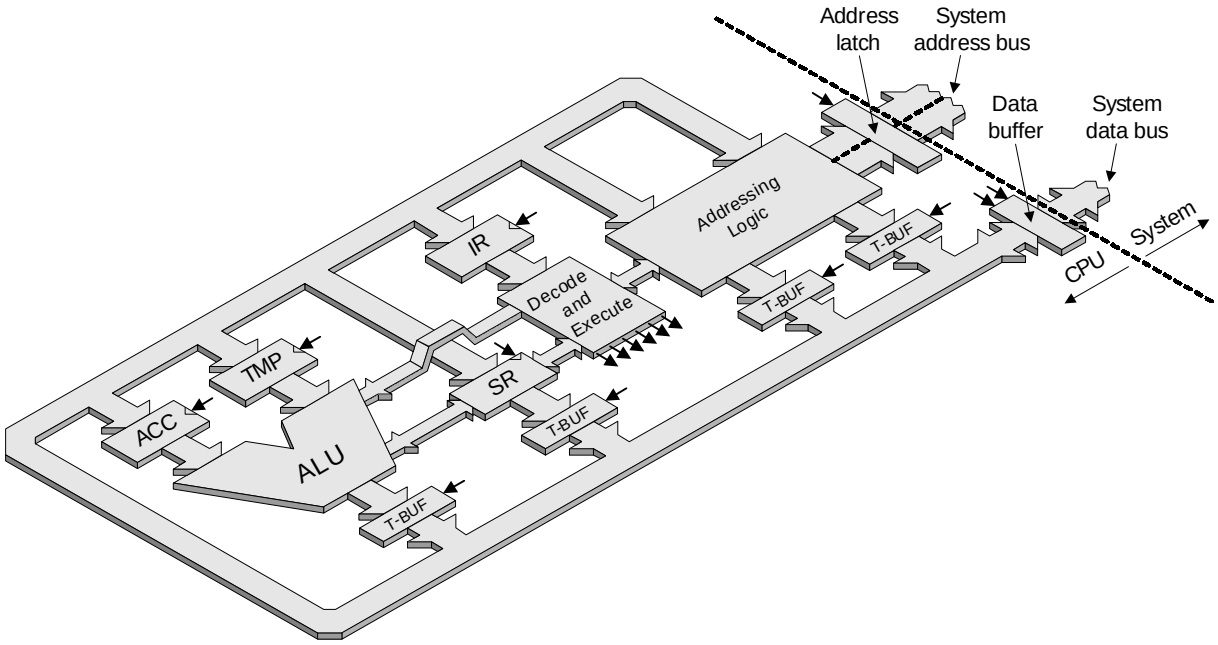


Figure C-4. The CPU's data buffer

Control signals from the CPU's instruction decoder and executer logic dictate whether this bi-directional data buffer will (a) allow data from the outside world to pass into the CPU, (b) allow data from the CPU to pass to the outside world, or (c) completely disconnect the internal data bus from the outside world (Figure C-5).

As this illustration shows, the data buffer basically consists of two tri-state buffer gates for each bit in the data bus. If both sets of tri-state buffers are disabled, the CPU's internal data bus is completely isolated from the outside world. Enabling one set of buffers allows the CPU to accept data *from* the main system's data bus, while enabling the other set of buffers allows the CPU to drive data *onto* the main system's data bus.

[4]See the discussions on the address latch later in this appendix for more details on why the tri-state buffers driving the data bus inside the CPU are disabled on falling clock edges.

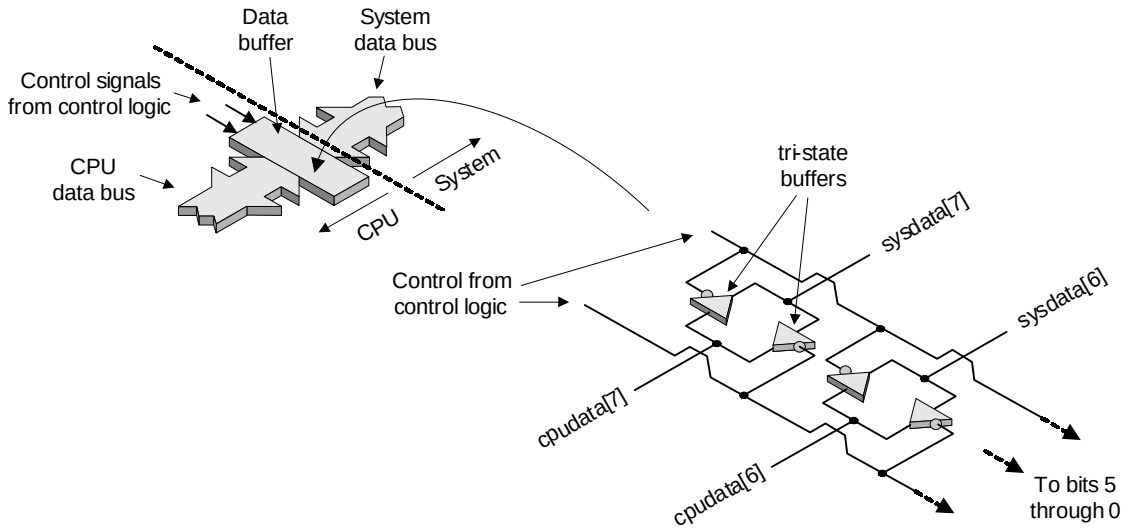


Figure C-5. A close-up look at the CPU's data buffer

For example, consider the case of the generic read cycle we discussed earlier. The clock edge that causes the `~read` signal to be driven to its active state also directs the data buffer to start accepting data from the outside world. And the clock edge that causes the `~read` signal to be driven to its inactive state also directs the data buffer to stop accepting data from the outside world.

Similarly, in the case of our generic write cycle, the clock edge that causes the `~write` signal to be driven to its active state also causes the data buffer to start driving data to the outside world. And the clock edge following the one that causes the `~write` signal to go inactive also causes the data buffer to stop driving data to the outside world.

One point you may be pondering is: *"Why do we bother having the data buffer at all?"* After all, as all of the devices driving the internal and external data busses can be individually enabled or disabled, we could get the same functionality using eight pieces of wire. Ah ha! This is a very good question that shows you're paying attention. As it happens, we *could* theoretically manage without the data buffer. In practice, however, we need this buffer to take weak signals from inside the CPU and drive them powerfully out into the main system. Similarly, devices in the outside world only have to drive the CPU's data buffer, which then handles the task of driving all of the blocks inside the CPU.

The CPU's address latch

The address latch is used to store current address of interest and to drive this value out onto the system's address bus. As we discussed in *Chapter 2*, this address may be supplied directly from one of the standard addressing registers – such as the *program counter (PC)* or *stack pointer (SP)* – or it may be generated by adding the contents of one of the standard addressing registers to the contents of the index register (Figure C-6).

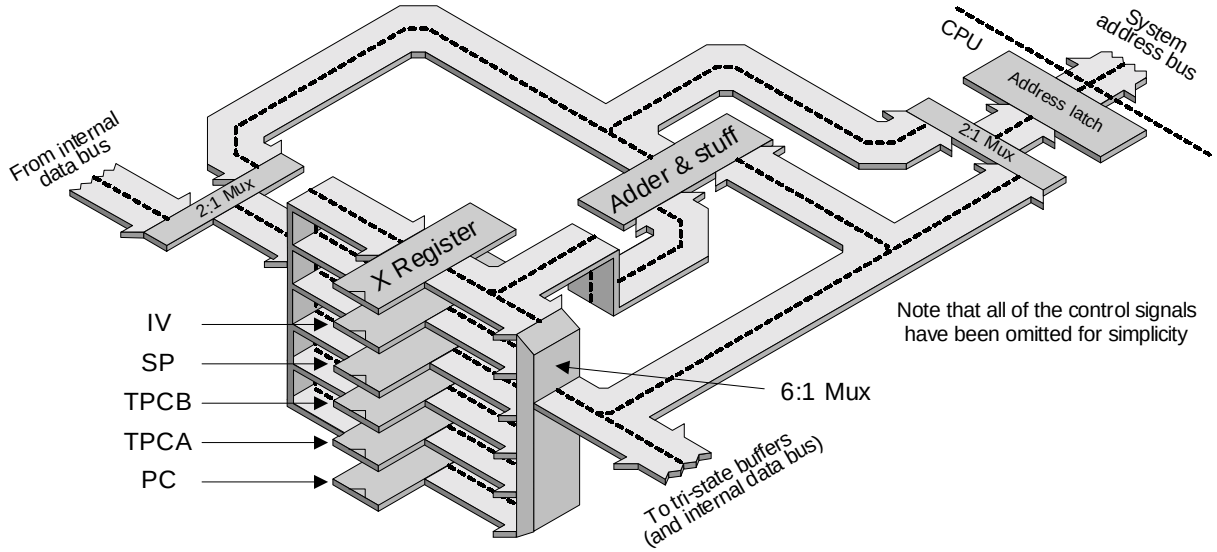


Figure C-6. The CPU's addressing logic

The adder block may also be used to increment (or decrement) the contents of the program counter and the other addressing registers. This leads to an interesting question, which is *"When should this incrementing (or decrementing) take place?"* For example, let's consider the program counter. One possibility would be to increment the program counter at the *same time* as we load its current contents into the address latch (Figure C-7).

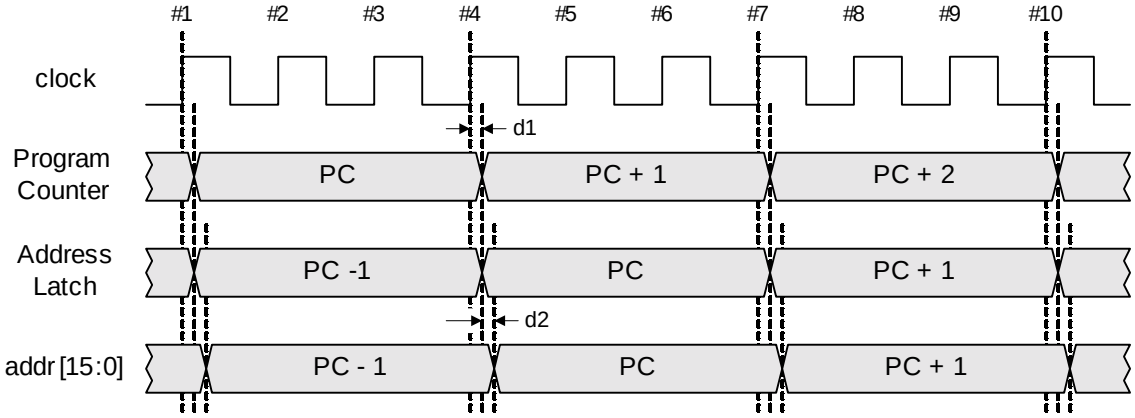


Figure C-7. Incrementing the program counter and loading the address latch at the same time

If you glance back to Figure C-6, you can see that at the same time as the value coming out of the program counter is being loaded into the address latch, we can also be incrementing it using the adder block and feeding it back into the program counter via the 2:1 multiplexers driving the addressing registers. The reason this works is because all of the logic blocks forming the loop (program counter > adder > program counter) have internal delays (albeit small ones), which means that the original value in the program counter can be safely stored in the address latch by the time the new (incremented) value emerges from the program counter.

Using this scheme, the value in the address latch would always be exactly one step behind the value in the program counter. For example, if the program counter contained some value that we might call PC, then the address latch would contain PC-1. In the case of Figure C-7, we're assuming that we load the address latch and increment the program counter on the #1, #4, #7, and #10 clock edges, and so forth. Due to delays in the elements forming the circuit, the outputs of the address latch and program counter would begin to respond after some delay (d1), and the address bus would follow the address latch after another delay (d2). (The address latch and the program counter are shown as having identical delays for simplicity, but in reality these delays are not directly related to each other.)

As an alternative scheme, the CPU's control logic could cause the program counter to be incremented on the clock edge *following* the one that loads the address latch (Figure C-8).

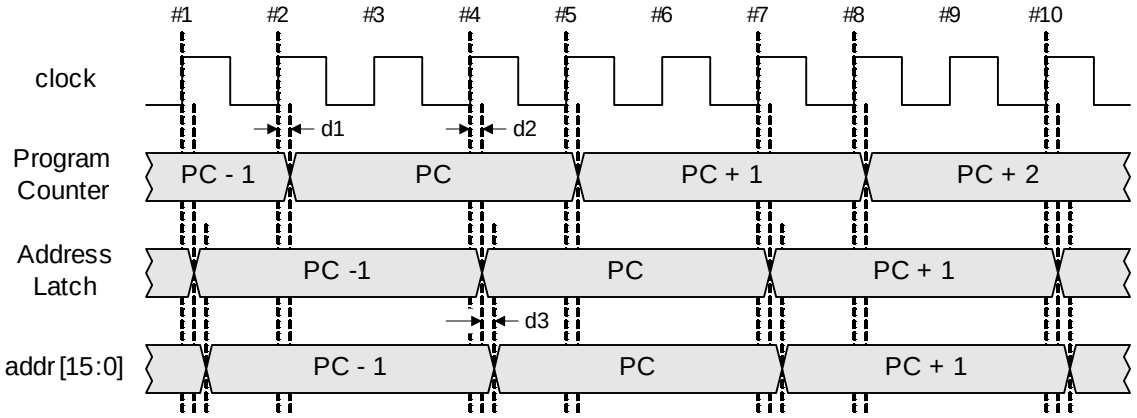


Figure C-8. Incrementing the program counter *after* loading the address latch

In an ideal world, it would be nice if we could decide to use just one of the above schemes and stick to it. In reality, however, we have to use both techniques depending on the type of operation being performed (see also the discussions on *Alternative schemes* below).

One very important point is that the control signals for the 2:1 multiplexer feeding the inputs to the addressing registers and the 6:1 multiplexer selecting between the outputs from the addressing registers are modified on falling edges of the system clock. For a variety of reasons that will become apparent if you decide to implement this CPU,[5] modifying these multiplexers on falling clock edges reduces the number of clock cycles required to execute certain instructions. As a related issue, the tri-state buffers connecting the ALU, status register, and addressing logic to the CPU's internal data bus are also disabled on falling clock edges (but they're enabled on rising clock edges).

Alternative schemes

It's important to note that the timing schemes used by the *DIY Calculator's* CPU are somewhat conservative. This is because these schemes are designed to be "safe" and relatively easy to understand and implement, but there are a variety of other techniques we might have used. For example, the discussions in this appendix are based on the fact that the majority of the CPU's actions are instigated by the rising edge of the clock. But we could modify the CPU such that

[5] Or if you decide to model it using a hardware description language (HDL) such as VHDL or Verilog.

rising clock edges only perform certain actions like loading the address latch, while actions such as loading the program counter could be performed on falling clock edges.

Alternatively, instead of having a single clock, we could modify the CPU to use two or more clocks (Figure C-9). In this case, we could use one clock to perform tasks like loading the address latch, while the other clock could perform tasks like loading the program counter.

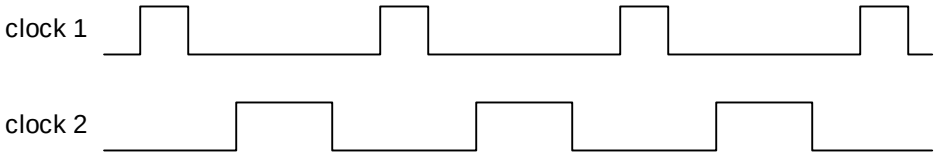


Figure C-9. One alternative CPU implementation would be to use two (or more) clocks

In the previous section, we noted that we sometimes increment the program counter using the same clock that loads the address latch, but on occasion we have to increment it on the following clock. An example of this latter case occurs during the execution of instructions which require data to be loaded from the CPU's internal data bus into one of the addressing registers. In order to understand this problem, cast your mind back to the architecture of our CPU's addressing logic (Figure C-10).

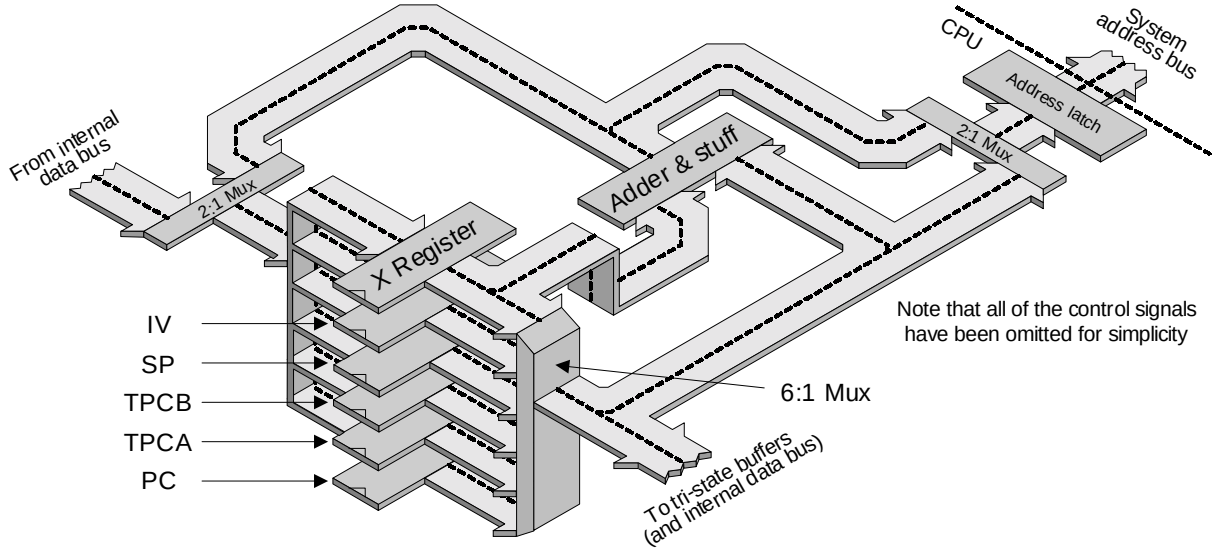


Figure C-10. The CPU's addressing logic

The CPU has only one set of 2:1 multiplexers selecting between the internal data bus and the output from the adder block. The outputs from these multiplexers feed all of the addressing registers, which therefore prevents us from loading two or more addressing registers with different values at the same time. In turn, this means that we have to postpone incrementing the program counter until the clock edge following the one that loads data into one of the other addressing registers.

If we wished, we could modify our architecture so as to provide each of the addressing registers with its own 2:1 multiplexer. This would allow one register to be loaded from the data bus while another register was being loaded from the adder block, but it would also require more logic gates and would increase the complexity of the CPU.

Similarly, Figure C-10 shows the 6:1 multiplexer that chooses between the six addressing registers and feeds the outputs from the selected register to both the address latch and the adder block. This is great if we wish to load the address latch from the same addressing register that we wish to increment, but it doesn't allow us to load the address latch with the contents of one register (say the stack pointer) whilst incrementing another register (say the program counter).

Once again, we could modify our architecture so as to provide one 6:1 multiplexer to feed the address latch and another to drive the adder block. This would allow us to simultaneously increment one addressing register and load the address latch with the contents of another register. But once again, this would require more logic gates and would increase the complexity of the CPU.

If fact, there are a whole slew of different possibilities. However, if you are attempting to model the *DIY Calcuator's* CPU in some hardware description language (HDL) such as VHDL or Verilog, then we strongly recommend that you start off by making your model conform to the specifications presented in this data book. Once you've got a working "base-level" model, you can start experimenting with alternative schemes and comparing them to your original model.

Resetting the CPU

As was discussed in *Chapter 2*, when power is first applied to the system, the CPU undergoes a process called *power-on reset*, which initializes it into a "known good" state. The CPU can also be re-initialized at any time by driving its `~reset` input to its active state (logic 0). In order to illustrate a reset sequence, let's first assume that the CPU is in the process of executing some program (Figure C-11).

Due to the fact that a reset can occur at any time, we can't be certain as to the values that will be present on the CPU's `~read`, `~write`, and `IACK` outputs when the `~reset` input is driven to its active state (logic 0). Thus, Figure C-11 initially employs question mark ('?') characters to indicate that these signals are carrying good logic 0 or logic 1 values, it's just that we don't happen to know what these values are when the reset commences.

Similarly, the "$????" and "$??" character strings that are initially associated with the data and address busses are used to indicate that we aren't sure of the current values on these busses. (The '$' characters mean that each '?' character represents a hexadecimal digit, where each of these digits equates to four binary bits. Thus, "$????" represents the 16 bits associated with the address bus, while $?? "represents" the 8 bits associated with the data bus.)

As soon as the `~reset` signal goes active, we lose all knowledge of the values on the signals coming out of the CPU, at which point Figure C-11 shows these signals as entering undefined states (represented by the 'U', "$UU", and "$UUUU" character strings as detailed in the "Legend" below). It's only after the `~reset` input has been maintained in its active state for 10 complete clock cycles that the signals coming out of the CPU can be guaranteed to be in a well-known

state.[6] Similarly, when power is first applied to the system (and once the clock signal is fully established), the power-on-reset circuit must hold the ~reset input in its active state for ≥10 full clock cycles. (If the ~reset input is active for less than 10 clock cycles, the subsequent actions of the CPU are undefined.)

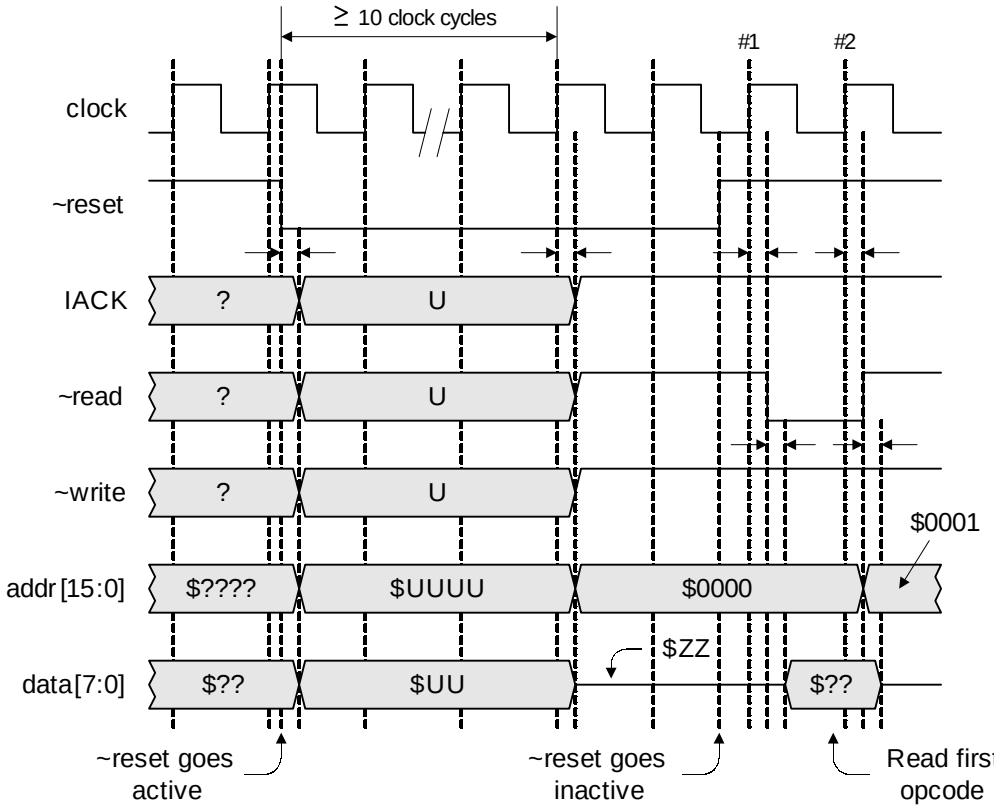


Figure C-11. Timing for reset-related actions

On the 10th clock cycle following the ~reset going active, we can guarantee that the CPU has been re-initialized into a known-good state. Amongst other things, this means that the ~read, ~write, and IACK signals will all be placed in their inactive (logic 1) states. Also, by this stage in the game we can guarantee that the data bus has been placed in its high-impedance $ZZ state.

On or before the 10th clock cycle, the address latch will be loaded with an address of $0000 and the program counter will be loaded with a value of $0001 (see also the discussions on the address latch earlier in this appendix).

Legend	
? =	Valid 0 or 1 on an individual signal
$?? =	Valid 0s and 1s on 8-bit data bus
$???? =	Valid 0s and 1s on 16-bit address bus
$ZZ =	High-impedance values on 8-bit data bus
U =	Undefined value on an individual signal
$UU =	Undefined values on 8-bit data bus
$UUUU =	Undefined values on 16-bit address bus

Nothing further will occur after the 10th clock cycle until the ~reset input is returned to its inactive (logic 1) state. Following this event, the next rising clock edge (indicated as #1 in Figure C-11) will drive the ~read signal to its active state, which will in turn start the

[6]The signals coming out of the CPU may become valid at any time during the reset operation, but they cannot be *guaranteed* to be valid until after the 10th clock cycle.

process of reading whatever opcode byte is to be found in address $0000 (the address currently stored in the address latch).

The next clock (#2) causes this opcode to be loaded into the CPU's *instruction register (IR).* This clock also returns the ~read signal to its inactive state and loads the current contents of the program counter (which are $0001) into the address latch, from whence this value will appear on the address bus. The clock after this (not shown in Figure C-11) will begin to execute this first opcode as discussed in the remainder of this appendix.

Hurray! {Fanfare of trumpets} The CPU is up and running! ... now read on ...

Implied mode instructions

Instructions using the *implied* addressing mode consist of only an opcode byte with no operand. These instructions can be categorized into six distinct groups based on the way in which they are executed:

	Implied Mode Instructions					
Group 1	HALT					
Group 2	INCA	DECA	INCX	DECX	SHL	SHR
	ROLC	RORC	NOP	SETIM	CLRIM	
Group 3	POPA	POPSR				
Group 4	PUSHA	PUSHSR				
Group 5	RTS					
Group 6	RTI					

Implied mode: Group 1

Just to ease our way into this, we'll commence with the HALT instruction, which causes the CPU to stop whatever it's currently doing and wait for an interrupt to occur (Figure C-12).

Note that the vast majority of our timing diagrams from here on assume the same starting conditions, which can be summarized as follows:

a) The address latch (and thus the address bus) is already loaded with some value from the previous instruction. We've called this value PC to indicate that it originated in the program counter.

b) The program counter itself has already been incremented such that it now contains a value of PC+1 (but this fact is not shown in the timing diagrams).

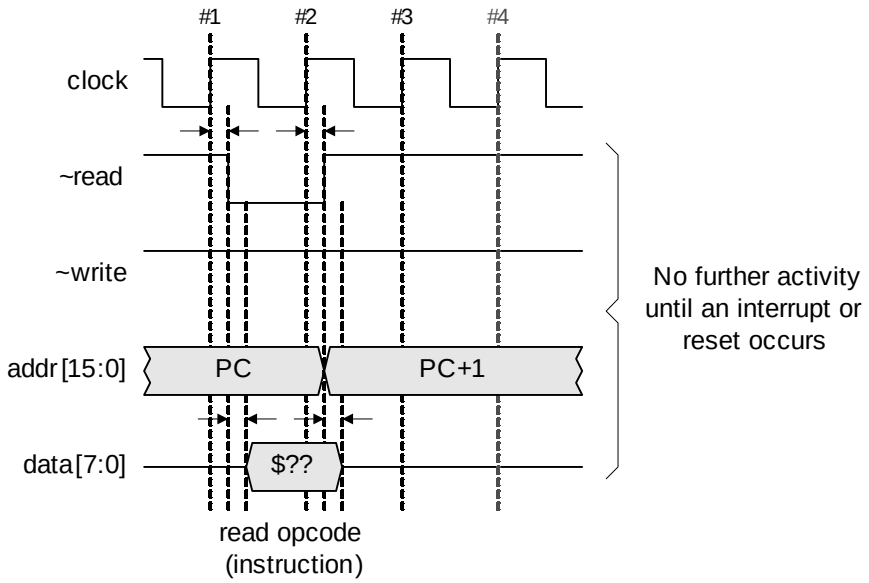


Figure C-12. Timing for implied mode group 1 actions

	Actions for implied mode group 1 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The `~read` signal goes active.
Clock #2	The `~read` signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The instruction is executed. In the case of the `HALT` instruction, this means executing internal `NOPs` (*"no operations"*) until an interrupt occurs.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Implied mode: Group 2

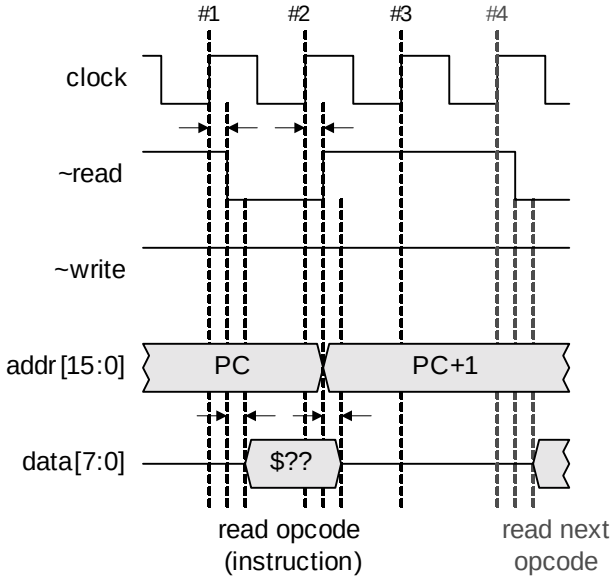


Figure C-13. Timing for implied mode group 2 actions

	Actions for implied mode group 2 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The `~read` signal goes active.
Clock #2	The `~read` signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The instruction is executed and any appropriate status flags are updated.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Implied mode: Group 3

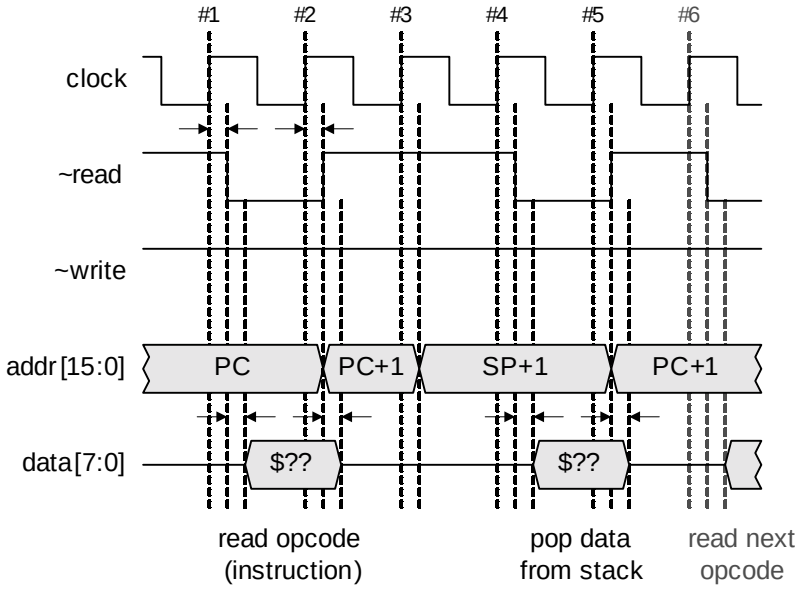


Figure C-14. Timing for implied mode group 3 actions

	Actions for implied mode group 3 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
	Stack pointer contains a value of SP.
Clock #1	The `~read` signal goes active.
Clock #2	The `~read` signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1). As is happens we won't need this value for a while, but the CPU doesn't know this yet.
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by incrementing the stack pointer to contain SP + 1 and also loading this value into the address latch (note #2).
Clock #4	The `~read` signal goes active.
Clock #5	The `~read` signal goes inactive.
	The data byte from the top of the stack is loaded into the appropriate register (either the accumulator or the status register).
	The address latch is loaded with the current contents of the program counter (PC + 2) minus 1; that is (PC + 2) – 1 = PC + 1.

Note #1: The program counter **must** be incremented to contain PC + 2 on clock #2, because the addressing logic is tied up loading the stack pointer into the address latch on clock #3.

Note #2: The stack pointer always points to the first free location on the top of the stack, so in order to access the last byte of data that was written to the stack we have to load the address latch with SP + 1.

Implied mode: Group 4

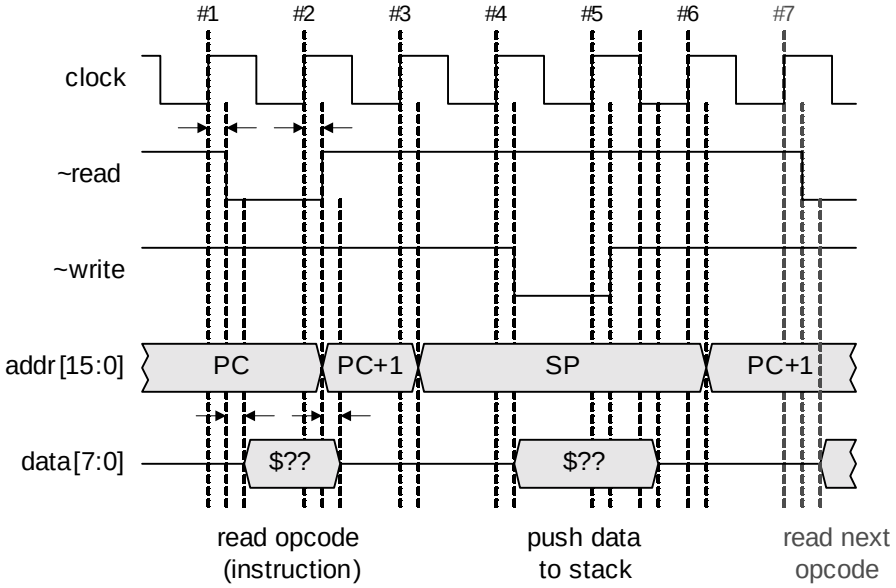


Figure C-15. Timing for implied mode group 4 actions

	Actions for implied mode group 4 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
	Stack pointer contains a value of SP.
Clock #1	The `~read` signal goes active.
Clock #2	The `~read` signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1). As is happens we won't need this value for a while, but the CPU doesn't know this yet.
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by loading the contents of the stack pointer (SP) into the address latch.
	The stack pointer is decremented to contain SP - 1 (note #2).
Clock #4	The `~write` signal goes active.
	The CPU starts driving the contents of the appropriate register (either the accumulator or the status register) onto the data bus.
Clock #5	The `~write` signal goes inactive.
	The data being driven by the CPU is pushed (written) onto the stack.
	The CPU stops driving data on the *falling edge* of clock #5 and returns the data bus to its high-impedance state.
Clock #6	The address latch is loaded with the current contents of the program counter (PC + 2) minus 1; that is (PC + 2) – 1 = PC + 1.

Note #1: The program counter **must** be incremented to contain PC + 2 on clock #2, because the addressing logic is tied up loading the stack pointer into the address latch on clock #3.

Note #2: The stack pointer may be decremented to contain SP – 1 on either clock #3, #4, or #5.

Implied mode: Group 5

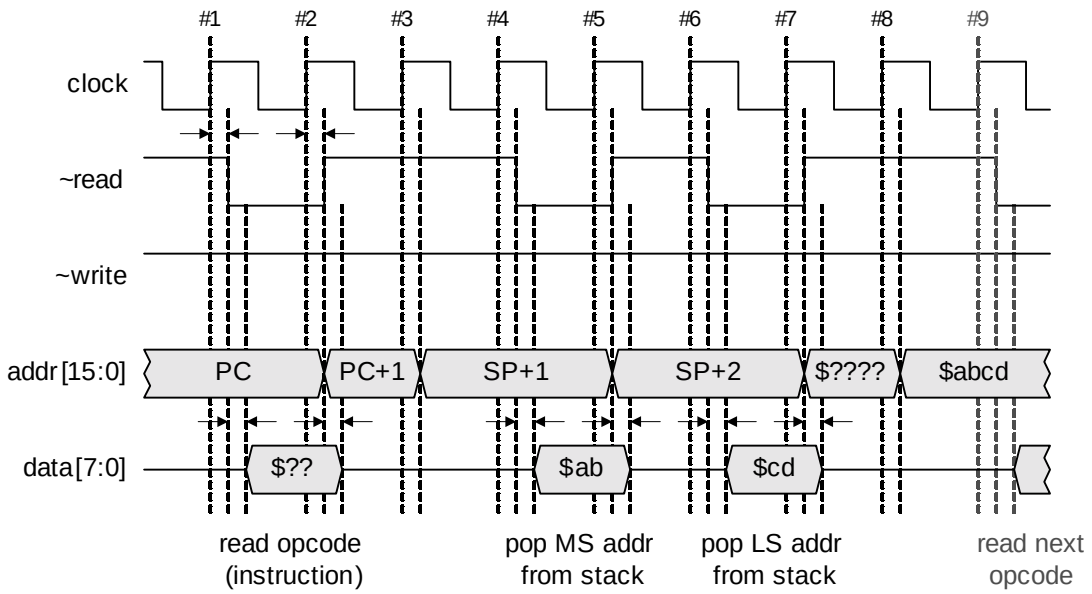


Figure C-16. Timing for implied mode group 5 actions

	Actions for implied mode group 5 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
	Stack pointer contains a value of SP.
Clock #1	The `~read` signal goes active.
Clock #2	The `~read` signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1). In fact we don't need this value, but the CPU doesn't know this yet.
Clock #3	The execution of the instruction commences by loading the address latch with the current value in the stack pointer (SP) plus one, leaving it containing SP + 1 (note #1).
	The stack pointer is incremented to contain SP + 1 (note #2).
Clock #4	The `~read` signal goes active.
Clock #5	The `~read` signal goes inactive.
	The data byte from the top of the stack (which we've shown as "$ab") is loaded into the most-significant byte of temporary register TPCA.
	The address latch is loaded with the current value in the stack pointer (SP + 1) plus one, leaving it containing SP + 2.
Clock #6	The `~read` signal goes active.
	The stack pointer is incremented to contain SP + 2 (note #3).

Clock #7	The ~read signal goes inactive.
	The data byte from the top of the stack (which we've shown as "$cd") is loaded into the least-significant byte of temporary register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #8	The address latch is loaded with the current contents of TPCA ($abcd)
	The program counter is loaded with $abcd + 1.

Note #1: The stack pointer always points to the first free location on the top of the stack, so in order to access the last byte of data that was written to the stack we have to load the address latch with SP + 1.

Note #2: The stack pointer may be incremented to contain SP + 1 on either clock #3 or #4.

Note #3: The stack pointer **must** be incremented to contain SP + 2 on clock #6 (TPCA is loaded on #5).

Implied mode: Group 6

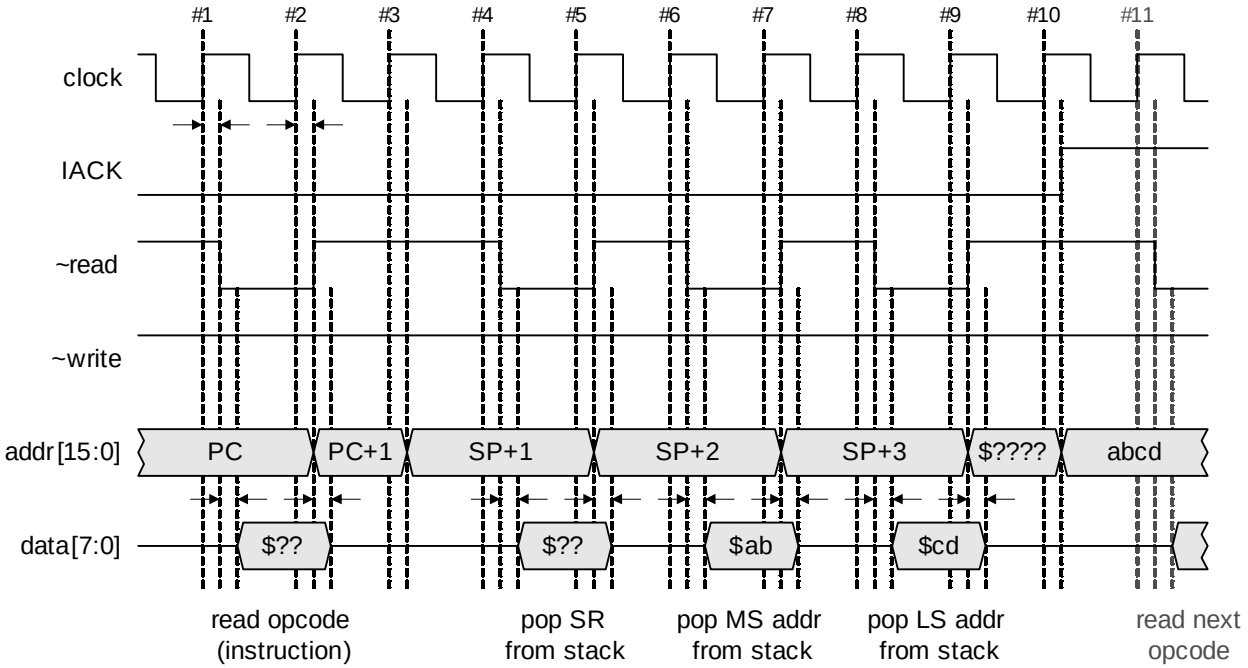


Figure C-17. Timing for implied mode group 6 actions

	Actions for implied mode group 6 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
	Stack pointer contains a value of SP.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1). In fact we don't need this value, but the CPU doesn't know this yet.

Clock #3	The execution of the instruction commences by loading the address latch with the current value in the stack pointer (SP) plus one, leaving it containing SP + 1 (note #1).
	The stack pointer is incremented to contain SP + 1 (note #2).
Clock #4	The ~read signal goes active.
Clock #5	The ~read signal goes inactive.
	The data byte from the top of the stack is loaded into the status register.
	The address latch is loaded with the current value in the stack pointer (SP + 1) plus one, leaving it containing SP + 2.
	The stack pointer is incremented to contain SP + 2 (note #3).
Clock #6	The ~read signal goes active.
Clock #7	The ~read signal goes inactive.
	The data byte from the top of the stack (which we've shown as $ab) is loaded into the most-significant byte of temporary register TPCA.
	The address latch is loaded with the current value in the stack pointer (SP + 2) plus one, leaving it containing SP + 3.
Clock #8	The ~read signal goes active.
	The stack pointer is incremented to contain SP + 3 (note #4).
Clock #9	The ~read signal goes inactive.
	The data byte from the top of the stack (which we've shown as $cd) is loaded into the least-significant byte of temporary register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #10	The address latch is loaded with the current contents of TPCA ($abcd)
	The program counter is loaded with $abcd + 1.
	The IACK (interrupt acknowledge) output returns to its inactive (logic 1) state.

Note #1: The stack pointer always points to the first free location on the top of the stack, so in order to access the last byte of data that was written to the stack we have to load the address latch with SP + 1.

Note #2: The stack pointer may be incremented to contain SP + 1 on either clock #3 or #4.

Note #3: The stack pointer may be incremented to contain SP + 2 on either clock #5 or #6.

Note #4: The stack pointer **must** be incremented to contain SP + 3 on clock #8 (TPCA is loaded on #7).

Immediate mode instructions

Instructions using the *immediate* addressing mode consist of an opcode byte followed by a single operand byte. These instructions can be categorized into two groups as follows:

	Immediate mode instruction groups					
Group 1	ADD	ADDC	SUB	SUBC		
	AND	OR	XOR	CMPA		
Group 2	LDA					

Immediate mode: Group 1

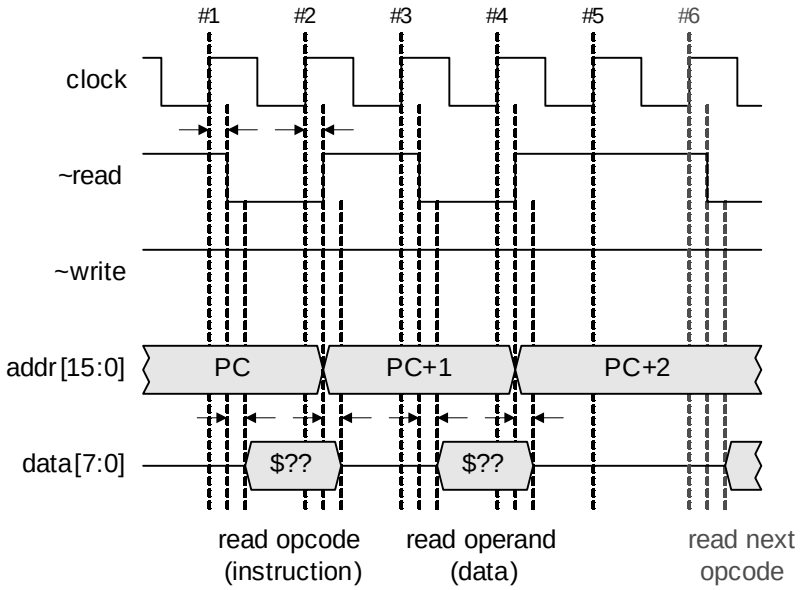


Figure C-18. Timing for immediate mode group 1 actions

	Actions for immediate mode group 1 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The `~read` signal goes active.
Clock #2	The `~read` signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by `~read` going active again.
Clock #4	The `~read` signal goes inactive.
	The operand byte is loaded into the temporary register feeding the "B" inputs to the ALU (the ALU's "A" inputs are driven by the accumulator. See *Chapter 2* for more details).
	The address latch is loaded with the current contents of the program counter (PC + 2).
	The program counter is incremented to contain PC + 3 (note #2).
Clock #5	The instruction is executed and any appropriate status flags are updated.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Note #2: The program counter may be incremented to contain PC + 3 on either clock #4 or #5.

Immediate mode: Group 2

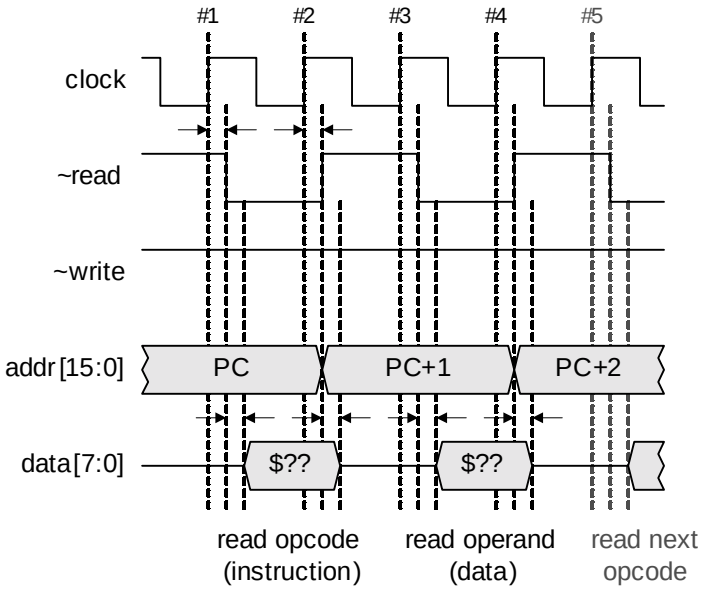


Figure C-19. Timing for immediate mode group 2 actions

	Actions for immediate mode group 2 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The `~read` signal goes active.
Clock #2	The `~read` signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by `~read` going active again.
Clock #4	The `~read` signal goes inactive.
	The operand byte is loaded into the accumulator and any appropriate status flags are updated.
	The address latch is loaded with the current contents of the program counter (PC + 2).
	The program counter is incremented to contain PC + 3

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Big immediate mode instructions

Instructions using the *big immediate* addressing mode consist of opcode byte followed by two operand bytes. The two operand bytes represent a 16-bit data value to be loaded into one of the 16-bit addressing registers: the interrupt vector (IV), stack pointer (SP), or index register (X). Big immediate mode instructions all fall into the same group.

	Big immediate mode instruction groups					
Group 1	BLDIV	BLDSP	BLDX			

Big immediate mode: Group 1

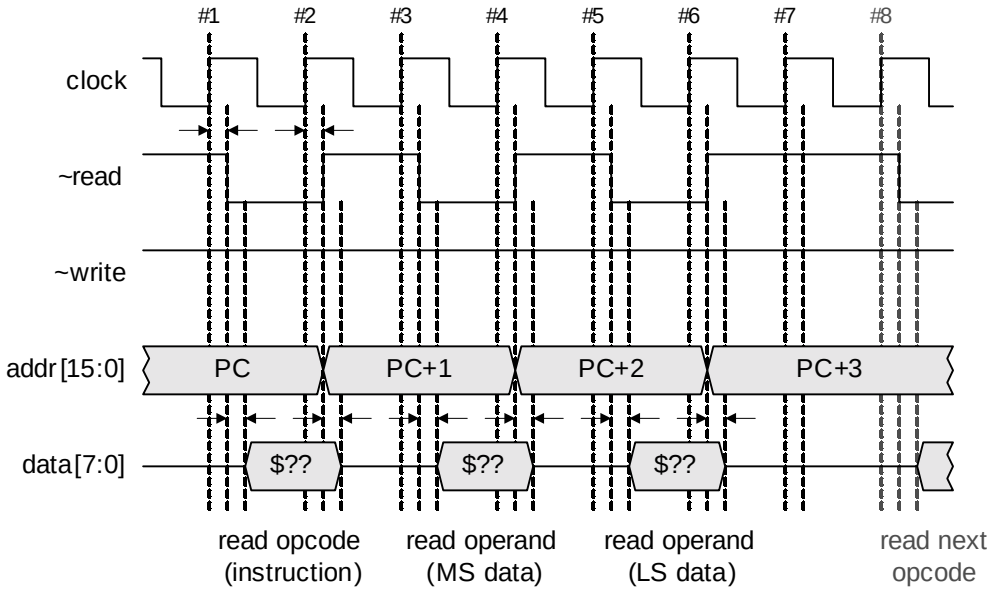


Figure C-20. Timing for big immediate mode group 1 actions

	Actions for big immediate mode group 1 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The `~read` signal goes active.
Clock #2	The `~read` signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by `~read` going active again.

Clock #4	The ~read signal goes inactive.
	The first operand byte is loaded into the most-significant byte of the target addressing register.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
	The program counter is incremented to contain PC + 3 (note #2).
Clock #6	The ~read signal goes inactive.
	The second operand byte is loaded into the least-significant byte of the target addressing register.
	The address latch is loaded with the current contents of the program counter (PC + 3).
Clock #7	The program counter is incremented to contain PC + 4 (note #3).

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Note #2: The program counter **must** be incremented to contain PC + 3 on clock #5 (the MS byte of the target addressing register is being loaded on clock #4).

Note #3: The program counter **must** be incremented to contain PC + 4 on clock #7 (the LS byte of the target addressing register is being loaded on clock #6).

Absolute mode instructions

Instructions using the *absolute* addressing mode consist of opcode byte followed by two operand bytes. The two operand bytes represent a 16-bit address that is used to point to a byte of data (or a byte in which to store data). Absolute mode instructions can be categorized into six distinct groups based on the way in which they are executed:

	Absolute mode instruction groups					
Group 1	ADD	ADDC	SUB	SUBC		
	AND	OR	XOR	CMPA		
Group 2	LDA					
Group 3	STA					
Group 4	JMP					
Group 5	JSR					
Group 6	JC	JNC	JN	JNN	JO	JNO
	JZ	JNZ				

Absolute mode: Group 1

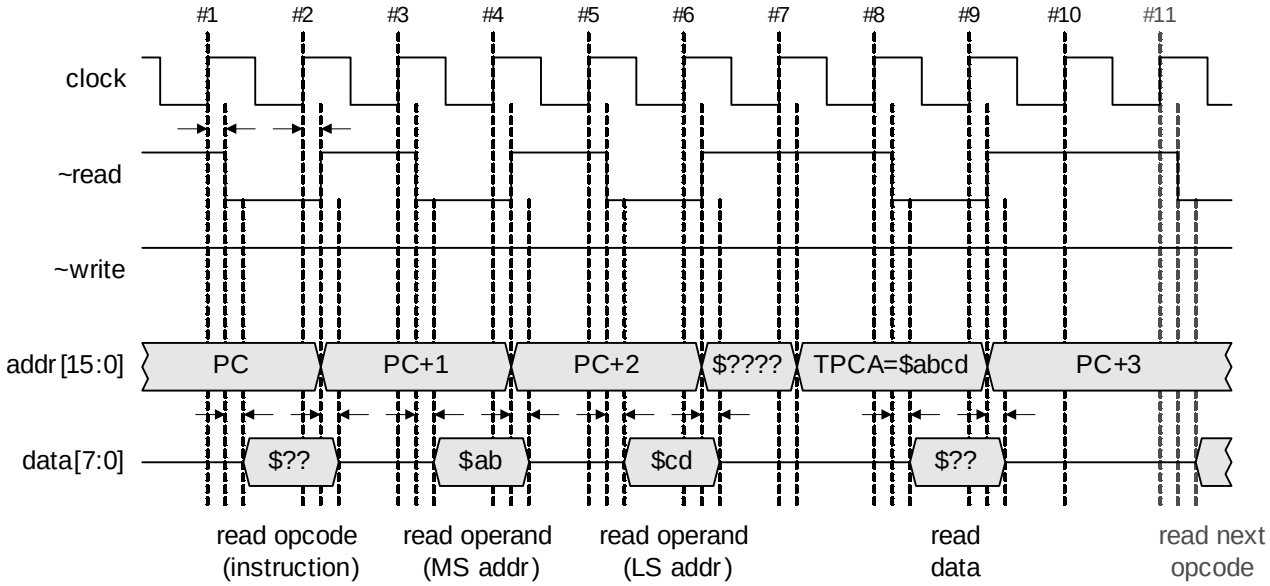


Figure C-21. Timing for absolute mode group 1 actions

	Actions for absolute mode group 1 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
	The program counter is incremented to contain PC + 3 (note #2).
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.

Clock #7	The address latch is loaded with the current contents of TPCA ($abcd).
Clock #8	The `~read` signal goes active.
Clock #9	The `~read` signal goes inactive.
	The operand byte is loaded into the temporary register feeding the “B” inputs to the ALU (the ALU’s “A” inputs are driven by the accumulator. See *Chapter 2* for more details).
	The address latch is loaded with the current contents of the program counter (PC + 3).
	The program counter is incremented to contain PC + 4 (note #3).
Clock #10	The instruction is executed and any appropriate status flags are updated.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Note #2: The program counter **must** be incremented to contain PC + 3 on clock #5 (the MS byte of TPCA is being loaded on clock #4).

Note #3: The program counter may be incremented to contain PC + 4 on either clock #9 or #10. The program counter can’t be incremented to contain PC + 4 on clock #6, because this clock is being used to load the LS byte of TPCA. Similarly, we can’t use clock #7, because this clock is being used to load the address latch with the contents of TPCA. We could use clock #8 to increment the program counter to contain PC + 4 if we wished; but, in this case, clock #9 would now have to load the address latch with the contents of the program counter minus one (that is, (PC + 4) - 1 = PC + 3). Phew!

Absolute mode: Group 2

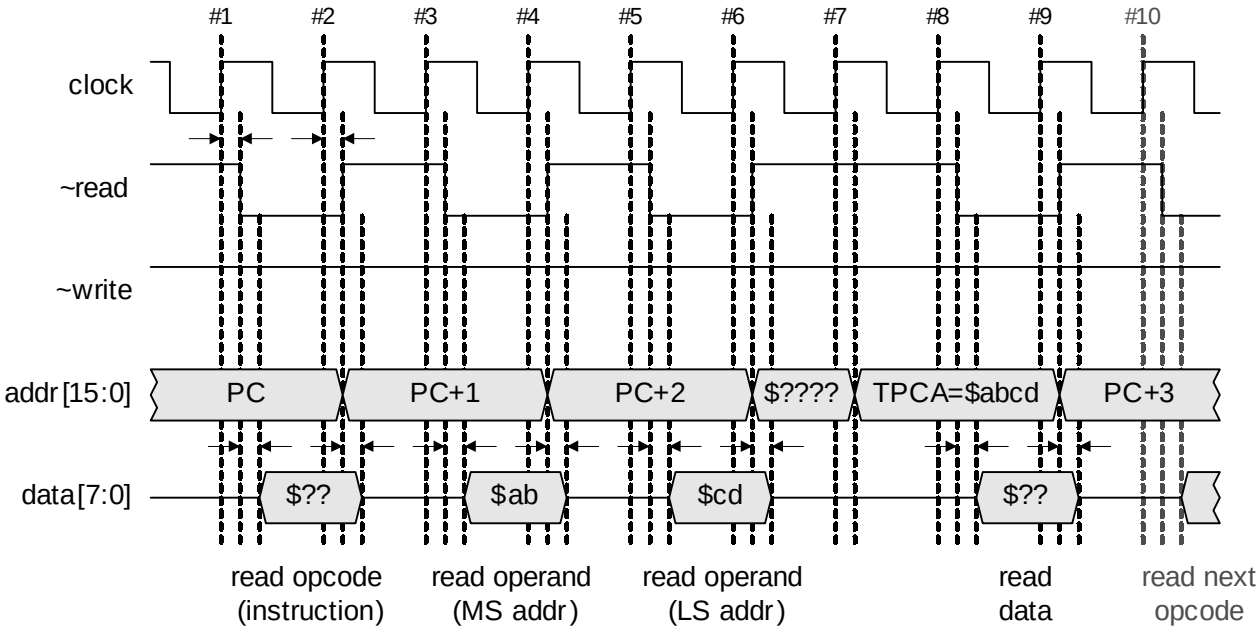


Figure C-22. Timing for absolute mode group 2 actions

	Actions for absolute mode group 2 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
	The program counter is incremented to contain PC + 3 (note #2).
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The address latch is loaded with the current contents of TPCA ($abcd).
Clock #8	The ~read signal goes active.
Clock #9	The ~read signal goes inactive.
	The data byte is loaded into the accumulator and any appropriate status flags are updated.
	The address latch is loaded with the current contents of the program counter (PC + 3).
	The program counter is incremented to contain PC + 4 (note #3).

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Note #2: The program counter **must** be incremented to contain PC + 3 on clock #5 (the MS byte of TPCA is being loaded on clock #4).

Note #3: The program counter can't be incremented to contain PC + 4 on clock #6, because this clock is being used to load the LS byte of TPCA. Similarly, we can't use clock #7, because this clock is being used to load the address latch with the contents of TPCA. We could use clock #8 to increment the program counter to contain PC + 4 if we wished, but in this case clock #9 would now have to load the address latch with the contents of the program counter minus one (that is, (PC + 4) - 1 = PC + 3).

Absolute mode: Group 3

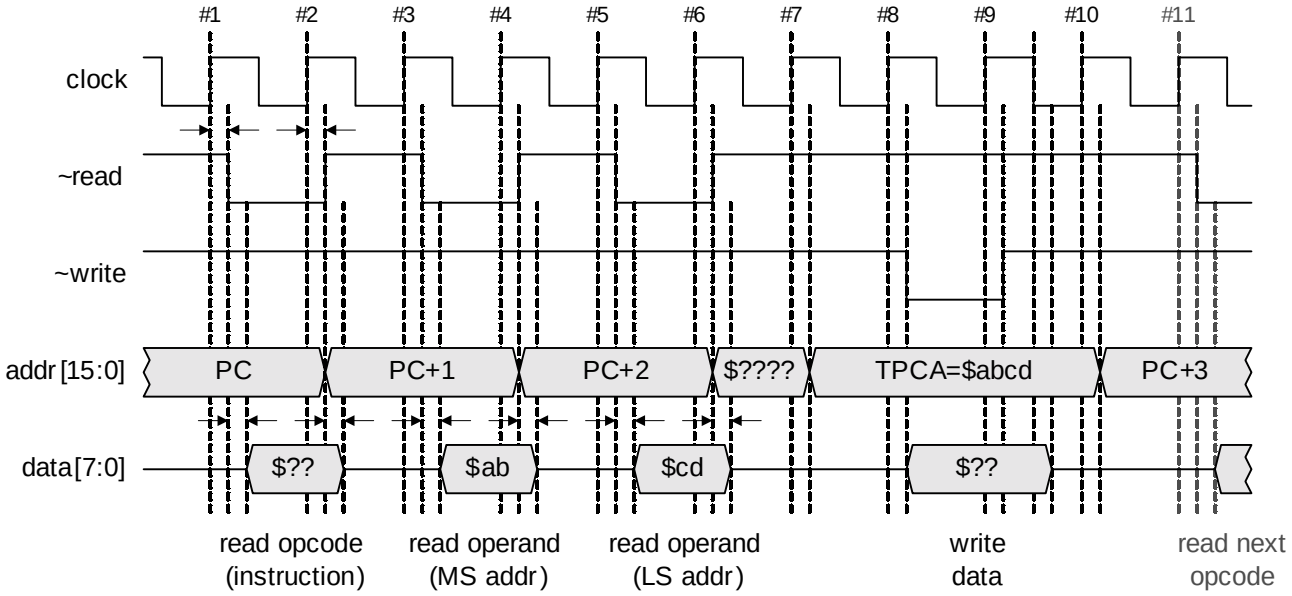


Figure C-23. Timing for absolute mode group 3 actions

	Actions for absolute mode group 3 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
	The program counter is incremented to contain PC + 3 (note #2).
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The address latch is loaded with the current contents of TPCA ($abcd).

Clock #8	The `~write` signal goes active.
	The CPU starts driving the contents of the accumulator onto the data bus.
Clock #9	The `~write` signal goes inactive.
	The data byte from the accumulator is loaded into the targeted address.
	The CPU stops driving data on the *falling edge* of clock #9 and returns the data bus to its high-impedance state.
Clock #10	The address latch is loaded with the current contents of the program counter (PC + 3).
	The program counter is incremented to contain PC + 4 (note #3).

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Note #2: The program counter **must** be incremented to contain PC + 3 on clock #5 (the MS byte of TPCA is being loaded on clock #4).

Note #3: The program counter can't be incremented to contain PC + 4 on clock #6, because this clock is being used to load the LS byte of TPCA. Similarly, we can't use clock #7, because this clock is being used to load the address latch with the contents of TPCA. We could use clocks #8 or #9 to increment the program counter to contain PC + 4 if we wished, but in this case clock #10 would now have to load the address latch with the contents of the program counter minus one (that is, (PC + 4) - 1 = PC + 3).

Absolute mode: Group 4

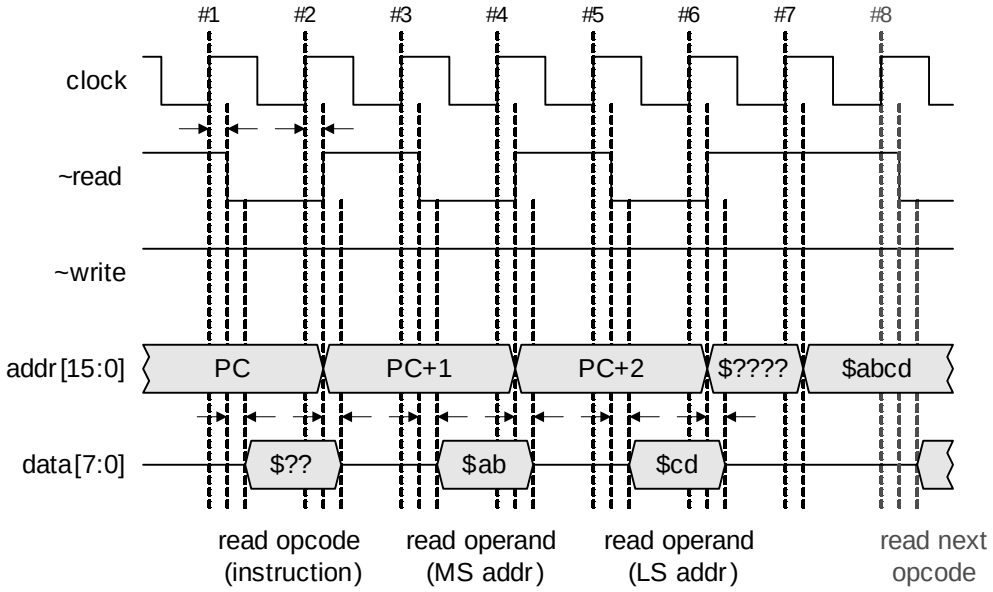


Figure C-24. Timing for absolute mode group 4 actions

	Actions for absolute mode group 4 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The address latch is loaded with the current contents of TPCA ($abcd).
	The program counter is loaded with $abcd + 1.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Absolute mode: Group 5

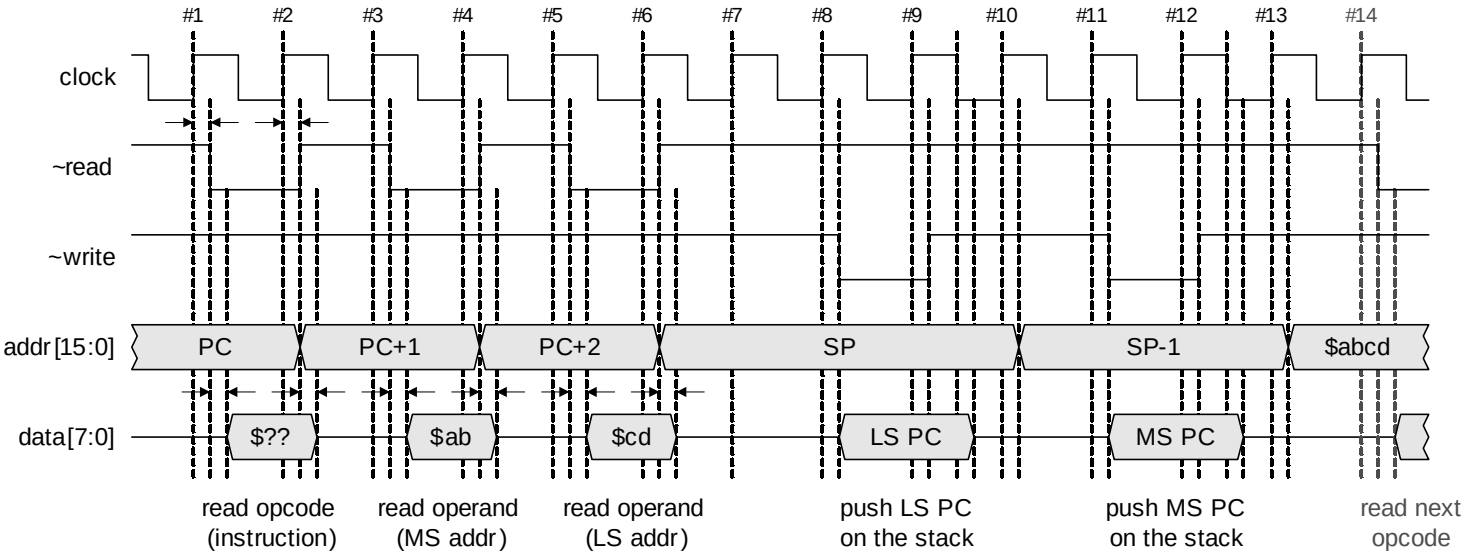


Figure C-25. Timing for absolute mode group 5 actions

	Actions for absolute mode group 5 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
	Stack pointer contains a value of SP.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the stack pointer (SP), which is pointing to the first free location on the top of the stack.
Clock #7	The stack pointer is decremented to contain SP -1, which is the new top of the stack (note #2).
Clock #8	The ~write signal goes active.
	The CPU starts driving the least-significant byte of the program counter out onto the data bus.
Clock #9	The ~write signal goes inactive.
	The least-significant byte from the program counter is stored on the top of the stack (at SP).
	The CPU stops driving the data bus on the *falling edge* of clock #9.
Clock #10	The address latch is loaded with the current contents of the stack pointer (SP - 1), which is pointing to the first free location on the top of the stack.
	The stack pointer is decremented to contain SP -2, which is the new top of the stack.
Clock #11	The ~write signal goes active.
	The CPU starts driving the most-significant byte of the program counter out onto the data bus.
Clock #12	The ~write signal goes inactive.
	The most-significant byte from the program counter is stored on the top of the stack (at SP - 1).
	The CPU stops driving the data bus on the *falling edge* of clock #12.
Clock #13	The address latch is loaded with the current contents of TPCA ($abcd).
	The program counter is loaded with $abcd + 1.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or clock #3.

Note #2: Ideally we would prefer to decrement the stack pointer to contain SP - 1 on clock #6, but this clock is being used to load the LS byte of TPCA, so we have to use an extra clock (#7).

Absolute mode: Group 6

And so we arrive at the *absolute* addressing mode group 6 instructions, which correspond to the conditional jumps (`JZ`, `JNZ`, and so forth). These instructions are quite interesting in that they have two sets of timing diagrams depending on whether the test passes or fails. First let's consider the case where the test passes:

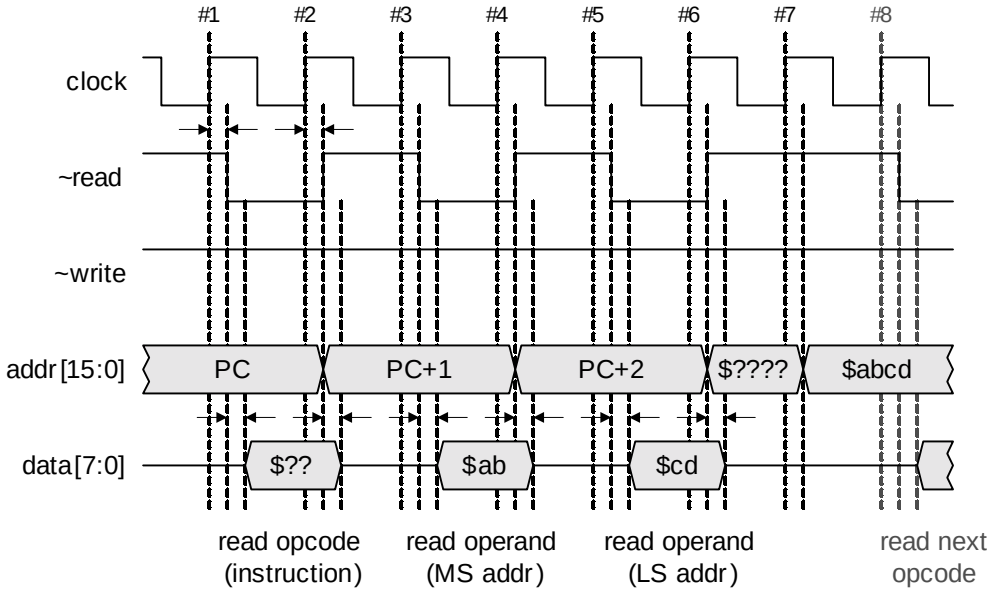


Figure C-26. Timing for absolute mode group 6 actions (test passes)

	Actions for absolute mode (group 6) instructions (test passes)
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The `~read` signal goes active.
Clock #2	The `~read` signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The control logic tests the relevant status flag at this point (in the case of a `JZ` instruction, for example, the control logic will test the value of the zero status flag). If the test passes, the execution of the instruction commences by `~read` going active again.
Clock #4	The `~read` signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The `~read` signal goes active.

Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The address latch is loaded with the current contents of TPCA ($abcd).
	The program counter is loaded with $abcd + 1.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

In the cases where the tests pass, these conditional jump instructions are functionally identical to the unconditional jump (group 4) instructions. Now consider the case where the test fails:

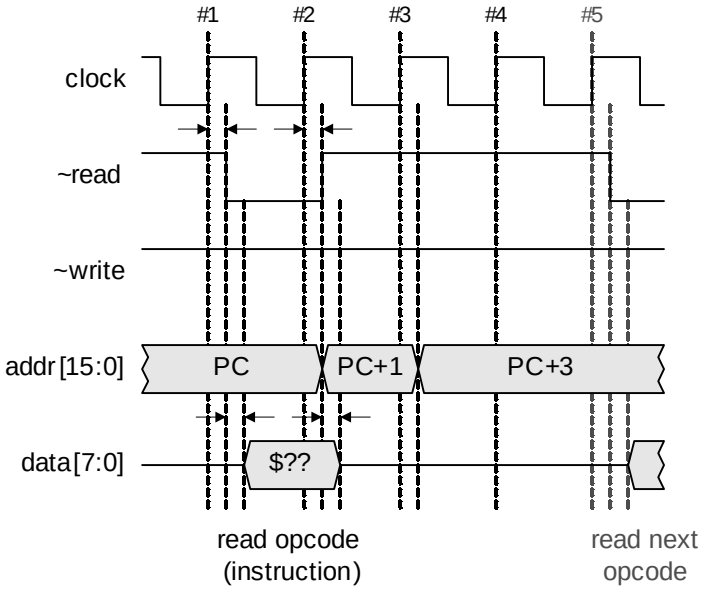


Figure C-27. Timing for absolute mode group 6 actions (test fails)

	Actions for absolute mode group 6 instructions (test fails)
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2.
Clock #3	The control logic tests the relevant status flag at this point. If the test fails, the instruction's execution commences by loading the address latch with the current contents of the program counter plus one: (PC + 2) + 1 = PC + 3.
	The program counter is incremented to contain PC + 3 (note #1).
Clock #4	The program counter is incremented to contain PC + 4 (note #1).

Note #1: The adder block in the CPU's addressing logic does provide the ability to add 2 to whatever value is being presented to its main inputs. Thus, as an alternative to incrementing the program counter to contain PC + 3 on clock #3 and then incrementing it again on clock #4, it would be possible to ignore the program counter on clock #3 and then add 2 to it on clock #4.

Big absolute mode instructions

The *big absolute* addressing mode is very similar to the standard absolute mode, except that big absolute instructions deal with three of our 16-bit addressing registers: the interrupt vector (IV), stack pointer (SP), and index register (X). In this case, the two operand bytes represent a 16-bit address that is used to point to the first byte in a *pair* of data bytes.

	Big absolute mode instruction groups					
Group 1	BLDIV	BLDSP	BLDX			
Group 2		BSTSP	BSTX			

Big absolute mode: Group 1

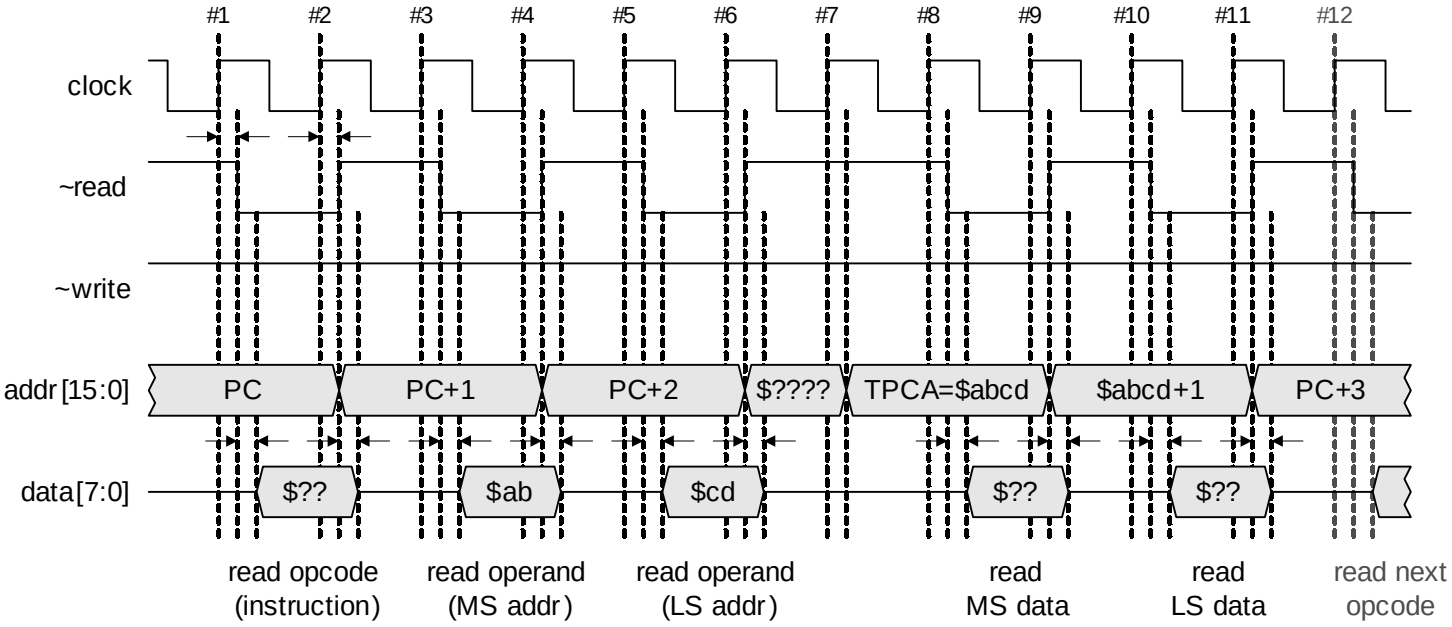


Figure C-28. Timing for big absolute mode group 1 actions

	Actions for big absolute mode group 1 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The `~read` signal goes active.

Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
	A value of 2 is added to the current contents of the program counter, which therefore now contains PC + 4 (note #2).
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The address latch is loaded with the current contents of TPCA ($abcd).
	TPCA is incremented to contain $abcd + 1 (note #3).
Clock #8	The ~read signal goes active.
Clock #9	The ~read signal goes inactive.
	The first data byte is loaded into the most-significant byte of the targeted addressing register.
	The address latch is loaded with the current contents of TPCA ($abcd + 1) (note #3).
Clock #10	The ~read signal goes active.
Clock #11	The ~read signal goes inactive.
	The second data byte is loaded into the least-significant byte of the targeted addressing register.
	The address latch is loaded with the current contents of the program counter minus one: (PC + 4) - 1 = PC + 3 (note #4).

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Note #2: The program counter **must** be modified to contain PC + 4 on clock #5 (the MS byte of TPCA is being loaded on clock #4). The reason for loading the program counter with a value of PC + 4 is that this is what we want it to contain by the end of this instruction.

Note #3: As opposed to using clock #7 to increment the contents of TPCA, we could achieve exactly the same effect by using clock #9 to load the address latch with TPCA + 1.

Note #4: As was discussed in note #2, we want the program counter to end up containing PC + 4, and the only clock free to do this was clock #5. For example, we can't use clock #11, because that's already busy loading the LS byte of the targeted addressing register. Thus, as the program counter already contains a value of PC + 4 by the time we reach clock #11, we use this clock to load the address latch with the contents of the program counter *minus* one, which results in the address latch containing PC + 3 (which is what we wanted in the first place).

Big absolute mode: Group 2

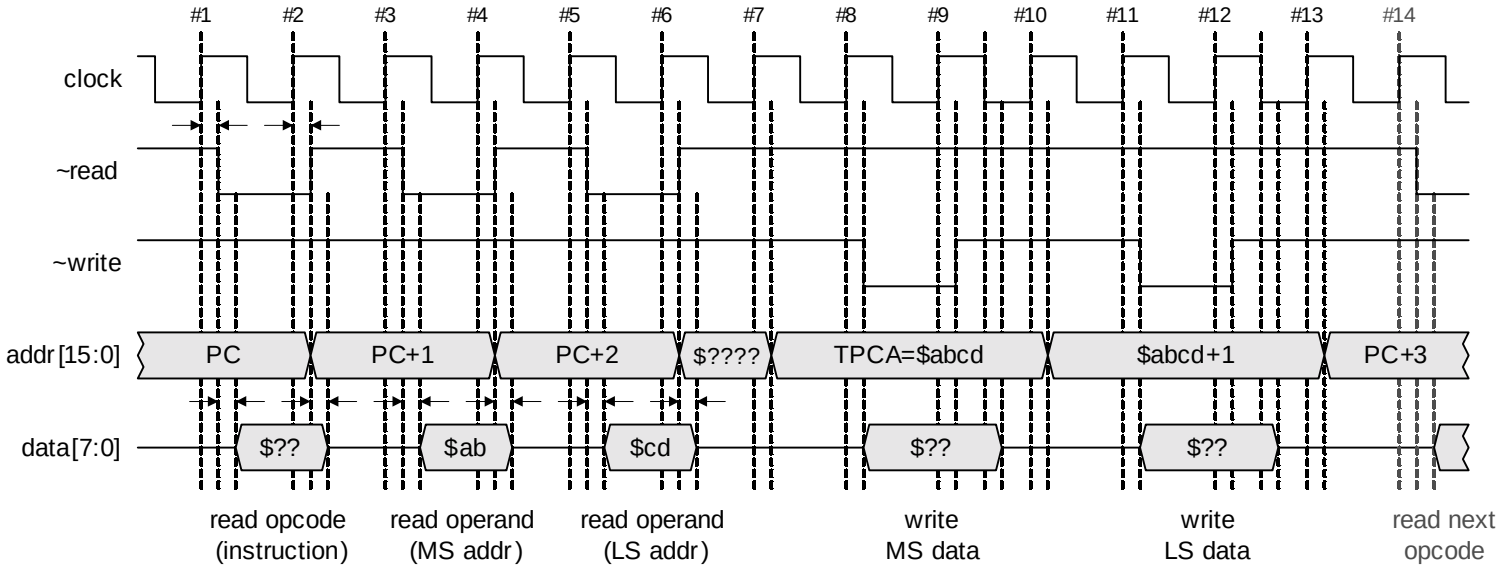


Figure C-29. Timing for big absolute mode group 2 actions

	Actions for big absolute mode group 2 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	Address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
	The program counter is incremented to contain PC + 3 (note #2).
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The address latch is loaded with the current contents of TPCA ($abcd).
	TPCA is incremented to contain $abcd + 1 (note #3).

Clock #8	The ~write signal goes active.
	The CPU starts driving the most-significant byte of the selected addressing register out onto the data bus.
Clock #9	The ~write signal goes inactive.
	The data is loaded into the targeted address ($abcd).
	The CPU stops driving the data bus on the *falling edge* of clock #9.
Clock #10	The address latch is loaded with the current contents of TPCA ($abcd + 1) (note #3).
Clock #11	The ~write signal goes active.
	The CPU starts driving the least-significant byte of the selected addressing register out onto the data bus.
Clock #12	The ~write signal goes inactive.
	The data is loaded into the targeted address ($abcd + 1).
	The CPU stops driving the data bus on the *falling edge* of clock #12.
Clock #13	The address latch is loaded with the current contents of the program counter (PC + 3). The program counter is incremented to contain PC + 4.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Note #2: The program counter **must** be incremented to contain PC + 3 on clock #5 (the MS byte of TPCA is being loaded on clock #4).

Note #3: As opposed to using clock #7 to increment the contents of TPCA, we could achieve exactly the same effect by using clock #10 to load the address latch with TPCA + 1.

Indexed mode instructions

Instructions using the *indexed* addressing mode are very similar to their absolute counterparts, in that they consist of opcode byte followed by two operand bytes, where these two operand bytes represent a 16-bit address. However, this value is now added to the contents of the index register (X), and it is this resulting address that is used to point to a byte of data (or a byte in which to store data). Indexed mode instructions can be categorized into five distinct groups based on the way in which they are executed:

	Indexed mode instruction groups					
Group 1	ADD	ADDC	SUB	SUBC		
	AND	OR	XOR	CMPA		
Group 2	LDA					
Group 3	STA					
Group 4	JMP					
Group 5	JSR					

Indexed mode: Group 1

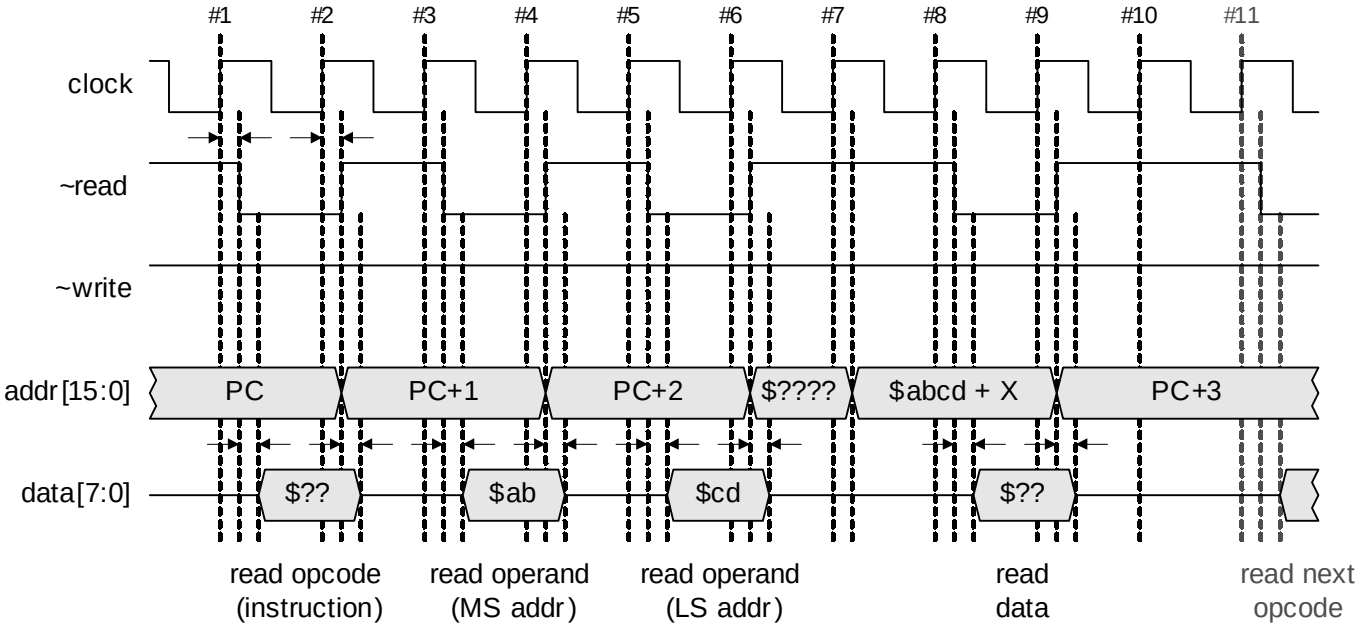


Figure C-30. Timing for indexed mode group 1 actions

	Actions for indexed mode group 1 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
	The program counter is incremented to contain PC + 3 (note #2).
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.

Clock #7	The address latch is loaded with the current contents of TPCA ($abcd) plus the contents of the index register (X). (This doesn't affect the contents of either of these registers.)
Clock #8	The `~read` signal goes active.
Clock #9	The `~read` signal goes inactive.
	The operand byte is loaded into the temporary register feeding the "B" inputs to the ALU (the ALU's "A" inputs are driven by the accumulator. See *Chapter 2* for more details).
	The address latch is loaded with the current contents of the program counter (PC + 3).
	The program counter is incremented to contain PC + 4 (note #3).
Clock #10	The instruction is executed and any appropriate status flags are updated.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Note #2: The program counter **must** be incremented to contain PC + 3 on clock #5 (the MS byte of TPCA is being loaded on clock #4).

Note #3: The program counter may be incremented to contain PC + 4 on either clock #9 or #10. The program counter can't be incremented to contain PC + 4 on clock #6, because this clock is being used to load the LS byte of TPCA. Similarly, we can't use clock #7, because this clock is being used to load the address latch with the contents of TPCA+X. We could use clock #8 to increment the program counter to contain PC + 4 if we wished, but in this case clock #9 would now have to load the address latch with the contents of the program counter minus one (that is, (PC + 4) - 1 = PC + 3). Phew!

Indexed mode: Group 2

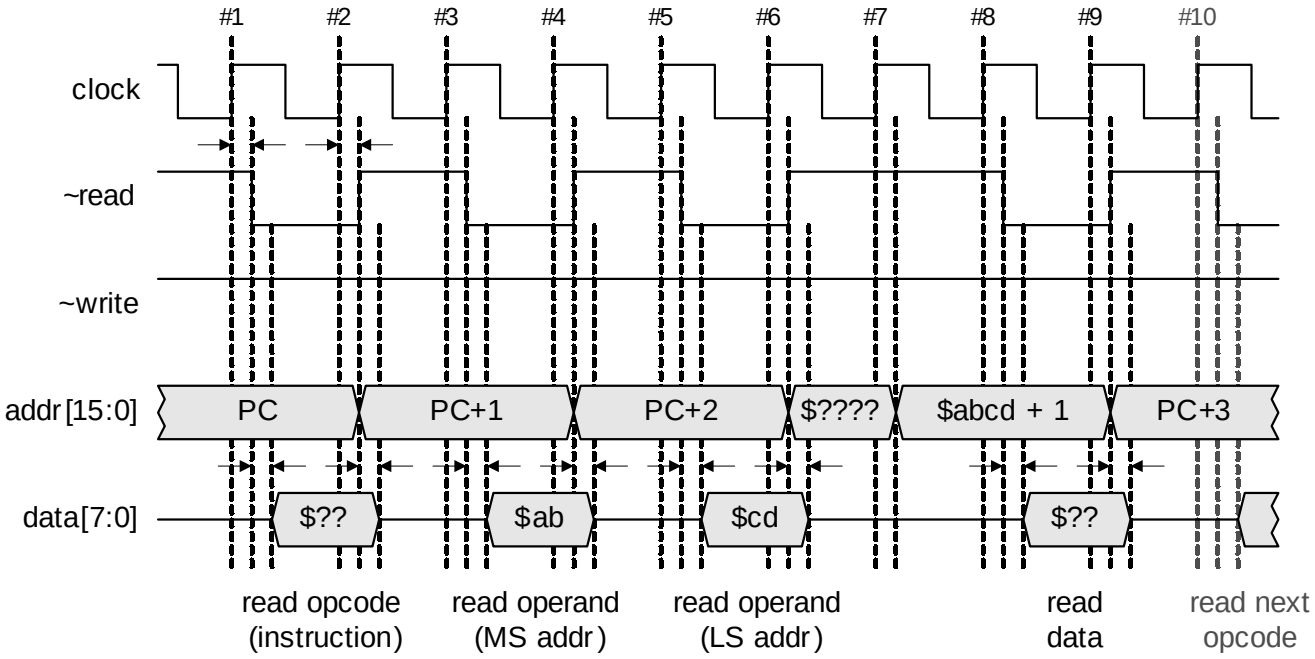


Figure C-31. Timing for indexed mode group 2 actions

	Actions for indexed mode group 2 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
	The program counter is incremented to contain PC + 3 (note #2).
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The address latch is loaded with the current contents of TPCA ($abcd) plus the contents of the index register (X). (This doesn't affect the contents of either of these registers.)
Clock #8	The ~read signal goes active.
Clock #9	The ~read signal goes inactive.
	The data byte is loaded into the accumulator and any appropriate status flags are updated.
	The address latch is loaded with the current contents of the program counter (PC + 3).
	The program counter is incremented to contain PC + 4 (note #3).

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Note #2: The program counter **must** be incremented to contain PC + 3 on clock #5 (the MS byte of TPCA is being loaded on clock #4).

Note #3: The program counter can't be incremented to contain PC + 4 on clock #6, because this clock is being used to load the LS byte of TPCA. Similarly, we can't use clock #7, because this clock is being used to load the address latch with the contents of TPCA+X. We could use clock #8 to increment the program counter to contain PC + 4 if we wished, but in this case clock #9 would now have to load the address latch with the contents of the program counter minus one (that is, (PC + 4) - 1 = PC + 3).

Indexed mode: Group 3

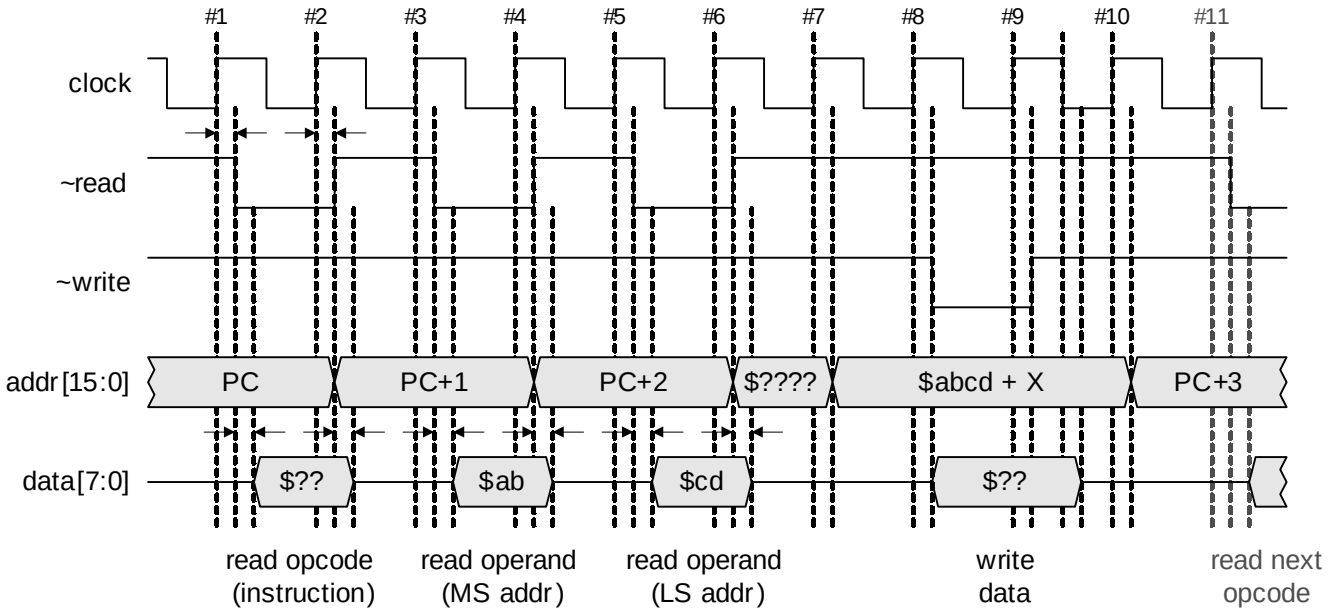


Figure C-32. Timing for indexed mode group 3 actions

	Actions for indexed mode group 3 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
	The program counter is incremented to contain PC + 3 (note #2).
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The address latch is loaded with the current contents of TPCA ($abcd) plus the contents of the index register (X). (This doesn't affect the contents of either of these registers.)
Clock #8	The ~write signal goes active.
	The CPU starts driving the contents of the accumulator onto the data bus.

Clock #9	The ~write signal goes inactive.
	The data byte from the accumulator is loaded into the targeted address.
	The CPU stops driving the data bus on the *falling edge* of clock #9.
Clock #10	The address latch is loaded with the current contents of the program counter (PC + 3).
	The program counter is incremented to contain PC + 4 (note #3).

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Note #2: The program counter **must** be incremented to contain PC + 3 on clock #5 (the MS byte of TPCA is being loaded on clock #4).

Note #3: The program counter can't be incremented to contain PC + 4 on clock #6, because this clock is being used to load the LS byte of TPCA. Similarly, we can't use clock #7, because this clock is being used to load the address latch with the contents of TPCA. We could use clocks #8 or #9 to increment the program counter to contain PC + 4 if we wished, but in this case clock #10 would now have to load the address latch with the contents of the program counter minus one (that is, (PC + 4) - 1 = PC + 3).

Indexed mode: Group 4

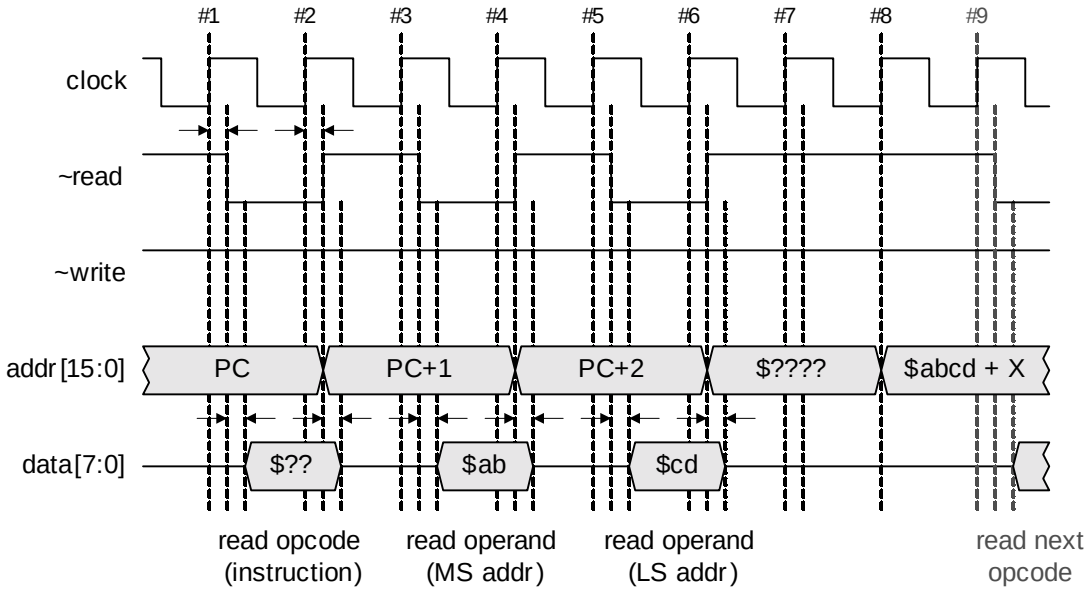


Figure C-33. Timing for indexed mode group 4 actions

	Actions for indexed mode group 4 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).

Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The program counter is loaded with the current contents of TPCA ($abcd) plus the contents of the index register (X). (This doesn't affect the contents of either register.)
Clock #8	The address latch is loaded with the current contents of the program counter ($abcd + X).
	The program counter is incremented to contain ($abcd + X) + 1.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Indexed mode: Group 5

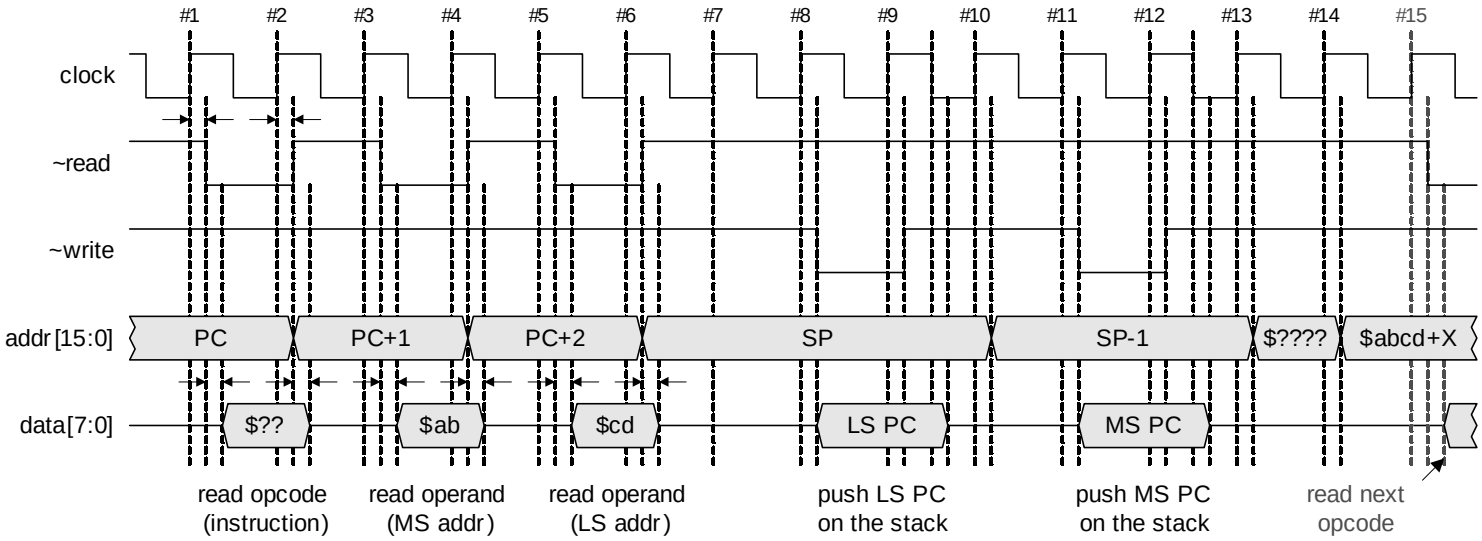


Figure C-34. Timing for indexed mode group 5 actions

	Actions for indexed mode group 5 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
	Stack pointer contains a value of SP.
Clock #1	The ~read signal goes active.

Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the stack pointer (SP), which is pointing to the first free location on the top of the stack.
Clock #7	The stack pointer is decremented to contain SP -1, which is the new top of the stack (note #2).
Clock #8	The ~write signal goes active.
	The CPU starts driving the least-significant byte of the program counter out onto the data bus.
Clock #9	The ~write signal goes inactive.
	The least-significant byte from the program counter is stored on the top of the stack (at SP).
	The CPU stops driving the data bus on the *falling edge* of clock #9.
Clock #10	The address latch is loaded with the current contents of the stack pointer (SP - 1), which is pointing to the first free location on the top of the stack.
	The stack pointer is decremented to contain SP -2, which is the new top of the stack.
Clock #11	The ~write signal goes active.
	The CPU starts driving the most-significant byte of the program counter out onto the data bus.
Clock #12	The ~write signal goes inactive.
	The most-significant byte from the program counter is stored on the top of the stack (at SP - 1).
	The CPU stops driving the data bus on the *falling edge* of clock #12.
Clock #13	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
	The program counter is loaded with the current contents of TPCA ($abcd) plus the contents of the index register (X). (This doesn't affect the contents of the TPCA or X registers).
Clock #14	The address latch is loaded with the current contents of the program counter ($abcd + X).
	The program counter is incremented to contain ($abcd + X) + 1.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or clock #3.

Note #2: Ideally we would prefer to decrement the stack pointer to contain SP - 1 on clock #6, but this clock is being used to load the LS byte of TPCA, so we have to use an extra clock (#7).

Indirect mode instructions

As for an absolute mode instruction, an *indirect* instruction has two address operand bytes following the opcode. However, these two bytes do not point to the target data themselves, but instead they point to the first byte of another pair of address bytes, and it is this second address that points to the data (or to a location in which to store data). Indirect mode instructions can be categorized into four distinct groups based on the way in which they are executed:

	Indirect mode instruction groups						
Group 1	LDA						
Group 2	STA						
Group 3	JMP						
Group 4	JSR						

Indirect mode: Group 1

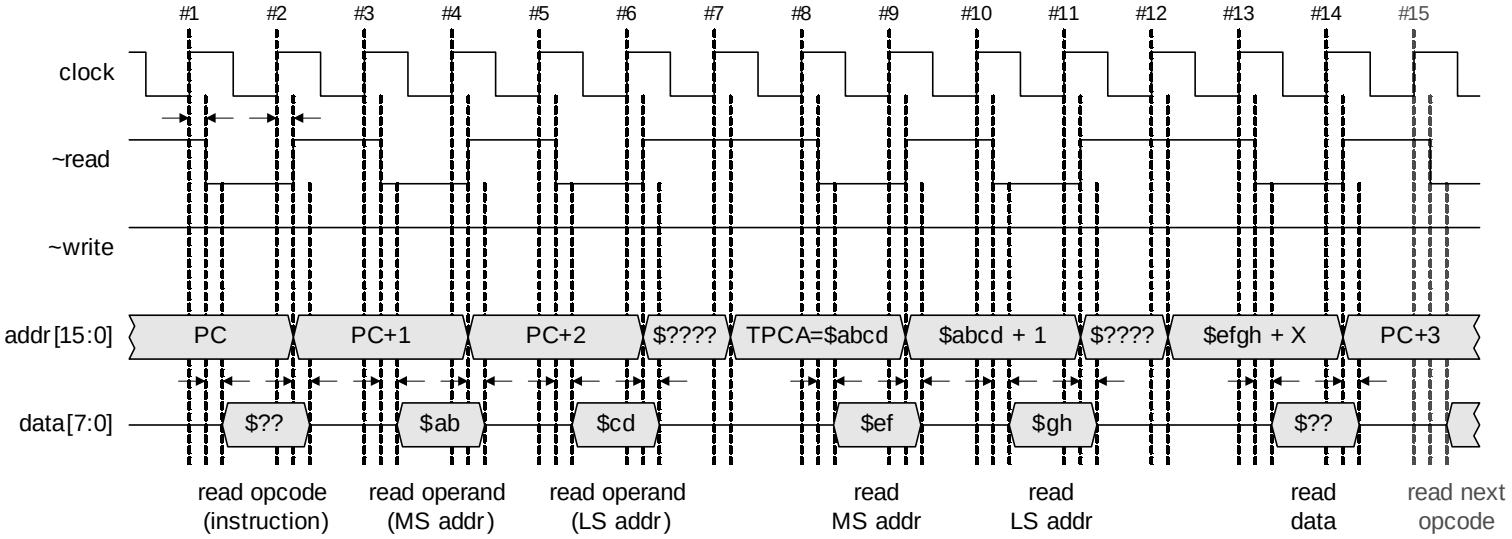


Figure C-35. Timing for indirect mode group 1 actions

	Actions for indirect mode group 1 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The `~read` signal goes active.
Clock #2	The `~read` signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).

Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
	The program counter is incremented to contain PC + 3 (note #2).
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The address latch is loaded with the current contents of TPCA ($abcd).
	TPCA is incremented to contain $abcd + 1 (note #3).
Clock #8	The ~read signal goes active.
Clock #9	The ~read signal goes inactive.
	The most-significant indirect address byte (which we've shown as $ef) is loaded into the most-significant byte of temporary addressing register TPCB.
	The address latch is loaded with the current contents of TPCA ($abcd+1).
Clock #10	The ~read signal goes active.
Clock #11	The ~read signal goes inactive.
	The least-significant indirect address byte (which we've shown as $gh) is loaded into the least-significant byte of temporary addressing register TPCB.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #12	The address latch is loaded with the current contents of TPCB ($efgh).
Clock #13	The ~read signal goes active.
Clock #14	The ~read signal goes inactive.
	The data byte is loaded into the accumulator and any appropriate status flags are updated.
	The address latch is loaded with the current contents of the program counter (PC + 3).
	The program counter is incremented to contain PC + 4.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Note #2: The program counter **must** be incremented to contain PC + 3 on clock #5 (the MS byte of TPCA is being loaded on clock #4).

Note #3: As opposed to incrementing TPCA on clock #7 and then loading this new value into the address latch on clock #9, we could leave TPCA alone on clock #7 and simply load the address latch with TPCA+1 on clock #9.

Indirect mode: Group 2

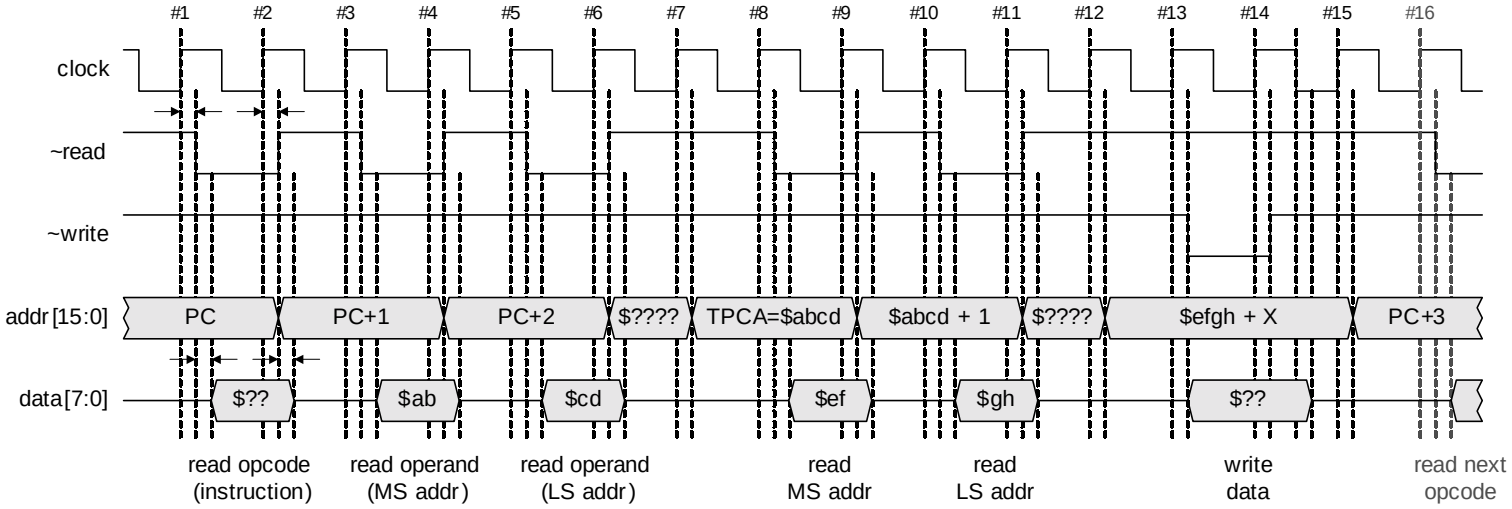


Figure C-36. Timing for indirect mode group 2 actions

	Actions for indirect mode group 2 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
	The program counter is incremented to contain PC + 3 (note #2).
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The address latch is loaded with the current contents of TPCA ($abcd).
	TPCA is incremented to contain $abcd + 1 (note #3).
Clock #8	The ~read signal goes active.

Clock #9	The ~read signal goes inactive.
	The most-significant indirect address byte (which we've shown as $ef) is loaded into the most-significant byte of temporary addressing register TPCB.
	The address latch is loaded with the current contents of TPCA ($abcd+1).
Clock #10	The ~read signal goes active.
Clock #11	The ~read signal goes inactive.
	The least-significant indirect address byte (which we've shown as $gh) is loaded into the least-significant byte of temporary addressing register TPCB.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #12	The address latch is loaded with the current contents of TPCB ($efgh).
Clock #13	The ~write signal goes active.
	The CPU starts driving the contents of the accumulator onto the data bus.
Clock #14	The ~write signal goes inactive.
	The data byte from the accumulator is loaded into the targeted address.
	The CPU stops driving the data bus on the falling edge of clock #14.
Clock #15	The address latch is loaded with the current contents of the program counter (PC + 3).
	The program counter is incremented to contain PC + 4.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Note #2: The program counter **must** be incremented to contain PC + 3 on clock #5 (the MS byte of TPCA is being loaded on clock #4).

Note #3: As opposed to incrementing TPCA on clock #7 and then loading this new value into the address latch on clock #9, we could leave TPCA alone on clock #7 and simply load the address latch with TPCA+1 on clock #9.

Indirect mode: Group 3

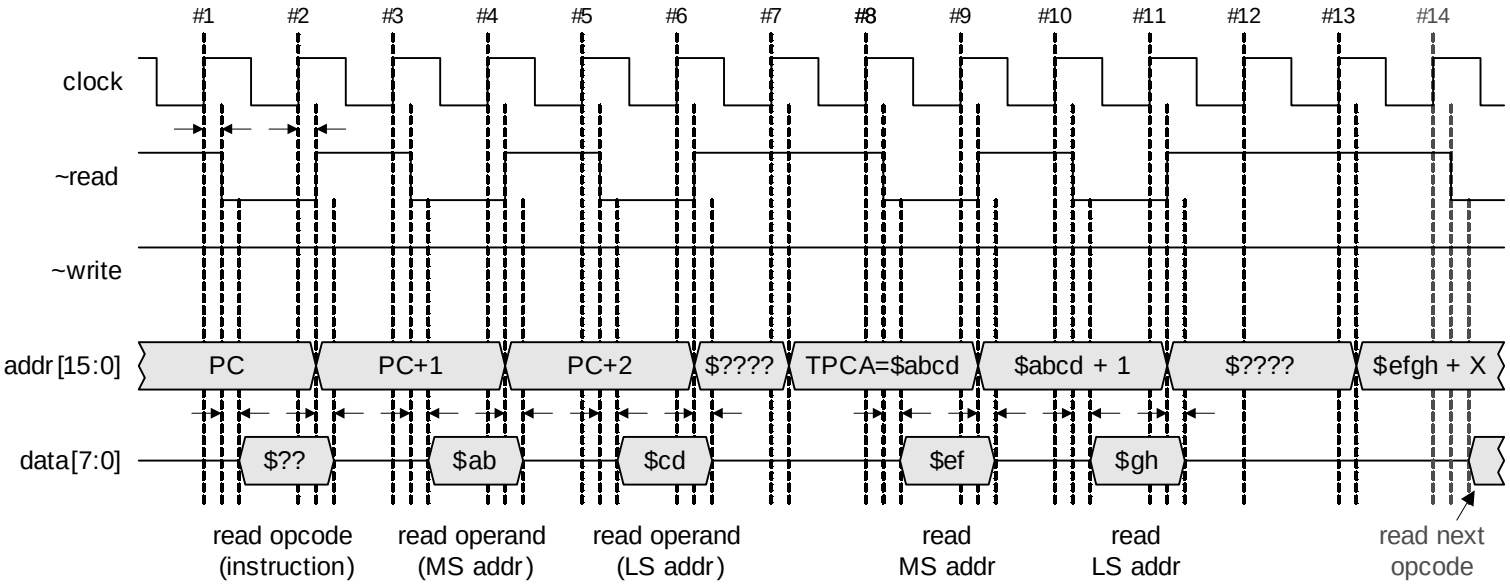


Figure C-37. Timing for indirect mode group 3 actions

	Actions for indirect mode group 3 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The address latch is loaded with the current contents of TPCA ($abcd).
	TPCA is incremented to contain $abcd + 1 (note #2).
Clock #8	The ~read signal goes active.
Clock #9	The ~read signal goes inactive.
	The most-significant indirect address byte (which we've shown as $ef) is loaded into the most-significant byte of temporary addressing register TPCB.
	The address latch is loaded with the current contents of TPCA ($abcd+1).
Clock #10	The ~read signal goes active.
Clock #11	The ~read signal goes inactive.
	The least-significant indirect address byte (which we've shown as $gh) is loaded into the least-significant byte of temporary addressing register TPCB.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #12	The address latch is loaded with the current contents of TPCB ($efgh).
	The program counter is loaded with $efgh + 1.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Note #2: As opposed to incrementing TPCA on clock #7 and then loading this new value into the address latch on clock #9, we could leave TPCA alone on clock #7 and simply load the address latch with TPCA+1 on clock #9.

Indirect mode: Group 4

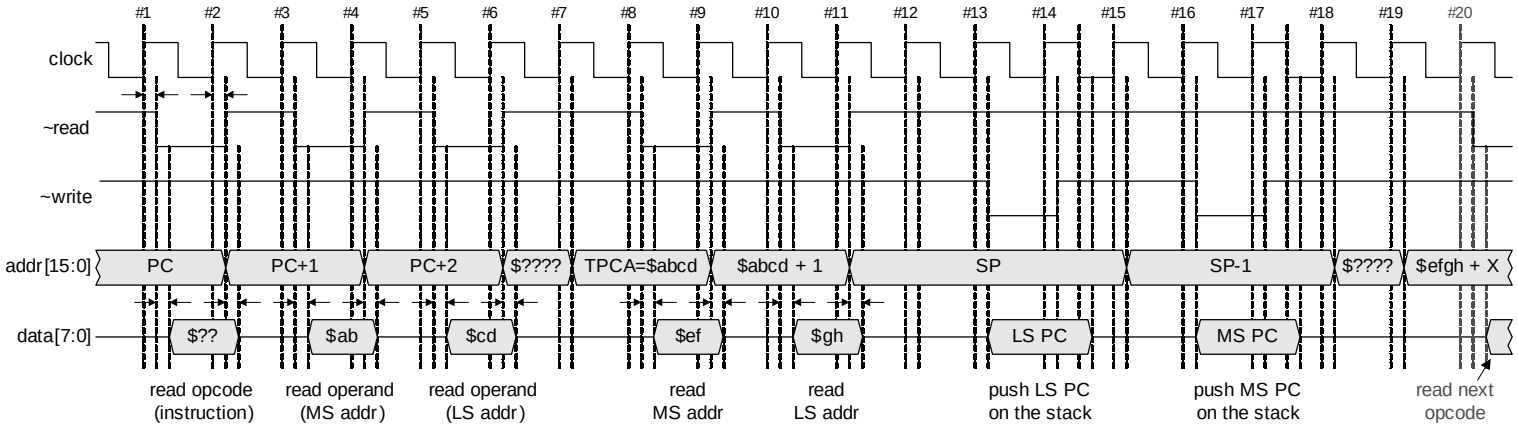


Figure C-38. Timing for indirect mode group 4 actions

	Actions for indirect mode group 4 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
	Stack pointer contains a value of SP.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The address latch is loaded with the current contents of TPCA ($abcd).
	TPCA is incremented to contain $abcd + 1 (note #2).
Clock #8	The ~read signal goes active.
Clock #9	The ~read signal goes inactive.
	The most-significant indirect address byte (which we've shown as $ef) is loaded into the most-significant byte of temporary addressing register TPCB.
	The address latch is loaded with the current contents of TPCA ($abcd+1).
Clock #10	The ~read signal goes active.

Clock #11	The ~read signal goes inactive.
	The least-significant indirect address byte (which we've shown as $gh) is loaded into the least-significant byte of temporary addressing register TPCB.
	The address latch is loaded with the current contents of the stack pointer (SP), which is pointing to the first free location on the top of the stack.
Clock #12	The stack pointer is decremented to contain SP -1, which is the new top of the stack (note #3).
Clock #13	The ~write signal goes active.
	The CPU starts driving the least-significant byte of the program counter out onto the data bus.
Clock #14	The ~write signal goes inactive.
	The least-significant byte from the program counter is stored on the top of the stack (at SP).
	The CPU stops driving the data bus on the *falling edge* of clock #14.
Clock #15	The address latch is loaded with the current contents of the stack pointer (SP - 1), which is pointing to the first free location on the top of the stack.
	The stack pointer is decremented to contain SP -2, which is the new top of the stack.
Clock #16	The ~write signal goes active.
	The CPU starts driving the most-significant byte of the program counter out onto the data bus.
Clock #17	The ~write signal goes inactive.
	The most-significant byte from the program counter is stored on the top of the stack (at SP - 1).
	The CPU stops driving the data bus on the *falling edge* of clock #17.
Clock #18	The address latch is loaded with the current contents of TPCB ($efgh).
	The program counter is loaded with $efgh + 1.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or clock #3.

Note #2: As opposed to incrementing TPCA on clock #7 and then loading this new value into the address latch on clock #9, we could leave TPCA alone on clock #7 and simply load the address latch with TPCA+1 on clock #9.

Note #3: Ideally we would prefer to decrement the stack pointer to contain SP - 1 on clock #11, but this clock is being used to load the LS byte of TPCB, so we have to use an extra clock (#12).

Pre-indexed indirect mode instructions

The *pre-indexed indirect* addressing mode is a combination of the indexed and indirect modes. This form of addressing is so-named because the address stored in the operand bytes is first added to the contents of the index register (X). The resulting address points to the first byte of another pair of address bytes, and it is this new address that points to the data (or to a location in which to store data). Pre-indexed indirect mode instructions can be categorized into four distinct groups based on the way in which they are executed:

	Pre-indexed indirect mode instruction groups					
Group 1	LDA					
Group 2	STA					
Group 3	JMP					
Group 4	JSR					

Pre-indexed indirect mode: Group 1

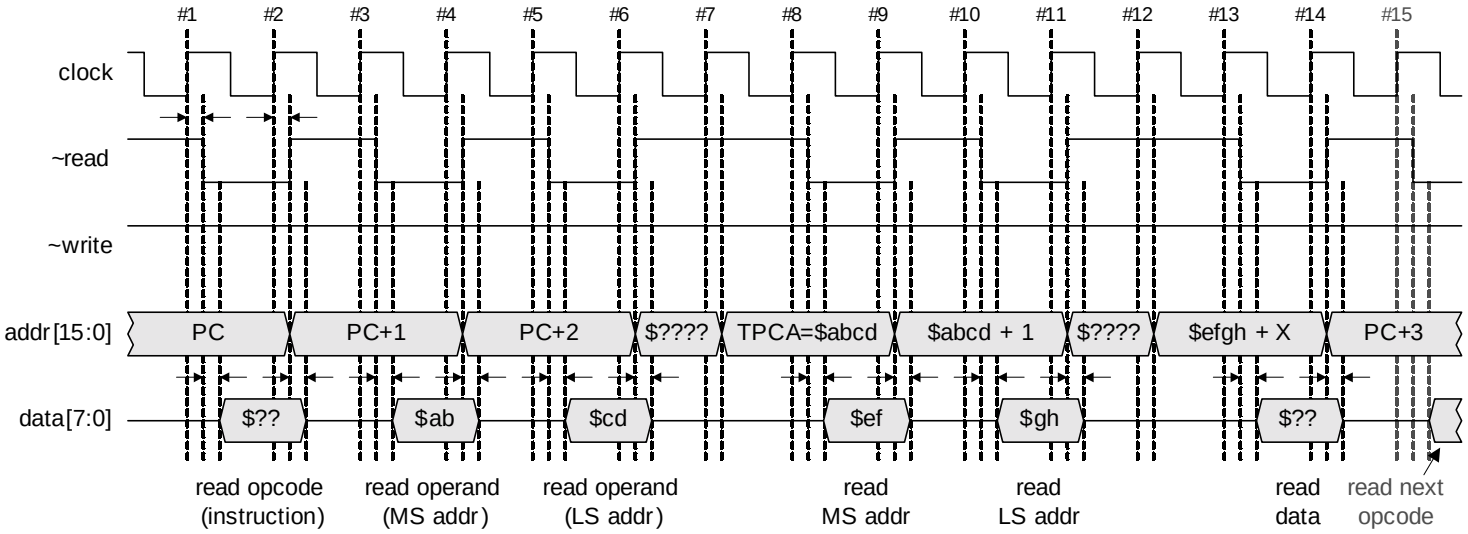


Figure C-39. Timing for pre-indexed indirect mode group 1 actions

	Actions for pre-indexed indirect mode group 1 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
	The program counter is incremented to contain PC + 3 (note #2).
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The address latch is loaded with the current contents of TPCA ($abcd) plus the contents of the index register (X) to give ($abcd + X).
	The contents of TPCA ($abcd) are added to the contents of the index register (X), and the result ($abcd + X) is stored back into TPCA.

Clock #8	The ~read signal goes active.
Clock #9	The ~read signal goes inactive.
	The most-significant indirect address byte (which we've shown as $ef) is loaded into the most-significant byte of temporary addressing register TPCB.
	The address latch is loaded with the current contents of TPCA (which are $abcd + X from clock #7) plus 1, which equals $abcd + X + 1.
Clock #10	The ~read signal goes active.
Clock #11	The ~read signal goes inactive.
	The least-significant indirect address byte (which we've shown as $gh) is loaded into the least-significant byte of temporary addressing register TPCB.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #12	The address latch is loaded with the current contents of TPCB ($efgh).
Clock #13	The ~read signal goes active.
Clock #14	The ~read signal goes inactive.
	The data byte is loaded into the accumulator and any appropriate status flags are updated.
	The address latch is loaded with the current contents of the program counter (PC + 3).
	The program counter is incremented to contain PC + 4.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Note #2: The program counter **must** be incremented to contain PC + 3 on clock #5 (the MS byte of TPCA is being loaded on clock #4).

Note #3: As opposed to incrementing TPCA on clock #7 and then loading this new value into the address latch on clock #9, we could leave TPCA alone on clock #7 and simply load the address latch with TPCA+1 on clock #9.

Pre-indexed indirect mode: Group 2

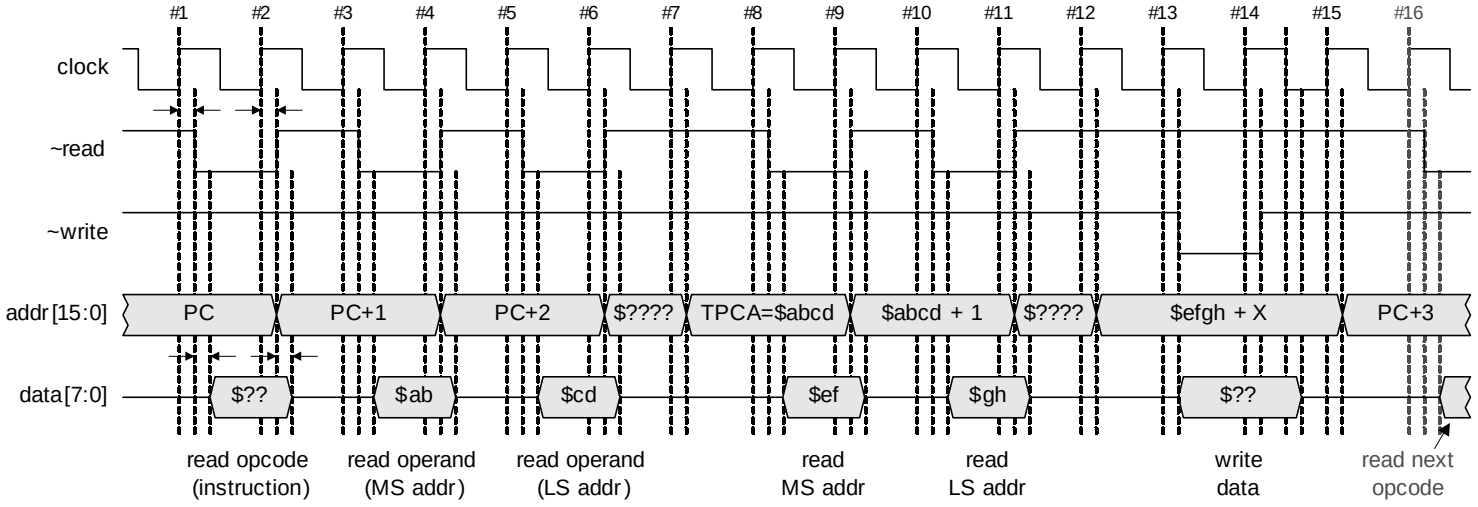


Figure C-40. Timing for pre-indexed indirect mode group 2 actions

	Actions for pre-indexed indirect mode group 2 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
	The program counter is incremented to contain PC + 3 (note #2).
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The address latch is loaded with the current contents of TPCA ($abcd) plus the contents of the index register (X) to give ($abcd + X).
	The contents of TPCA ($abcd) are added to the contents of the index register (X), and the result ($abcd + X) is stored back into TPCA.
Clock #8	The ~read signal goes active.
Clock #9	The ~read signal goes inactive.
	The most-significant indirect address byte (which we've shown as $ef) is loaded into the most-significant byte of temporary addressing register TPCB.
	The address latch is loaded with the current contents of TPCA (which are $abcd + X from clock #7) plus 1, which equals $abcd + X + 1.
Clock #10	The ~read signal goes active.
Clock #11	The ~read signal goes inactive.
	The least-significant indirect address byte (which we've shown as $gh) is loaded into the least-significant byte of temporary addressing register TPCB.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #12	The address latch is loaded with the current contents of TPCB ($efgh).
Clock #13	The ~write signal goes active.
	The CPU starts driving the contents of the accumulator onto the data bus.
Clock #14	The ~write signal goes inactive.
	The data byte from the accumulator is loaded into the targeted address.
	The CPU stops driving the data bus on the falling edge of clock #14.
Clock #15	The address latch is loaded with the current contents of the program counter (PC + 3).
	The program counter is incremented to contain PC + 4.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Note #2: The program counter **must** be incremented to contain PC + 3 on clock #5 (the MS byte of TPCA is being loaded on clock #4).

Pre-indexed indirect mode: Group 3

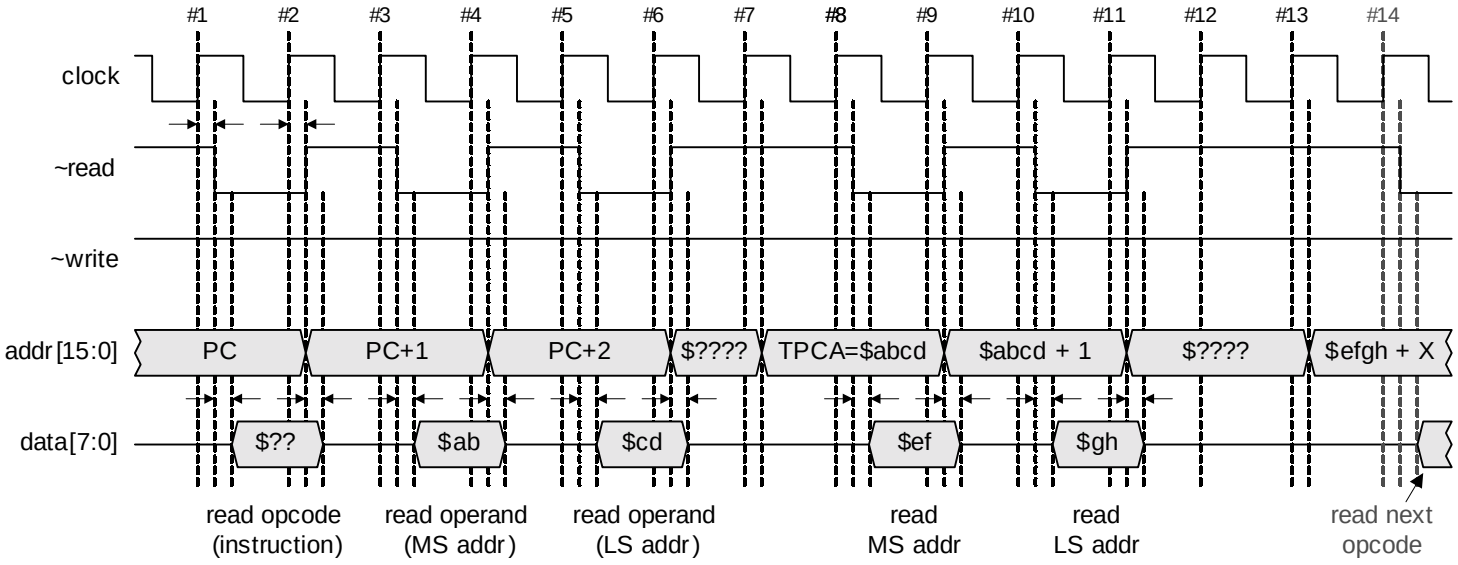


Figure C-41. Timing for pre-indexed indirect mode group 3 actions

	Actions for pre-indexed indirect mode group 3 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.

Clock #7	The address latch is loaded with the current contents of TPCA ($abcd) plus the contents of the index register (X) to give ($abcd + X).
	The contents of TPCA ($abcd) are added to the contents of the index register (X), and the result ($abcd + X) is stored back into TPCA.
Clock #8	The ~read signal goes active.
Clock #9	The ~read signal goes inactive.
	The most-significant indirect address byte (which we've shown as $ef) is loaded into the most-significant byte of temporary addressing register TPCB.
	The address latch is loaded with the current contents of TPCA (which are $abcd + X from clock #7) plus 1, which equals $abcd + X + 1.
Clock #10	The ~read signal goes active.
Clock #11	The ~read signal goes inactive.
	The least-significant indirect address byte (which we've shown as $gh) is loaded into the least-significant byte of temporary addressing register TPCB.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #12	The address latch is loaded with the current contents of TPCB ($efgh).
	The program counter is loaded with $efgh + 1.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Pre-indexed indirect mode: Group 4

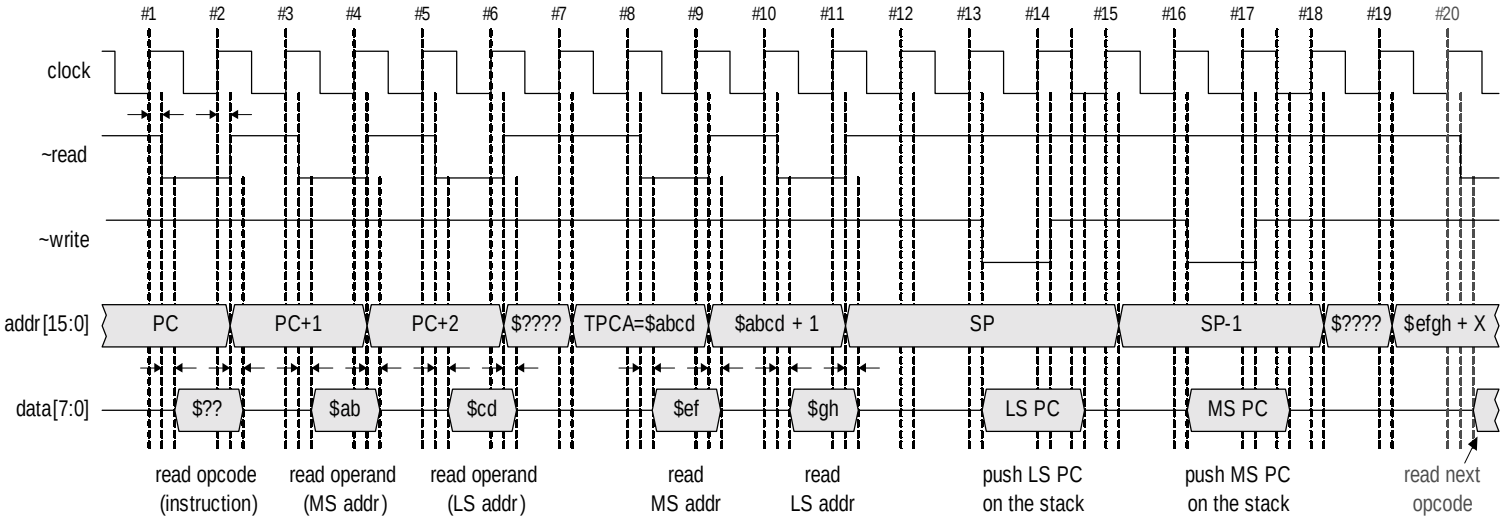


Figure C-42. Timing for pre-indexed indirect mode group 4 actions

	Actions for pre-indexed indirect mode group 4 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
	Stack pointer contains a value of SP.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The address latch is loaded with the current contents of TPCA ($abcd) plus the contents of the index register (X) to give ($abcd + X).
	The contents of TPCA ($abcd) are added to the contents of the index register (X), and the result ($abcd + X) is stored back into TPCA.
Clock #8	The ~read signal goes active.
Clock #9	The ~read signal goes inactive.
	The most-significant indirect address byte (which we've shown as $ef) is loaded into the most-significant byte of temporary addressing register TPCB.
	The address latch is loaded with the current contents of TPCA (which are $abcd + X from clock #7) plus 1, which equals $abcd + X + 1.
Clock #10	The ~read signal goes active.
Clock #11	The ~read signal goes inactive.
	The least-significant indirect address byte (which we've shown as $gh) is loaded into the least-significant byte of temporary addressing register TPCB.
	The address latch is loaded with the current contents of the stack pointer (SP), which is pointing to the first free location on the top of the stack.
Clock #12	The stack pointer is decremented to contain SP -1, which is the new top of the stack (note #2).
Clock #13	The ~write signal goes active.
	The CPU starts driving the least-significant byte of the program counter out onto the data bus.
Clock #14	The ~write signal goes inactive.
	The least-significant byte from the program counter is stored on the top of the stack (at SP).
	The CPU stops driving the data bus on the *falling edge* of clock #14.

Clock #15	The address latch is loaded with the current contents of the stack pointer (SP - 1), which is pointing to the first free location on the top of the stack.
	The stack pointer is decremented to contain SP -2, which is the new top of the stack.
Clock #16	The `~write` signal goes active.
	The CPU starts driving the most-significant byte of the program counter out onto the data bus.
Clock #17	The `~write` signal goes inactive.
	The most-significant byte from the program counter is stored on the top of the stack (at SP - 1).
	The CPU stops driving the data bus on the *falling edge* of clock #17.
Clock #18	The address latch is loaded with the current contents of TPCB ($efgh).
	The program counter is loaded with $efgh + 1.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or clock #3.

Note #2: Ideally we would prefer to decrement the stack pointer to contain SP - 1 on clock #11, but this clock is being used to load the LS byte of TPCB, so we have to use an extra clock (#12).

Indirect post-indexed mode instructions

As for the pre-indexed indirect instructions we discussed in the previous section, *indirect post-indexed* addressing is a combination of the indexed and indirect modes. In this case, however, the address stored in the operand bytes points to the first byte of another pair of address bytes. This second address is then added to the contents of the index register (X), and the resulting address points to the data (or to a location in which to store data). Indirect post-indexed mode instructions can be categorized into four distinct groups based on the way in which they are executed:

	Indirect post-indexed mode instruction groups					
Group 1	LDA					
Group 2	STA					
Group 3	JMP					
Group 4	JSR					

Indirect post-indexed mode: Group 1

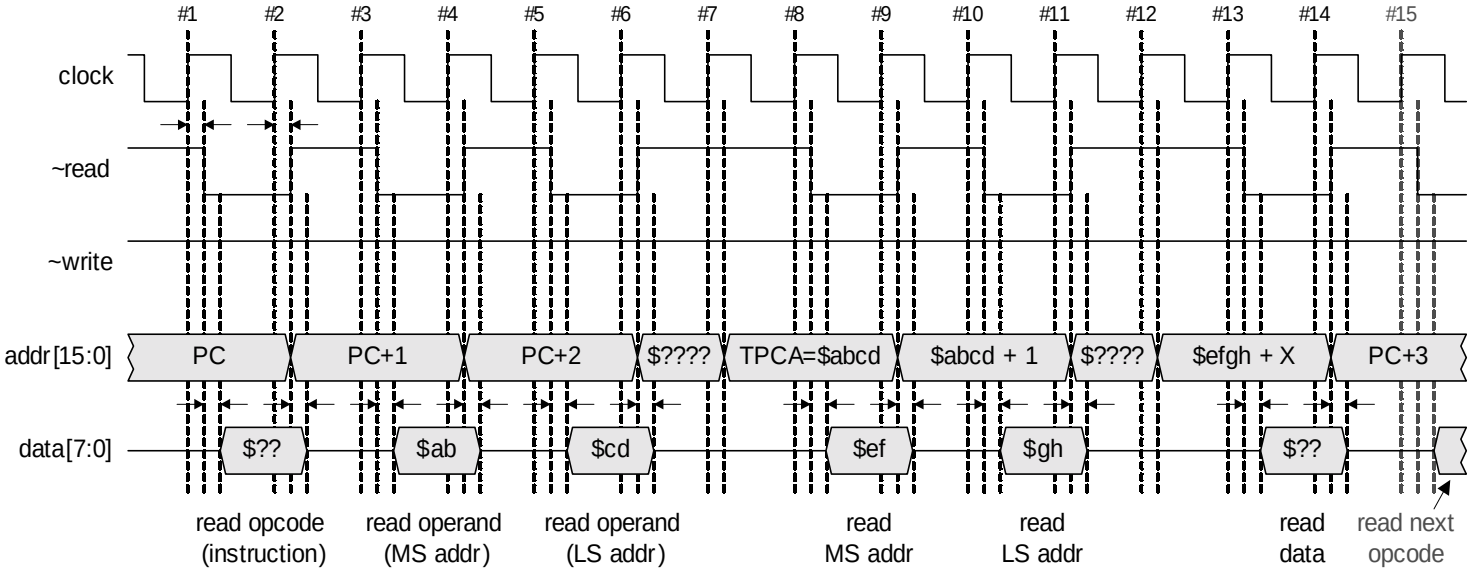


Figure C-43. Timing for indirect post-indexed mode group 1 actions

	Actions for indirect post-indexed mode group 1 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
	The program counter is incremented to contain PC + 3 (note #2).
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The address latch is loaded with the current contents of TPCA ($abcd).
	TPCA is incremented to contain $abcd + 1 (note #3).
Clock #8	The ~read signal goes active.

Clock #9	The ~read signal goes inactive.
	The most-significant indirect address byte (which we've shown as $ef) is loaded into the most-significant byte of temporary addressing register TPCB.
	The address latch is loaded with the current contents of TPCA ($abcd+1).
Clock #10	The ~read signal goes active.
Clock #11	The ~read signal goes inactive.
	The least-significant indirect address byte (which we've shown as $gh) is loaded into the least-significant byte of temporary addressing register TPCB.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #12	The address latch is loaded with the current contents of TPCB ($efgh) plus the contents of the index register (X).
Clock #13	The ~read signal goes active.
Clock #14	The ~read signal goes inactive.
	The data byte is loaded into the accumulator and any appropriate status flags are updated.
	The address latch is loaded with the current contents of the program counter (PC + 3).
	The program counter is incremented to contain PC + 4.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Note #2: The program counter **must** be incremented to contain PC + 3 on clock #5 (the MS byte of TPCA is being loaded on clock #4).

Note #3: As opposed to incrementing TPCA on clock #7 and then loading this new value into the address latch on clock #9, we could leave TPCA alone on clock #7 and simply load the address latch with TPCA+1 on clock #9.

Indirect post-indexed mode: Group 2

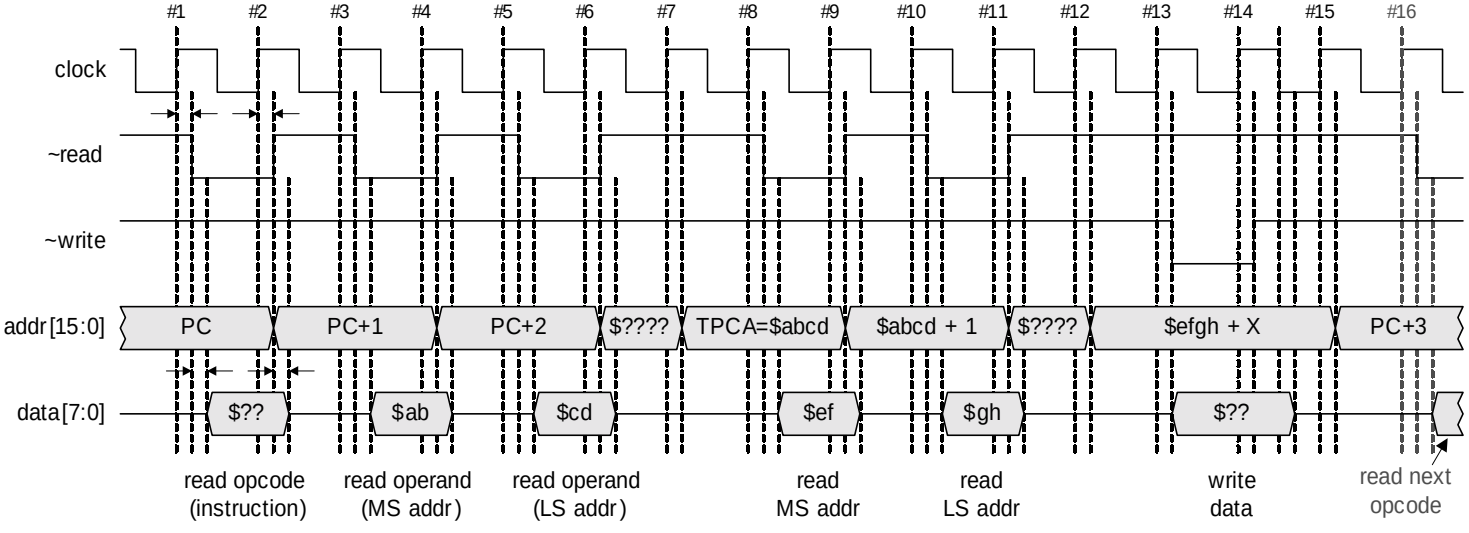


Figure C-44. Timing for indirect post-indexed mode group 2 actions

	Actions for indirect post-indexed mode group 2 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
	The program counter is incremented to contain PC + 3 (note #2).
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The address latch is loaded with the current contents of TPCA ($abcd).
	TPCA is incremented to contain $abcd + 1 (note #3).
Clock #8	The ~read signal goes active.
Clock #9	The ~read signal goes inactive.
	The most-significant indirect address byte (which we've shown as $ef) is loaded into the most-significant byte of temporary addressing register TPCB.
	The address latch is loaded with the current contents of TPCA ($abcd+1).
Clock #10	The ~read signal goes active.
Clock #11	The ~read signal goes inactive.
	The least-significant indirect address byte (which we've shown as $gh) is loaded into the least-significant byte of temporary addressing register TPCB.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #12	The address latch is loaded with the current contents of TPCB ($efgh) plus the contents of the index register (X).
Clock #13	The ~write signal goes active.
	The CPU starts driving the contents of the accumulator onto the data bus.
Clock #14	The ~write signal goes inactive.
	The data byte from the accumulator is loaded into the targeted address.
	The CPU stops driving the data bus on the falling edge of clock #14.
Clock #15	The address latch is loaded with the current contents of the program counter (PC + 3).
	The program counter is incremented to contain PC + 4.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Note #2: The program counter **must** be incremented to contain PC + 3 on clock #5 (the MS byte of TPCA is being loaded on clock #4).

Note #3: As opposed to incrementing TPCA on clock #7 and then loading this new value into the address latch on clock #9, we could leave TPCA alone on clock #7 and simply load the address latch with TPCA+1 on clock #9.

Indirect post-indexed mode: Group 3

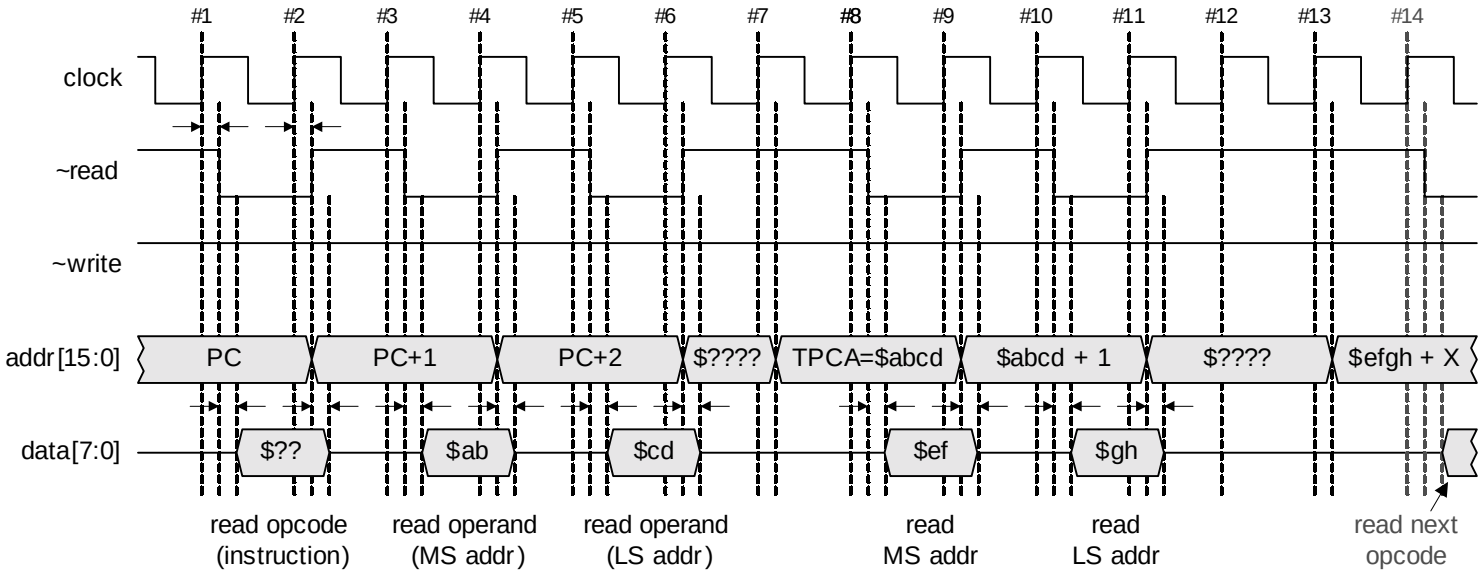


Figure C-45. Timing for indirect post-indexed mode group 3 actions

	Actions for indirect post-indexed mode group 3 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.

Clock #7	The address latch is loaded with the current contents of TPCA ($abcd).
	TPCA is incremented to contain $abcd + 1 (see note #2).
Clock #8	The ~read signal goes active.
Clock #9	The ~read signal goes inactive.
	The most-significant indirect address byte (which we've shown as $ef) is loaded into the most-significant byte of temporary addressing register TPCB.
	The address latch is loaded with the current contents of TPCA ($abcd+1).
Clock #10	The ~read signal goes active.
Clock #11	The ~read signal goes inactive.
	The least-significant indirect address byte (which we've shown as $gh) is loaded into the least-significant byte of temporary addressing register TPCB.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #12	The program counter is loaded with the current contents of TPCB ($efgh) plus the contents of the index register (X). (This doesn't affect the contents of the TPCB or X registers.)
Clock #13	The address latch is loaded with the current contents of the program counter ($efgh + X).
	The program counter is incremented to contain ($efgh + X) + 1.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or #3.

Note #2: As opposed to incrementing TPCA on clock #7 and then loading this new value into the address latch on clock #9, we could leave TPCA alone on clock #7 and simply load the address latch with TPCA+1 on clock #9.

Indirect post-indexed mode: Group 4

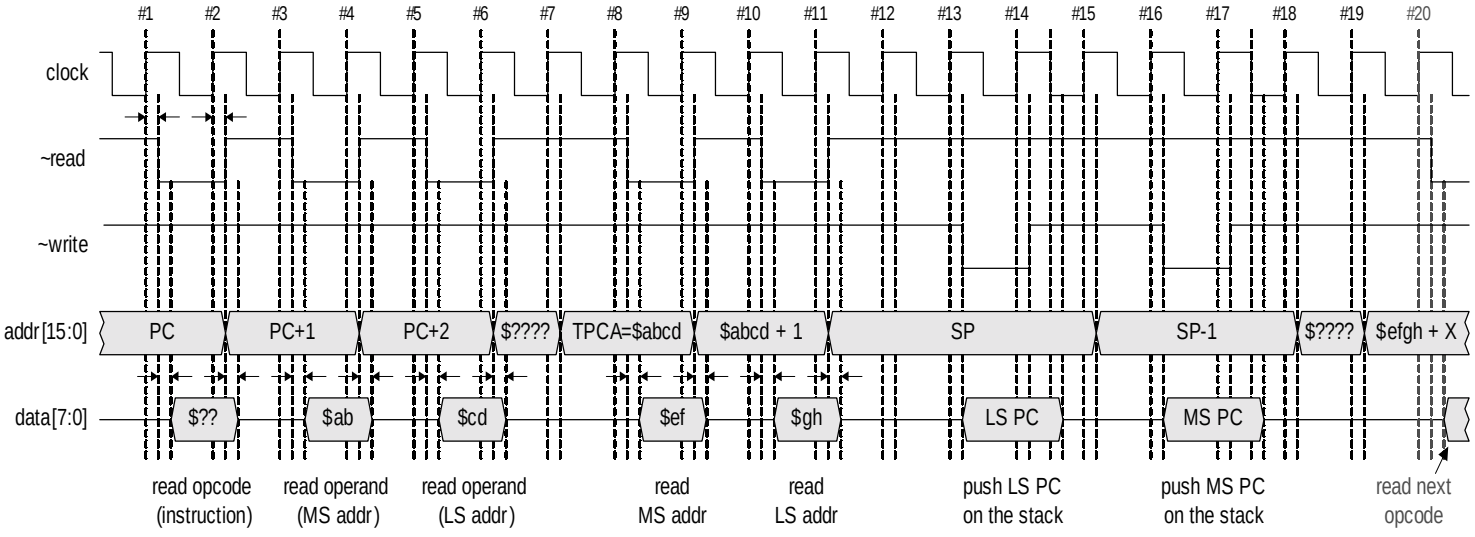


Figure C-46. Timing for indirect post-indexed mode group 4 actions

	Actions for indirect post-indexed mode group 4 instructions
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
	Stack pointer contains a value of SP.
Clock #1	The ~read signal goes active.
Clock #2	The ~read signal goes inactive.
	The new opcode is loaded into the instruction register.
	The address latch is loaded with the current contents of the program counter (PC + 1).
	The program counter is incremented to contain PC + 2 (note #1).
Clock #3	The execution of the instruction commences by ~read going active again.
Clock #4	The ~read signal goes inactive.
	The first operand byte (which we've shown as $ab) is loaded into the most-significant byte of temporary addressing register TPCA.
	The address latch is loaded with the current contents of the program counter (PC + 2).
Clock #5	The ~read signal goes active.
Clock #6	The ~read signal goes inactive.
	The second operand byte (which we've shown as $cd) is loaded into the least-significant byte of temporary addressing register TPCA.
	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
Clock #7	The address latch is loaded with the current contents of TPCA ($abcd).
	TPCA is incremented to contain $abcd + 1 (note #2).
Clock #8	The ~read signal goes active.
Clock #9	The ~read signal goes inactive.
	The most-significant indirect address byte (which we've shown as $ef) is loaded into the most-significant byte of temporary addressing register TPCB.
	The address latch is loaded with the current contents of TPCA ($abcd+1).
Clock #10	The ~read signal goes active.
Clock #11	The ~read signal goes inactive.
	The least-significant indirect address byte (which we've shown as $gh) is loaded into the least-significant byte of temporary addressing register TPCB.
	The address latch is loaded with the current contents of the stack pointer (SP), which is pointing to the first free location on the top of the stack.
Clock #12	The stack pointer is decremented to contain SP -1, which is the new top of the stack (note #3).
Clock #13	The ~write signal goes active.
	The CPU starts driving the least-significant byte of the program counter out onto the data bus.
Clock #14	The ~write signal goes inactive.
	The least-significant byte from the program counter is stored on the top of the stack (at SP).
	The CPU stops driving the data bus on the *falling edge* of clock #14.
Clock #15	The address latch is loaded with the current contents of the stack pointer (SP - 1), which is pointing to the first free location on the top of the stack.
	The stack pointer is decremented to contain SP -2, which is the new top of the stack.

Clock #16	The `~write` signal goes active.
	The CPU starts driving the most-significant byte of the program counter out onto the data bus.
Clock #17	The `~write` signal goes inactive.
	The most-significant byte from the program counter is stored on the top of the stack (at SP - 1).
	The CPU stops driving the data bus on the *falling edge* of clock #17.
Clock #18	It doesn't really matter what's loaded into the address latch at this point because we're not going to use this value.
	The program counter is loaded with the current contents of TPCB ($efgh) plus the contents of the index register (X). (This doesn't affect the contents of the TPCB or X registers.)
Clock #19	The address latch is loaded with the current contents of the program counter ($efgh + X).
	The program counter is incremented to contain ($efgh + X) + 1.

Note #1: The program counter may be incremented to contain PC + 2 on either clock #2 or clock #3.

Note #2: As opposed to incrementing TPCA on clock #7 and then loading this new value into the address latch on clock #9, we could leave TPCA alone on clock #7 and simply load the address latch with TPCA+1 on clock #9.

Note #3: Ideally we would prefer to decrement the stack pointer to contain SP - 1 on clock #11, but this clock is being used to load the LS byte of TPCB, so we have to use an extra clock (#12).

Interrupts

In fact there isn't really an interrupt instruction per se. However, in the same way as a JSR instruction jumps to a subroutine, we might consider an interrupt as being equivalent to an imaginary `JIR` ("*jump to interrupt*") instruction.

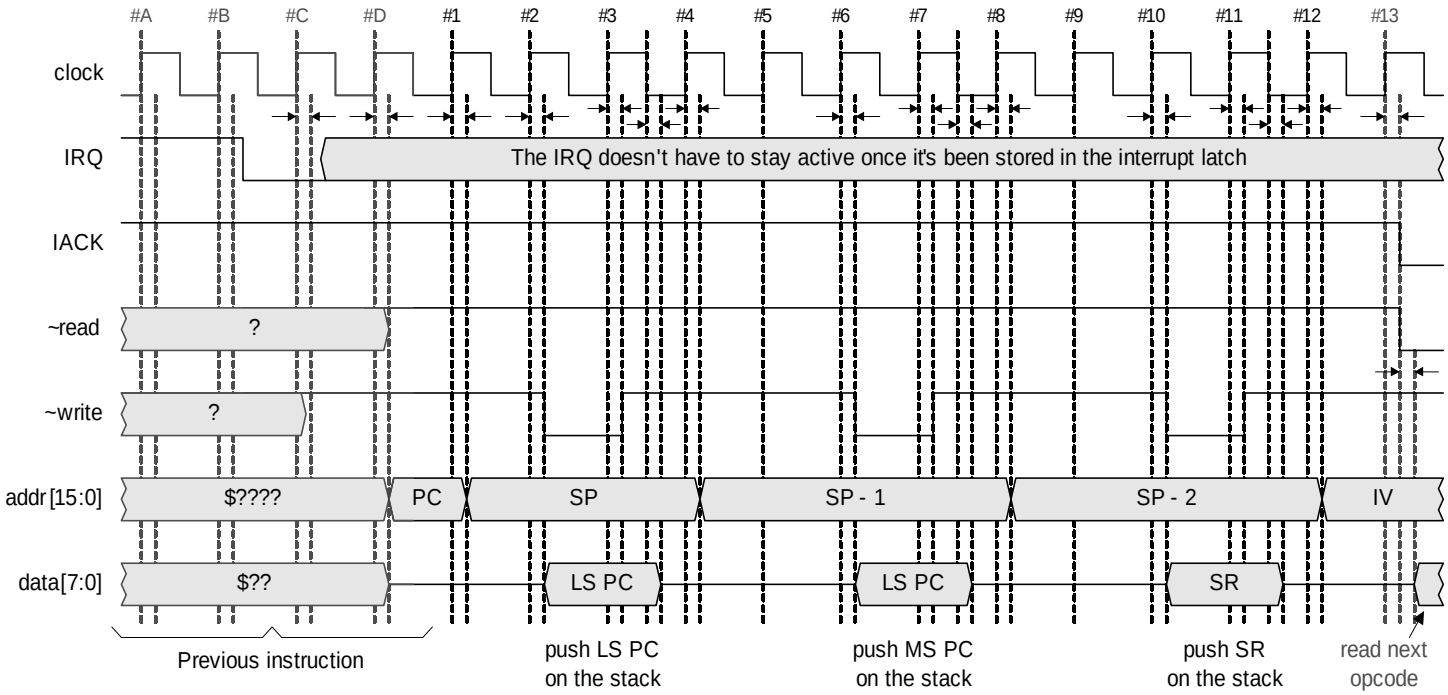


Figure C-47. Timing for interrupt actions

An interrupt is requested by a *falling edge* on the CPU's `IRQ` input. Remember that the CPU will only respond if its interrupt mask (I) flag in the status register is set to logic 1 (the interrupt mask flag is cleared to a logic 0 on a reset or power-on reset). Also, using a `SETIM` ("*set interrupt mask*") instruction to load the interrupt mask with a logic 1 will also clear the CPU's interrupt latch. This means that the CPU will only respond to interrupts that occur after the `SETIM` instruction is executed. Last but not least, before an interrupt can be handled appropriately, the stack pointer must contain a valid address and a `BLDIV` ("*big load interrupt vector*") instruction must have been used to load the CPU's interrupt vector with the address of an interrupt service routine.

An interrupt can occur at any time. For example, in the figure above we're assuming that the interrupt occurs sometime between clocks #B and #C. The falling edge on the `IRQ` signal will be stored in the interrupt latch, and the CPU will continue to execute whatever instruction it's currently working on. When the CPU has finished its current instruction, it first looks to see if the interrupt status flag is enabled (logic 1). If this flag is enabled, the CPU next looks at the interrupt latch to see if an interrupt has occurred, and only then will the CPU begin to process the interrupt request (see also *Chapter 3*). (Note that if the last instruction was a `HALT`, the CPU will start to service the interrupt on the first rising clock edge after the `IRQ` goes active.)

For the purposes of these discussions, we'll assume that when an interrupt occurs, the interrupt mask bit has been loaded with a logic 1, the stack pointer contains a value of SP, and the interrupt vector contains a value of IV. Also, when the interrupt occurs, we're assuming that the

address latch contains a value we'll call PC (which is pointing to the next opcode to be executed), while the program counter will contain a value of PC + 1. This means that when we come to store the return address on the top of the stack, we'll actually have to store the current value in the program counter *minus one*; that is (PC + 1) - 1 = PC ..... Good Grief!

	Actions for an interrupt
Initial	Address latch contains a value of PC.
	Program counter contains a value of PC + 1.
	Stack pointer contains a value of SP.
	Interrupt vector contains a value of IV.
Clock #1	The address latch is loaded with the current contents of the stack pointer (SP), which is pointing to the first free location on the top of the stack.
	The stack pointer is decremented to contain SP – 1; this will be the new top of the stack.
Clock #2	The `~write` signal goes active.
	The CPU starts driving the least-significant byte of the program counter out onto the data bus.
Clock #3	The `~write` signal goes inactive.
	The least-significant byte from the program counter is stored on the top of the stack (at SP).
	The CPU stops driving the data bus on the *falling edge* of clock #3.
Clock #4	The address latch is loaded with the current contents of the stack pointer (SP - 1), which is pointing to the first free location on the top of the stack.
Clock #5	The stack pointer is decremented to contain SP – 2; this will be the new top of the stack.
Clock #6	The `~write` signal goes active.
	The CPU starts driving the most-significant byte of the program counter out onto the data bus.
Clock #7	The `~write` signal goes inactive.
	The most-significant byte from the program counter is stored on the top of the stack (at SP - 1).
	The CPU stops driving the data bus on the *falling edge* of clock #7.
Clock #8	The address latch is loaded with the current contents of the stack pointer (SP - 2), which is pointing to the first free location on the top of the stack.
Clock #9	The stack pointer is decremented to contain SP – 3; this will be the new top of the stack.
Clock #10	The `~write` signal goes active.
	The CPU starts driving the contents of the status register (SR) out onto the data bus.
Clock #11	The `~write` signal goes inactive.
	The most-significant byte from the program counter is stored on the top of the stack (at SP - 2).
	The CPU stops driving the data bus on the *falling edge* of clock #11.
Clock #12	The address latch is loaded with the current contents of the interrupt vector (IV).
	The program counter is loaded with IV + 1.

Clock cycle summary

	imp		imm		abs		abs-x		ind		x-ind		ind-x	
	op	#	op	#	op	#	op	#	op	#	op	#	op	#
ADD			$10	5	$11	10	$12	10						
ADDC			$18	5	$19	10	$1A	10						
AND			$30	5	$31	10	$32	10						
BLDIV			$F0	7	$F1	11								
BLDSP			$50	7	$51	11								
BLDX			$A0	7	$A1	11								
BSTSP					$59	13								
BSTX					$A9	13								
CLRIM	$09	3												
CMPA			$60	5	$61	10	$62	10						
DECA	$81	3												
DECX	$83	3												
HALT	$01	3												
INCA	$80	3												
INCX	$82	3												
JC					$E1	*7								
JMP					$C1	7	$C2	8	$C3	12	$C4	12	$C5	13
JN					$D9	*7								
JNC					$E6	*7								
JNN					$DE	*7								
JNO					$EE	*7								
JNZ					$D6	*7								

* The conditional jump instructions (JC, JNC, JN, JNN, ...) require 7 clock cycles if the test passes, but only 4 clock cycles if the test fails.

Table C-1. Clock cycle summary (continued on next page)

Legend

op = Opcode
$ = Hexadecimal value
# = Number of clocks required to execute the instruction

Addressing Modes

imp = Implied
imm = Immediate
abs = Absolute
abs-x = Indexed
ind = Indirect
x-ind = Pre-indexed indirect
ind-x = Indirect post-indexed

	imp		imm		abs		abs-x		ind		x-ind		ind-x	
	op	#	op	#	op	#	op	#	op	#	op	#	op	#
JO					$E9	*7								
JSR					$C9	13	$CA	14	$CB	18	$CC	18	$CD	19
JZ					$D1	*7								
LDA			$90	4	$91	9	$92	9	$93	14	$94	14	$95	14
NOP	$00	3												
OR			$38	5	$39	10	$3A	10						
POPA	$B0	5												
POPSR	$B1	5												
PUSHA	$B2	6												
PUSHSR	$B3	6												
ROLC	$78	3												
RORC	$79	3												
RTI	$C7	10												
RTS	$CF	8												
SETIM	$08	3												
SHL	$70	3												
SHR	$71	3												
STA					$99	10	$9A	10	$9B	15	$9C	15	$9D	15
SUB			$20	5	$21	10	$22	10						
SUBC			$28	5	$29	10	$2A	10						
XOR			$40	5	$41	10	$42	10						

* The conditional jump instructions (JC, JNC, JN, JNN, ...) require 7 clock cycles if the test passes, but only 4 clock cycles if the test fails.

Table C-1. Clock cycle summary (continued from previous page)

Legend

op = Opcode
$ = Hexadecimal value
# = Number of clocks required to execute the instruction

Addressing Modes

imp = Implied
imm = Immediate
abs = Absolute
abs-x = Indexed
ind = Indirect
x-ind = Pre-indexed indirect
ind-x = Indirect post-indexed

Appendix D

Assembly Language Overview

A positive plethora of programming languages

One way in which we can feed programs and data to a computer is as a series of numerical values; for example:

```
$90 $10 $99 $F0 $31 $A0 $00 $00
$92 $40 $15 $D1 $00 $00 $99 $F0
```

This style of representation is known as *machine language* or *machine code*, because this is the form that is directly understood and executed by the machine (computer). Sad to relate, machine code is a pain, and working at this level is only slightly more fun than banging your head against a wall.

Apart from anything else, working with machine code is time-consuming and prone to error. Humans tend to find it difficult to conceptualize things purely in terms of numbers; we much prefer to describe things using symbolic representations consisting of words and symbols. Thus, we would ideally prefer to describe our programs at a high level of abstraction (along the lines of "*First do this, then do that, then do*....") and then translate them into machine code for the computer to work with (Figure D-1).

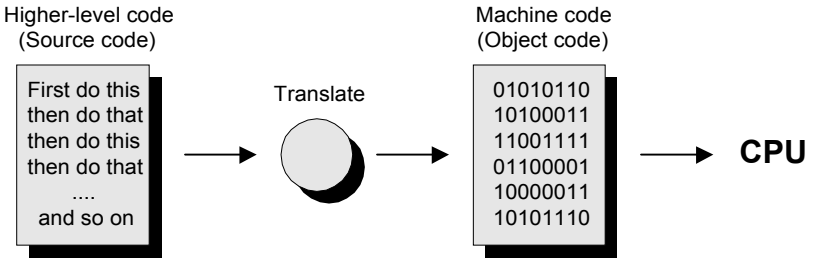


Figure D-1. It's preferable to work at a higher level than machine code

1GLs: There are literally thousands of different general-purpose and special-purpose computer programming languages. As we previously noted, at the bottom of the pile (in terms of abstraction) we have machine code. Each type of computer (CPU) has its own machine code, and these languages are collectively referred to as *first generation languages*, or 1GLs for short (Figure D-2).

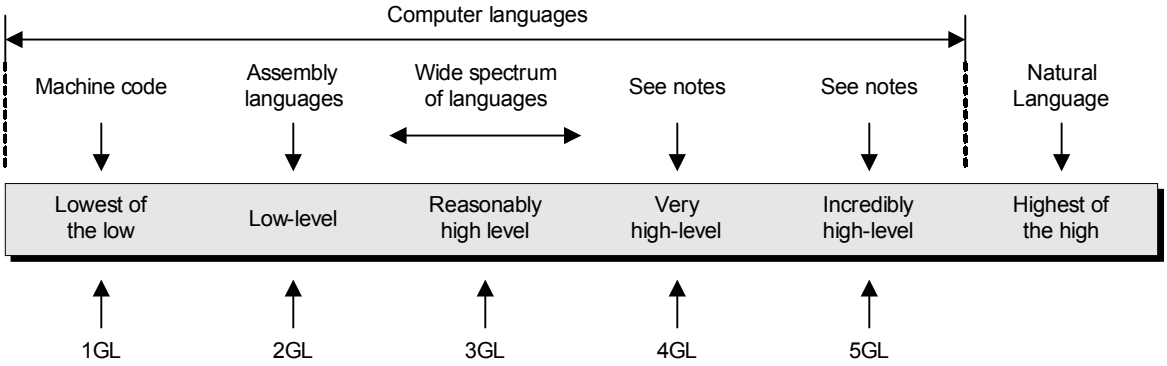


Figure D-2. Varying levels of language abstraction

Natural languages: The highest level of abstraction we might consider here would be a *natural language*, such as written English. Unfortunately, the present state of our technology does not allow us to use a natural language for programming purposes, because such languages contain numerous ambiguities and logical inadequacies. For this reason we use symbolic languages, which are similar in concept to natural languages, but which are highly formalized with regard to their *syntax* and *semantics*. In this context, the term "syntax" refers to the grammar of the language, such as the ordering of the words and symbols in relation to each other, while the term "semantics" refers to the underlying meaning of the words and symbols and the relationships between the things they denote -- phew!

2GLs: One step up the evolutionary ladder from machine code is *assembly language* (the focus of this appendix). In this case, mnemonics are assigned to the various instructions and the programmer can also declare and use labels to make things easier to understand. For example, a typical assembly language instruction might look like the following:

```
STORE: STA [MAINDISP]
```

Most assembly languages are designed to support only one computer or a family of related computers. A special program called an *assembler* is used to translate the assembly source code into its corresponding machine code. In some respects, assembly languages – which are referred to as *second generation languages (2GLs)* – may be considered only a small improvement over machine codes, but in other respects we may regard them as being a quantum leap.

3GLs: The vast majority of today's programming languages are classed as *third generation languages (3GLs)*. These medium to high-level languages are designed to be relatively easy for a human to understand (although they still require a lot of programming knowledge) and to be portable across multiple computers. A representative statement from this class of language might look like the following:

```
for i = 1 to 10 print (i);
```

Some examples of this general class of language are Ada, Algol, BASIC, C, C++, COBOL, Forth, FORTRAN, Java, Lisp, Pascal, Perl, and Prolog (to name but a few). Depending on the language, a special program called a *compiler* or an *interpreter* is used to translate the source code into corresponding machine code.

4GLs: A fourth-generation language (4GL) is designed to be closer to natural language than a third-generation language. These languages are intended to reduce the time, cost, and effort associated with programming. A good example would be a language that is designed to interface with databases so as to ease the task of locating and extracting data. A statement from such a language might look somewhat like the following:

```
Find customers whose current purchases exceed $1000
```

5GLs: This is where things start to get a little "fluffy" around the edges. Some folks regard *fifth-generation languages (5GLs)* as comprising visual or graphical programming environments that are used to generate 3GL or 4GL source code (this code will subsequently be translated into machine code using appropriate 3GL or 4GL compilers or interpreters). However, other pundits

regard 5GLs as being textual languages that are closer to natural language than 4GLs. In this latter scenario, these languages, which are often associated with *artificial intelligence (AI)* research, are designed to make the computer determine the solution to a problem (as opposed to 3GLs and 4GLS which are designed to create focused programs to address specific problems). To put this another way, programmers using a fifth-generation language need only worry about what problems need to be solved, what constraints need to be specified, and what conditions need to be met without concerning themselves as to the nitty-gritty algorithms actually required to solve the problems.

The *DIY Calculator's* Assembly language

As we previously noted, assembly language is much easier to understand and work with than machine code. As with any language (natural or computer), however, it's necessary to understand the vocabulary and the linguistic rules. Thus, the remainder of this appendix is devoted to introducing the *DIY Calculator's* assembly language (where any particularly pertinent points will be proffered with a pointing finger ☞ character).[1]

Statements

Just as one of the fundamental building blocks of a natural language is a group of words in the form of a *sentence*, the equivalent construct in a computer language is called a *statement*. For the purposes of the *DIY Calculator's* assembly language, we may regard a statement as encompassing a single thought or idea. An assembly source file is composed of a series of these statements, each of which typically consists of four fields (Figure D-3).

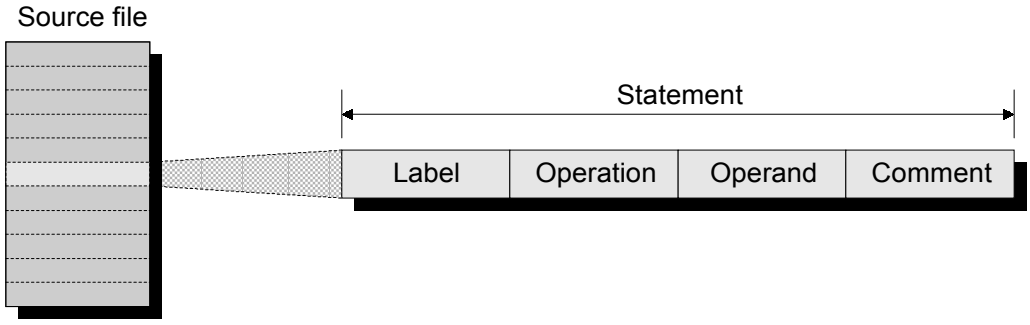


Figure D-3. Statements typically consist of four fields

☞ Many computer languages (including some assembly languages) allow statements to span multiple lines, in which case they would be terminated by a special character such as a semicolon. In the case of our simple language, however, a statement may only occupy a single line and is terminated by a carriage return `<cr>` character (which equates to the key marked "Enter" or "Return" on your keyboard).

☞ The majority of the early assembly languages had extremely restrictive rules, such as specifying the columns in which each field must commence. By comparison, our syntax is relatively free-format – you can use as many whitespace (`<tab>` and `<space>`) characters and be as messy as you wish. Having said this, we strongly recommend that you keep your source code as neat and tidy as possible. You can follow the style we use in our examples or feel free to develop your

[1] A formal syntax definition of the *DIY Calculator's* assembly language is provided in *Appendix E*.

own; but whatever you do, try to be as consistent as possible – you'll find that consistency pays dividends in the long run when you return to blow the dust off a neglected source code file some time in the future.

☞ Many early assemblers allowed source files to be created using only uppercase characters. In the case of our language you may use uppercase, lowercase, or a mixture of both. However, for the purposes of its machinations, our assembler internally converts everything to uppercase, which means that it will consider labels such as `fred`, `Fred`, `FrEd` and `FRED` to be identical.

Comments and blank lines

Generally speaking it's a good idea to liberally sprinkle your assembly source with comments, and to distinguish logically distinct portions of your program with blank lines. Hopefully, this will mean that when you return to your source code in the future, you will actually have a clue as to what's going on. Some authorities hold that it isn't possible to have too many comments, but if you're too extravagant it can be difficult to locate your program amongst the commentary. Ultimately it's a matter of personal choice — it's up to you to find a style you like and stick to it (Figure D-4).

```
# This is a program wot wos wrote on 3rd day of Grunge in
# the year of the lesser-spotted Mugwump (if we're lucky
# it will flash our 6-bit LED display).

LEDS: .EQU  $F032   # $F032 is the address of the output
                    # port driving the 6-bit LED display

      .ORG  $4000   # Start of program is address $4000
       LDA  $03     # Initialize the ACC
```

Figure D-4. Comments and blank lines

☞ A comment can occur anywhere on a source line and commences with a hash '#' character (this character is also often referred to as a *number sign*, *pound sign*, or a *sharp*). The hash character may be followed by any printable text characters (including spaces, tabs, and other hashes). A comment is terminated by a carriage return `<cr>` character (once again, this equates to the key marked "Enter" or "Return" on your keyboard).

Label names

Our assembly language supports two kinds of labels, which are referred to as *constant* and *address* labels. We will examine the differences in the way these labels are used in a little while; for the moment we need only note that they have identical naming conventions (Figure D-5).

```
LEDS: .EQU  $F032   # "LEDS" is a constant label
      .ORG  $4000   # Start of program is address $4000
       LDA  $03     # Initialize the ACC
LOOP:  STA [LEDS]   # "LOOP" is an address label
```

Figure D-5. Label names

- Both types of label can consist of a mixture of alphabetic, numeric, and underscore '_' characters, but the first character *must* be either an alpha or an underscore (not a number).
- When a label is declared, it is terminated with a colon ':' character, but this character doesn't form part of the label's name, and is not used thereafter.
- The maximum length of a label name is eight characters. This includes any underscores, but excludes the colon ':' character used to terminate the label.
- Labels can include both uppercase and lowercase characters, but the assembler internally converts everything to uppercase, so it will consider labels such as `fred`, `Fred`, `FrEd` and `FRED` to be identical.

```
 RET_ADDR: # Legal and useful, because just by reading it
           # you get the picture "return address"
  BIG_BOY: # Legal, but not very indicative of its function
   _LOOP1: # Legal, but we recommend that you don't start
           # your labels with underscore characters
 1ST_LOOP: # Illegal, can't start with a numeric character
LOOP_FIVE: # Illegal, can't have more than 8 characters
      LDA: # Illegal, this is one of our reserved words
```

Figure D-6. Legal and illegal label names

- In this last example, note that labels cannot be the same as any of our reserved words. These include directive mnemonics, instruction mnemonics, and also the special reserved word "X" (a complete list of reserved words is provided in *Appendix E*).

The .ORG and .END directives

In addition to standard instructions, our assembly language supports special instructions to direct the assembler to do certain things. These special instructions may be referred to as *pseudo-instructions* or *directives* (because they direct the assembler). Every program must contain at least two directives: the `.ORG` and the `.END` (Figure D-7).

```
# Any declaration statements come here (before the .ORG)

      .ORG  $4000   # Start of program is address $4000

# The statemants forming the body of the program go here

      .END          # Tells the assembler to stop here

# Anything appearing after the .END will be ignored
```

Figure D-7. The `.ORG` and `.END` directives

- The .ORG directive instructs the assembler as to the origin of the program; that is, the memory location into which it should place the program's first byte. Thus, this directive must have an operand in the form of a number (as shown here) or a constant label from a declaration statement (as described below).
- The .END directive simply informs the assembler that it's reached the end of the program; thus, this directive does not require any operand.
- Neither the .ORG or .END directives are allowed to have labels associated with them.

The .EQU directive

Another directive is the .EQU, which stands for "*equates to.*" This directive appears in *declaration statements*, which are used to declare constant values for later use (Figure D-8).

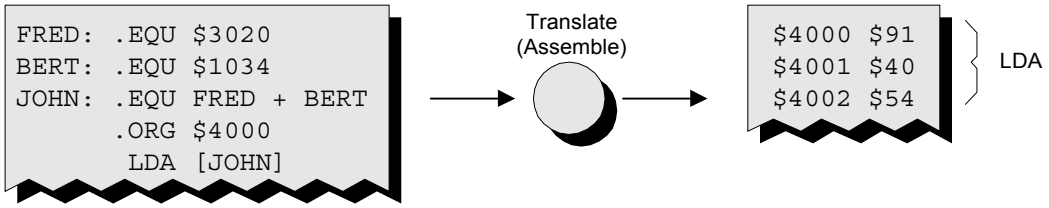


Figure D-8: The .EQU directive

- If declaration statements are used, they must appear before the .ORG directive at the beginning of the program.
- Each .EQU directive must have a label assigned to it. These constant labels may be used in the body of the program instead of literal (numerical) values. When the assembler is assembling the program, it will automatically substitute any constant labels in the body of the program with their numerical equivalents. For example, the assembler will automatically substitute JOHN in the LDA instruction in the above example for the numerical value $4054.
- With regard to the previous point, note that constant labels are used only by the assembler, and they don't appear in (or occupy any space in) the resulting machine code. Also note that constant labels can be used to represent addresses, data, or both (this will be discussed in more detail a little later).
- The assignment to an .EQU statement may be presented in the form of an expression, as is illustrated by the assignment to JOHN in the above example (expressions are introduced in more detail below). However, it is important to note that forward-referencing is not allowed; that is, JOHN is allowed to reference FRED and BERT, because they've already been declared, but BERT isn't allowed to reference JOHN, while FRED isn't allowed to reference either BERT or JOHN.

The .BYTE, .2BYTE, and .4BYTE directives

The .BYTE, .2BYTE, and .4BYTE directives are used in *reserve statements* to set aside (reserve) memory locations for later use. Not surprisingly, the .BYTE directive reserves a single byte, the .2BYTE directive reserves two bytes, and the .4BYTE directive reserves ..... well, we'll leave that as an exercise for the reader (Figure D-9)

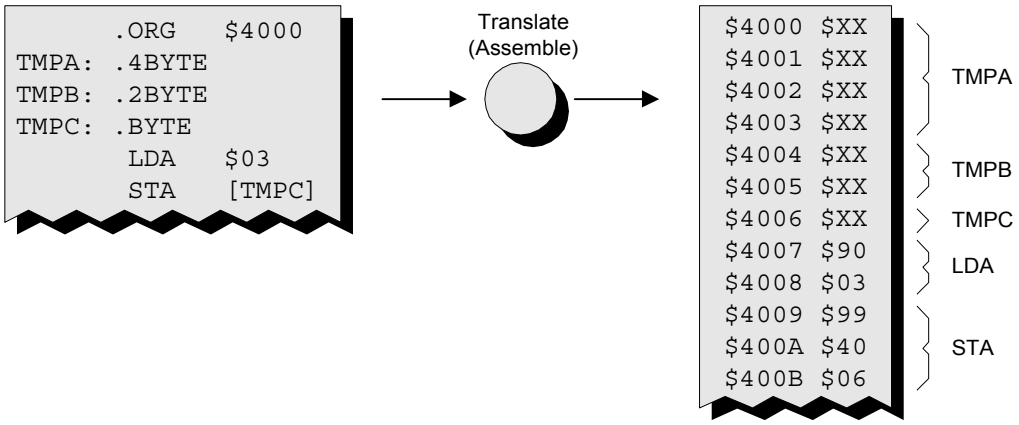


Figure D-9. The `.BYTE`, `.2BYTE`, and `.4BYTE` directives

- If reserve statements are used, they *must* appear in the body of the program; that is, between the `.ORG` and `.END` directives. Purely for the purposes of this example, we've shown the reserve statements as appearing immediately after the `.ORG` directive. However, although this is perfectly legal, it means that we can't run the program from address $4000, but instead have to remember to run from address $4007. Thus, in a real program, it would be more common to find reserve statements hanging around toward the end of the program instead of the beginning.
- Each reserve statement may have an optional label assigned to it. These address labels may be used in the body of the program instead of literal (numerical) values. When the assembler is assembling the program, it will automatically substitute numerical equivalents for any address labels in the body of the program. For example, the assembler will automatically substitute `TMPC` in the `STA` instruction for the address $4006, which is the location the assembler decided to reserve for this byte.
- If a label is used with a `.2BYTE` or a `.4BYTE` directive, then the address the assembler associates with that label will correspond to the first byte of that field. Thus, in the case of this example, the assembler associates the labels `TMPA`, `TMPB`, and `TMPC` with addresses $4000, $4004, and $4006, respectively.
- With regard to the previous points, labels are optional with these directives, because we may not always require the ability to individually reference every location set aside by a reserve statement. This is due to the fact that we can reference one location as an offset from another location (Figure D-10).

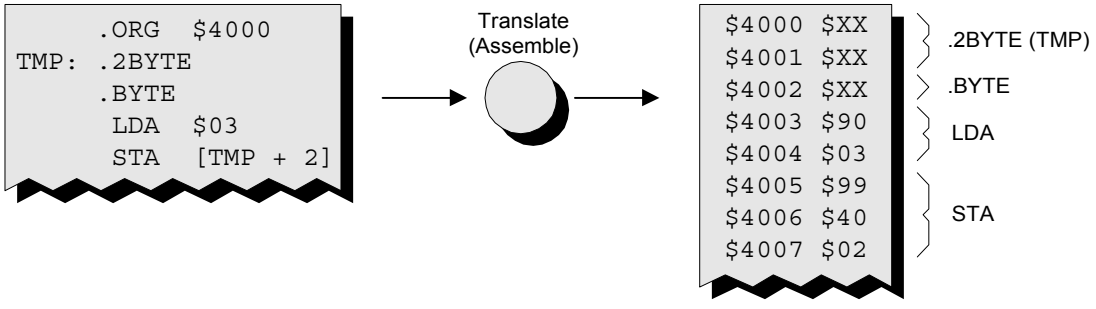


Figure D-10. Referencing one location as an offset from another location

In the example shown in Figure D-10, we first reserve a 2-byte field using a `.2BYTE` directive with a label of `TMP`, and we immediately follow this by reserving a 1-byte field using a `.BYTE` directive without a label. From our previous discussions, we know that the assembler will associate the label `TMP` with the address of the first location in the 2-byte field (which happens to be address $4000). Thus, even though the 1-byte field doesn't have a label of its own, we can still reference it using "`TMP+2`".

When writing a program, it is often necessary to reserve a number of consecutive memory locations. As an alternative to painstakingly reserving locations individually, the `.BYTE`, `.2BYTE`, and `.4BYTE` directives support an optional operand in the form "`*n`", where '`n`' is any expression that resolves into a positive integer. For example, let's suppose that we wish to reserve three 1-byte fields and two 2-byte fields (Figure D-11).

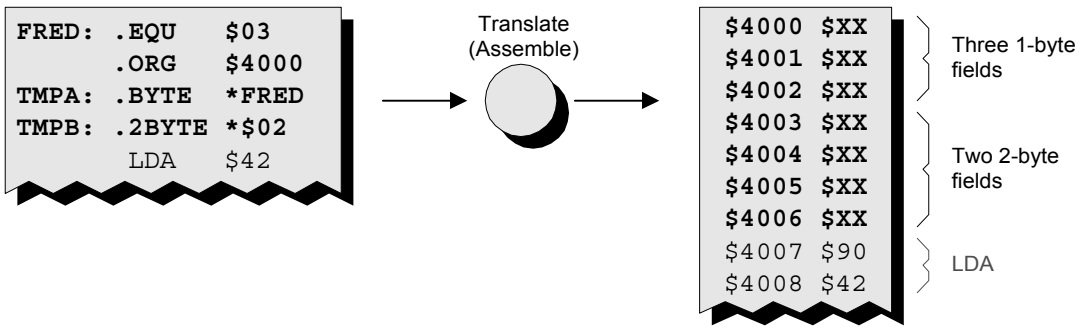


Figure D-11. Reserving multiple locations

Purely for the purpose of these discussions, we start by assigning the numerical value $03 to the constant label `FRED`. Thus, when the assembler comes to consider the reserve statement associated with the address label `TMPA`, it will replace the "`*FRED`" with "`*$03`" and reserve three 1-byte fields. Similarly, when the assembler comes to consider the reserve statement associated with `TMPB`, it will understand that we're instructing it to reserve two 2-byte fields (it's just that we're being a bit more explicit in this case).

As usual, the labels will be associated with the first byte in their corresponding groups; thus, the assembler will associate `TMPA` with address $4000 and `TMPB` with address $4003. Also, although we've not illustrated this point here, note that the '`n`' in "`*n`" can be a full-blown expression. For example, assuming that we're still using `FRED` from the previous example, the statement "`.BYTE *(FRED + 5)`" would reserve eight bytes of memory (expressions are discussed in more detail later in this appendix).

All of the machine code resulting from the reserve statement examples we've considered thus far has been shown as containing $XX values, which indicate that we have not defined the contents of these locations. As it happens, our assembler will automatically cause these undefined locations to contain zero values, but that's beside the point. In fact the *DIY Calculator's* assembler allows us to associate values with these locations as a comma-separated list (Figure D-12).

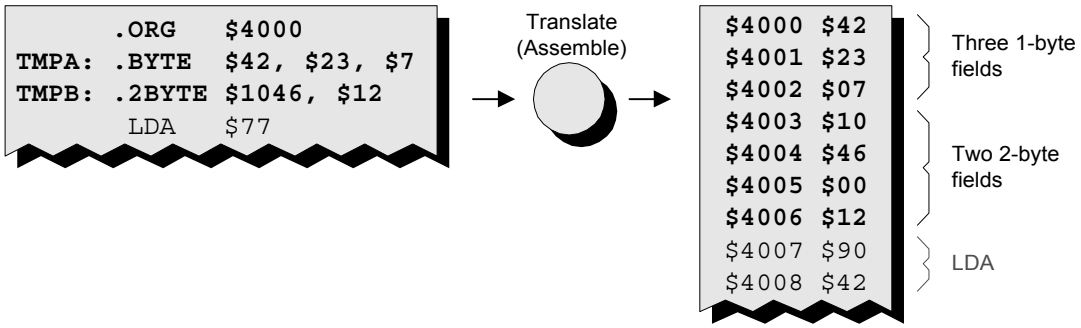


Figure D-12. Assigning values in reserve statements

Once again, the labels will be associated with the first byte in their corresponding groups; thus, the assembler will associate `TMPA` with address $4000 and `TMPB` with address $4003. Observe that, as we promised, the assembler accepts the values in the comma separated lists and inserts them into the appropriate locations. Note that the assembler will automatically zero-fill numbers to the requisite size, so the $7 in this example is automatically coerced to $07 because it's being assigned to a 1-byte field, while the $12 is automatically coerced to $0012 because it's being assigned to a 2-byte field (we'll return to consider this aspect of things in more detail in a little while).

Also note that any numbers assigned to multi-byte fields associated with `.2BYTE` and `.4BYTE` directives will be stored with their most-significant byte first. This style of storage is referred to as *big-endian* because we're storing the numbers "big-end-first". (Some computers store multi-byte values using a *little-endian* technique, in which the least-significant byte is stored first. These terms were derived from the part of the tale in *Gulliver's Travels* (written by Jonathan Swift in 1726) whereby two countries go to war over which end of a hard-boiled egg should be eaten first – the big end or the little end!) Finally, although we've not illustrated this point here, each of the values in these comma-separated lists could be a full-blown expression (expressions are discussed in more detail later in this appendix).

Literals (binary, decimal, and hexadecimal values)

Thus far, all of our examples have employed hexadecimal numbers, because this is the base we've used predominantly throughout this book. However, it is sometimes more appropriate to use numbers with other bases to better clarify the intent of the program. For this reason, the *DIY Calculator's* assembly language supports binary, decimal, and hexadecimal values (Figure D-13).

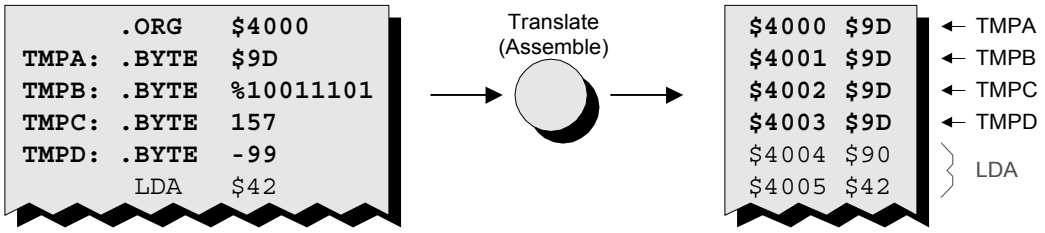


Figure D-13. Binary, decimal, and hexadecimal values

- Binary values are prefixed with a '%' character; hexadecimal values are prefixed with a '$' character; decimal values have no prefix at all.
- Space characters are not permitted between '%' or '$' characters and their respective numbers. Also, decimal values are not permitted to contain commas or any other characters, so 32565 is legal, but 32,565 is not.

The example shown in Figure D-13 commences by reserving four 1-byte fields called `TMPA`, `TMPB`, `TMPC`, and `TMPD`, and assigning them values in hexadecimal, binary, decimal, and decimal (again), respectively. You will note that all of these values actually result in identical bit-patterns when they are translated into machine code (which is, of course, why we selected them in the first place). Note especially the negative value assigned to `TMPD`, where the unary minus operator instructs the assembler to take the twos complement of 99, resulting in the same value as all of the other 1-byte fields.

The reason why we emphasized "1-byte fields" in the preceding paragraph is that the assembler actually represents every number as a 4-byte field internally, and it only considers them as being 1-byte or 2-byte fields when it comes to assign them to reserve statements or to use them as operands for instructions (Figure D-14).

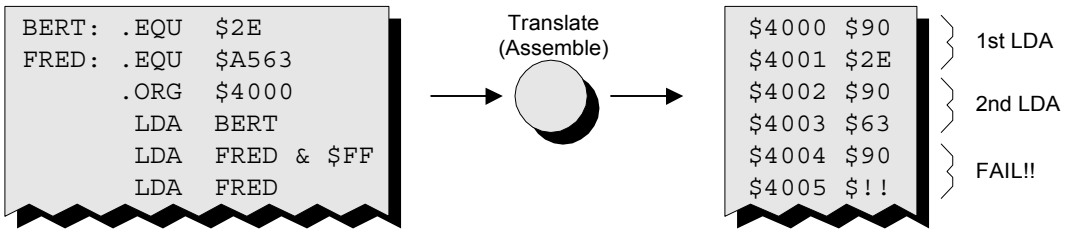


Figure D-14. The assembler checks the size of the target destination

In this case, the first `LDA` instruction passes the assembler's scrutiny without any problems, because the constant `BERT` being loaded into the accumulator was only assigned an 8-bit value, which has no trouble fitting into the CPU's 8-bit accumulator. The second `LDA` instruction is more interesting, because `FRED` was originally assigned a 16-bit value. However, this is an example of an expression using the logical AND operator '&', in which the $FF is used as a mask to extract `FRED's` least-significant byte (this is discussed in more detail below). Finally, the third `LDA` would cause the assembler to issue an error message, because it can't squeeze the un-masked 16-bit value assigned to `FRED` into the CPU's 8-bit accumulator.

The concept of expressions will be introduced shortly, but as we've opened this can of worms here, it behooves us to explain why the second `LDA` in the above example passed muster. Remember that the assembler internally regards every number as a 4-byte field, and it's only when it comes to assign a value to a target destination that it checks to see whether the value will fit. Also, don't forget that *we're* the ones who are defining what *our* assembly language looks like, and we've decided that our language should support simple expressions. So in the case of the second `LDA` instruction, we've decided that when our assembler sees an expression like "`FRED & $FF`", it will perform a logical AND operation between the two values (Figure D-15).

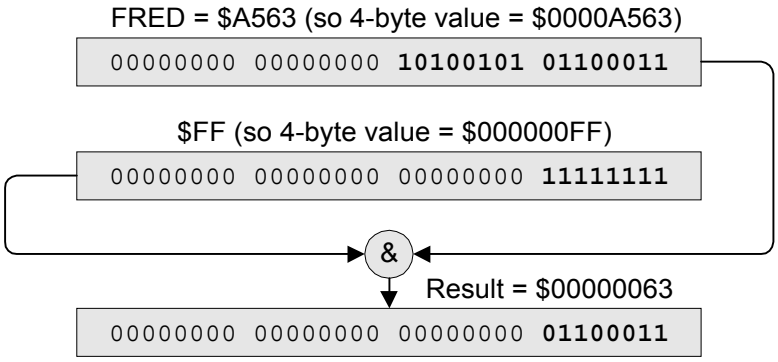


Figure D-15: Pictorial representation of the assembler performing an internal logical AND function on two 4-byte values

First, the 2-byte value assigned to `FRED` is zero-filled by the assembler to boost it up to its 4-byte value of $0000A563, while the $FF value is zero-filled to its 4-byte value of $000000FF. The logical AND operator ('&' in our assembly language) is a bit-wise operator (it performs its actions on a bit-by-bit basis) that functions in a similar manner to the logical AND in the *DIY Calculator's* instruction set. To put this another way, the least-significant bit of the first number is AND-ed with the least-significant bit of the second number to give the least-significant bit of the result, and so on for all of the other bits.

Now, the way in which an AND works is that its output is only logic 1 if both of its the inputs are logic 1, so the three most-significant bytes of the result are forced to zero by the six 0's in the $000000FF value. The outcome of all of this is that only the least-significant eight bits of the result contain a value, which will therefore fit into our 8-bit accumulator.

Expressions

Ah, so finally we arrive at the topic of expressions, which can be exceptionally useful as long as we quickly establish who's in charge ..... us (hopefully) or the expressions (if we turn our backs on them). The trick is to look them straight in the eye and show no fear! First of all, we have decided that our assembly language should support the following selection of *binary operators*:

Arithmetic:	+	Add	**Logical:**	&	AND
	–	Subtract		|	OR
	*	Multiply		^	Exclusive-OR
	/	Divide			

- We call these "binary operators," because each of them requires two operands to work with; for example, in the case of the expression "6 + 3" we see two operands (numbers), one on either side of the operator.
- The operands in expressions can be a mixture of numbers (binary, decimal, and hexadecimal) and labels; for example, assuming that `BERT` has been declared as a label, the expression "`BERT & %11001100`" is perfectly acceptable.
- Our language only supports integer expressions, which are expressions involving whole numbers. Thus, "`BERT * 3`" is legal, while "`BERT * 3.5`" is not.

- Any remainder from a division operation will be automatically discarded by the assembler without any error messages being issued. Similarly, the assembler will not report any errors if the actions from addition, subtraction, and multiplication operations overflow or underflow the 4-byte fields it uses internally to store the results. Having said this, the assembler will make its feelings known in no uncertain terms should you try to divide anything by zero.

- As long as they fit on a single line of source code, expressions can be of arbitrary length and employ any mixture of arithmetic and logical operators. For example, the following is a legal expression (assuming that `BERT`, `JOHN`, and `HARRY` are labels that have been declared elsewhere in the program):

```
BERT ^ 2 * JOHN & $A5 / HARRY
```

- With regard to the previous point, all of our binary operators have equal precedence, and expressions are evaluated from left to right. What does this mean? Well, consider the following example:

At school we're taught that:	6 * 3 + 5 * 4 = 38
But, in our assembly language:	6 * 3 + 5 * 4 = 92

Eeeek! How can this be? Well, at school we are taught that multiplication and division have a higher precedence (are more powerful) than addition or subtraction. This means that we usually evaluate expressions by first performing any multiplications and divisions and then performing any additions and subtractions. Thus, in the first portion of the above example, we'd multiply 6 by 3 to get 18, multiply 5 by 4 to get 20, then add the 18 and 20 to generate the final result of 38.

Had we so wished, we could easily have decided to make our language (and therefore our assembler) work in just this way. However, we decided that you need to broaden your horizons, because it's common for simple programs (like our assembler) to treat all of the binary operators as having equal precedence (being equally powerful) and, as we already noted, to evaluate expressions from left to right. This means that in the case of our language, we would solve this expression by multiplying 6 by 3 to get 18, then adding 5 to get 23, and then multiplying by 4 to generate a final result of 92.

Although this may seem to be a pain, it's really just another way of looking at the world and you'll soon learn to get used to it – let's face it, what other choice do you have? Actually, there are options open to you, because you can use parenthesis to force your expressions to evaluate in any order you like; for example, if you were to write your expression as "(6 * 3) + (5 * 4)", then the assembler will evaluate the contents of any parenthesis before moving on to the rest of the expression.

Note that it's possible to nest parenthesis within each other, but that each '(' must have a corresponding ')'. For example, in addition to being legal, "(((6 * 3) + 5) * 4)" would force the assembler to evaluate the expression in exactly the same order as if the parenthesis weren't there at all (from left to right), while "(6 * (3 + (5 * 4)))" would force the assembler to evaluate the expression in reverse order (from right to left).

Generally speaking you'll find that most of the expressions you create will be short, sweet, and to the point; but even simple expressions can be incredibly useful. Expressions can be employed in a variety of roles, including assignments to `.EQU` statements, modifying data values, and calculating addresses. Consider an example involving expressions in `.EQU` statements (Figure D-16).

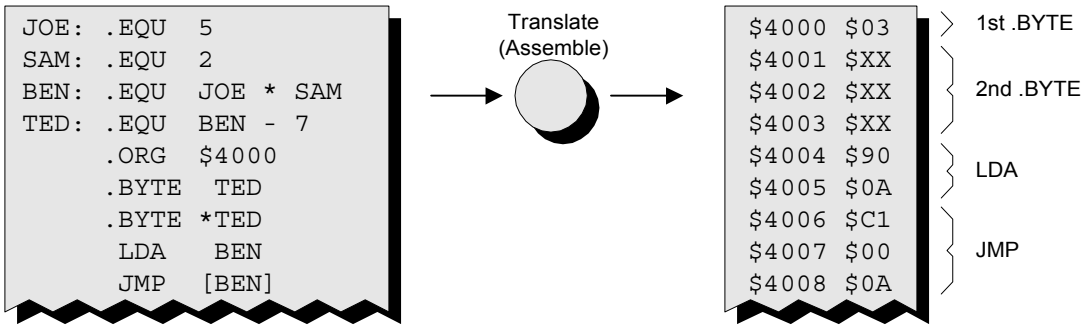


Figure D-16. Using expressions in .EQU statements

☞ Remember that our language does not support forward-referencing in `.EQU` statements, which means that `BEN` can reference `JOE` and `SAM` because they've already been declared; `TED` can reference `BEN` because that's already been declared; but `BEN` could not reference `TED`, `SAM` could not reference `BEN` or `TED`, and `JOE` could not reference `SAM`, `BEN`, or `TED`.

It shouldn't take you too long to work out that the value assigned to `BEN` is 10 (or $A in hexadecimal), while the value assigned to `TED` is 3 (or $3 in hexadecimal). As we see, we directly assign the value represented by `TED` to our first `.BYTE` statement, so the assembler will store $03 in address $4000 (note that the assembler adds a leading zero to pad the value to 8 bits). In the second `.BYTE` statement we use `TED` in a different role, because the construct "`*TED`" instructs the assembler to set aside a number of bytes equal to `TED`, which appear at addresses $4001 through $4003. Note that the assembler would actually load these bytes with default $00 values, but we've shown them as containing $XXs to indicate that we didn't explicitly assign these values.

Next we use `BEN` as the data to be loaded into the accumulator from the `LDA` instruction in its immediate addressing mode. The $90 at address $4004 is the opcode for the `LDA`, while the $0A at address $4005 is the value that was in `BEN`. As usual, the assembler adds a leading zero to pad this data value to eight bits. Finally, we use `BEN` as the target address for a `JMP` instruction in its absolute addressing mode. The $C1 at address $4006 is the opcode for the `JMP`, while the $000A stored in addresses $4007 and $4008 is the value that was in `BEN`. In this case, the assembler automatically adds three leading zeros to pad the address value to sixteen bits.

The above example used the results from a previously-calculated expression in the role of a data value and an address. In fact, we can also use expressions directly in data and address fields in the body of the program (Figure D-17).

☞ Unlike the expressions in `.EQU` statements, expressions in instructions can make forward references to future address labels, such as the "`JMP [YOW – 2]`" statement in the example shown in Figure D-17.

Remember that this is an artificial test case whose sole purpose is to illustrate some simple concepts, so don't spend an inordinate amount of time trying to wring some hidden meaning from it, or saying "*I wonder why they did it that way; wouldn't it have been simpler to .....?*" Yes it would and no we didn't!

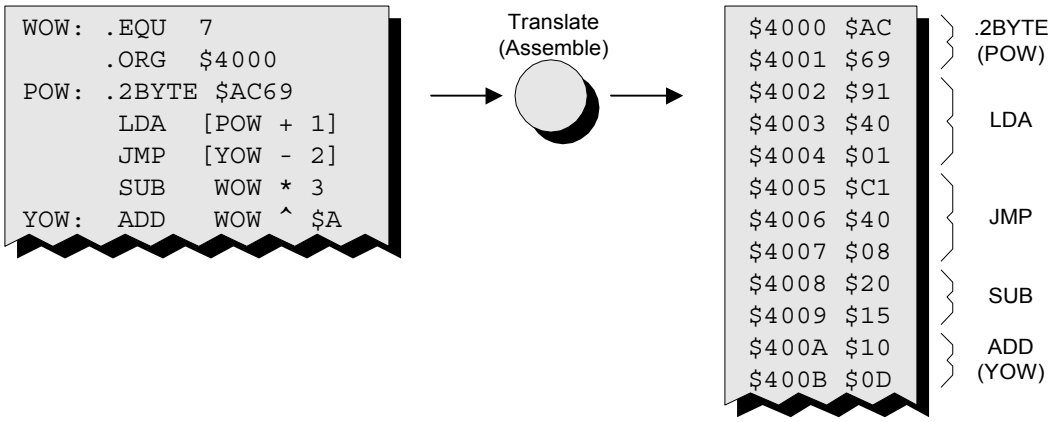


Figure D-17. Expressions in instruction statements

The first thing we do in this program is to declare a 2-byte field called `POW`, and load it with the number $AC69 (the value that appears in addresses $4000 and $4001). As we know, our assembler will consider `POW` to be associated with $4000, which is the address of the first byte in this 2-byte field. Thus, the statement "`LDA [POW + 1]`" (which is an `LDA` using its absolute addressing mode) is understood by the assembler to mean "`LDA [$4000 + 1]`", or "`LDA [$4001]`". This means "*Load the accumulator with the contents of address $4001*", so when we run the program (starting with the first real instruction at address $4002, of course), the accumulator will be loaded with $69.

Now glance at the end of this program segment, where we associate the label `YOW` with an `ADD` instruction. Our assembler will consider `YOW` to be associated with $400A, which is the address of the first byte in this 2-byte instruction. Thus, the "`JMP [YOW – 2]`" statement is understood by the assembler to mean "`JMP [$400A – 2]`", or "`JMP [$4008]`". By some strange quirk of fate, this happens to be the address of the first byte in the `SUB` instruction. (This means that our program would do exactly the same things (functionally speaking) if we chopped the `JMP` statement right out of it!)[2]

At the bottom of this program snippet are the `SUB` and `ADD` instructions. Both are using their immediate addressing modes and both employ expressions to calculate their data values. The "`SUB WOW * 3`" statement is understood by the assembler to mean "`SUB 7 * 3`", or "`SUB 21`" ("*subtract 21 from the current contents of the accumulator*"). The $20 at address $4008 is the opcode for this instruction, while the $15 at address $4009 is the hexadecimal equivalent of 21. Similarly, the "`ADD WOW ^ $A`" statement is understood by the assembler to mean "`ADD 7 ^ $A`" (where '^' is the exclusive OR operator), which boils down to "`ADD $0D`" ("*add $0D to the current contents of the accumulator*"). Thus, the $10 at address $400A is the opcode for this instruction, while the $0D at address $400B is the value to be added.

In common with many other assembly languages, our language includes a special symbol in its syntax that allows you (well, the assembler, really) to access the value that the program counter would contain at any particular point in the code. The symbol we decided to use is the '`@`' character, and an example of its use is illustrated in Figure D-18.

[2]Note the cunning use of nested parenthesis in this sentence. Our English teachers might not approve, but did they ever?

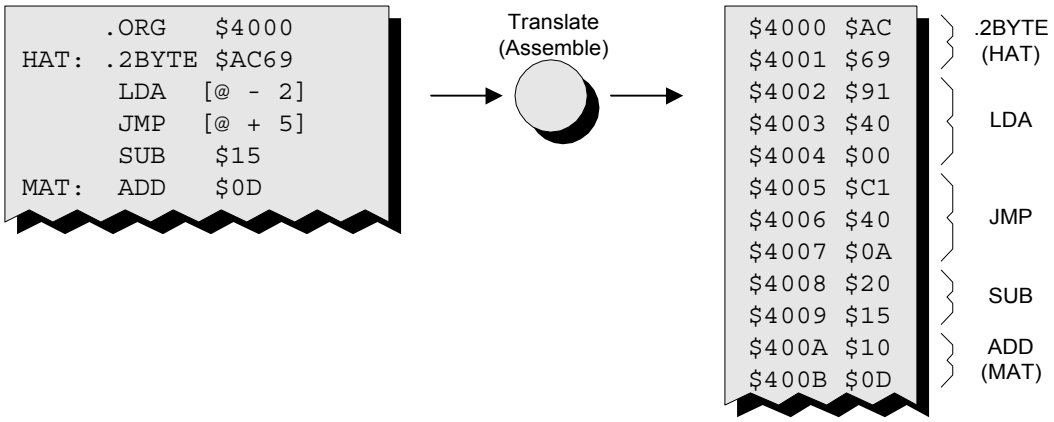


Figure D-18. Using the '@' symbol

When the assembler sees the '@' character, it substitutes it with the address of the first byte (the opcode byte) in that instruction. Thus, as the first byte of the LDA instruction occurs at address $4002, the assembler will understand the statement "LDA [@ – 2]" to mean "LDA [$4002 – 2]" or "LDA [$4000]". Similarly, as the first byte of the JMP instruction occurs at address $4005, the assembler will understand "JMP [@ + 5]" to mean "LDA [$4005 + 5]" or "LDA [$400A]". Note, however, that the '@' symbol can only be used in expressions in the body of the program, but it has no meaning in declaration (.EQU) statements.

Last but not least, it would be extremely remiss of us if we failed to mention that the *DIY Calculator's* assembly language supports two *unary operators*: the unary minus '–' and the unary negation '!'. (The exclamation mark we used to represent the unary negation is often referred to as a *bang*, *ping*, or *shriek*). The reason these are referred to as "unary operators" is that they only require a single operand upon which to work their magic (Figure D-19).

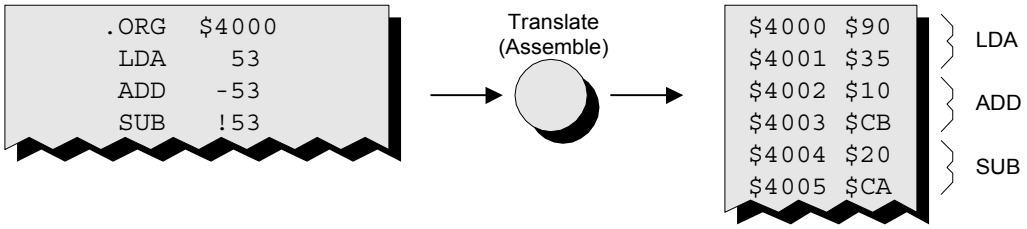


Figure D-19. The unary operators '–' and '!'

- The unary operators are associated with the value immediately to their right, where said value can be a numeric literal (in binary, decimal, or hexadecimal) or a label. Note that space characters are not allowed between a unary operator and its associated value.
- Both of the unary operators have a higher precedence than the binary operators. This means that in expressions containing a mixture of unary and binary operators, the effects of the unary operators will be evaluated first. For example, in the expression "TOM + !BERT", the unary negation will be applied to BERT and the result will be added to TOM. Similarly, in the expression "!TOM + BERT", the unary negation will be applied to TOM and the result will be

added to BERT. We can, of course, use parenthesis to force the issue, so if we wanted to add TOM to BERT and then negate the result, we could do so using "!(TOM + BERT)".

By some weird and wonderful coincidence, the $35 shown in address $4001 in Figure D-19 is the hexadecimal equivalent of 53 in decimal, which is the data value associated with the LDA instruction. In the case of the ADD instruction, the unary minus '–' takes the two's complement of 53, resulting in the $CB value at address $4001. By comparison, in the case of the SUB, the unary negation takes the one's complement of 53, resulting in the $CA value at address $4003.[3] This leads to an interesting point, which is related to the fact that the assembler stores all of its values in 4-byte fields. Consider how the assembler views these three values as it's manipulating them (Figure D-20).

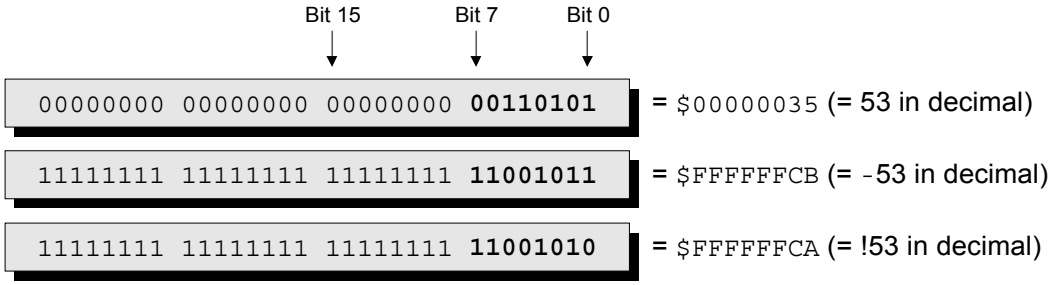


Figure D-20. How the assembler views the effects of the unary operators.

We know that the ADD and SUB instructions shown in Figure D-19 expect 8-bit data values for their immediate addressing modes. We also know that the assembler will issue an error message if a number is too big to fit into its target destination. However, although both of the values resulting from the unary operations in this example leave the three most-significant bytes of the assembler's 4-byte fields packed to bursting with 1s, we seem to be implying that the assembler will let them slip past without so much as raising a metaphorical eyebrow. Strange things are afoot indeed.

As it happens, there's a reasonably simple explanation for all of this (thank goodness). Like the heroes who paved the way before us, we decided that we wish to be able to assign negative numbers as data values in our language, but we also recognized that this might cause the assembler to "throw a wobbly." In order to solve this conundrum, we created the assembler in such a way that we might imagine it using the following reasoning:

> "*Hmmm, I'm supposed to assign this value to an 8-bit field, but the three most-significant bytes of my internal 4-byte field contain non-zero values, which suggests that these numbers are too big. Perhaps I ought to issue a warning message .....?*
>
> *But just a moment, let's look at this from another point of view. If the most-significant bit of the least significant byte (bit 7) is a* logic 1*, and if* all of the bits *in the three most significant bytes are also* logic 1s*, then I'd be justified in assuming that this is a negative number, in which case I can just use the least-significant byte, ignore the three most-significant bytes, and a nod's as good as a wink to a blind fruit bat.*"

[3]The concepts of ones, twos, nines, and tens complements are discussed in excruciating detail in our book *How Computers Do Math.*

And, in fact, this is just what our assembler does. Of course this means that things can mess up sometimes, but this would be a classic case of "*Let the programmer beware!*"

Last but not least, if the assembler is trying to assign one of these values to a 16-bit field, it essentially follows the same procedure, except that in this case it looks to see if bit 15 is a *logic* 1 and if all of the bits in the *two* most-significant bytes are logic 1s.

Instructions and addressing modes

Cease that nervous twitching and sit up straight, because this is the last section in the language specification and it's quite possibly the easiest one of all. As we discussed in *Appendix A*, the *DIY Calculator's* CPU supports a variety of addressing modes. Thus, our assembly language requires some way of distinguishing which mode we're trying to use, especially since some instructions can employ more than one mode.

The implied addressing mode: The simplest mode of all is the *implied* addressing mode, in which the instruction only requires a single byte for the opcode and there are no operand bytes. For example, consider a `SHL` ("*shift left*"), which instructs the CPU to shift the contents of the accumulator left by one bit (Figure D-21).

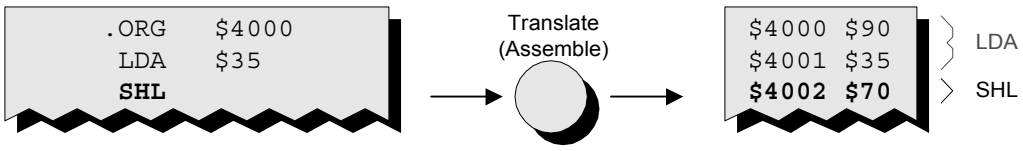


Figure D-21. The implied addressing mode

Note that the only reason we included an `LDA` instruction in this example is that we need something to be the accumulator before we can do anything with it. The point we're trying to make here is that instructions like `SHL` that use the implied addressing mode don't have an operand in the source (assembly) code and they occupy only a single byte in the object (machine) code.

☞ Instructions that support the implied addressing mode are `CLRIM`, `DECA`, `DECX`, `HALT`, `INCA`, `INCX`, `NOP`, `POPA`, `POPSR`, `PUSHA`, `PUSHSR`, `ROLC`, `RORC`, `RTI`, `RTS`, `SETIM`, `SHL`, and `SHR`.

The immediate addressing mode: In the case of the *immediate* addressing mode, the data to be used is directly associated with the instruction. Such instructions occupy two bytes: one for the opcode and one for the data (Figure D-22).

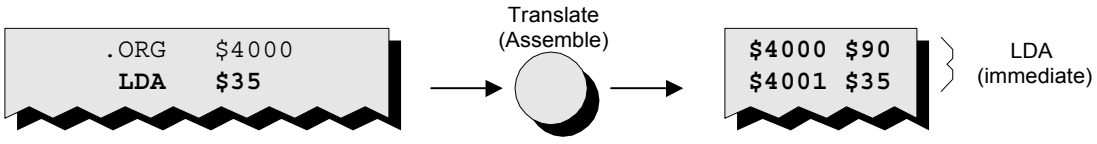


Figure D-22. The immediate addressing mode

☞ Instructions that support the immediate addressing mode are `ADD`, `ADDC`, `AND`, `CMPA`, `LDA`, `OR`, `SUB`, `SUBC`, and `XOR`.

The "big immediate" addressing mode: This mode is very similar to the standard immediate mode, in that the data to be used is directly associated with the instruction. We decided to distinguish between these modes because, unlike the `LDA` which loads an 8-bit accumulator, certain instructions are used to load 16-bit registers, such as the stack pointer (SP), the index register (X), and the interrupt vector (IV). Thus, these instructions occupy three bytes, one for the opcode and two for the data (Figure D-23).

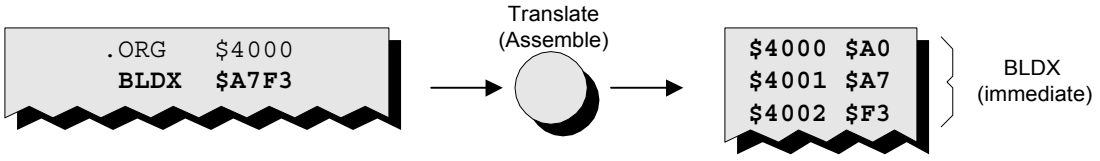


Figure D-23. The "big immediate" addressing mode

- All of the "big" instructions (those that involve 16-bit registers) are distinguished by the fact that the first letter in their mnemonics is 'B' for "Big" (how subtle can you get?). Instructions that support the "big immediate" addressing mode are `BLDSP`, `BLDX`, and `BLDIV`.

The absolute addressing mode: Unlike the immediate addressing mode, in which the instruction's operand represents a data value, the operand for an absolute instruction is a 2-byte address. Thus, these instructions occupy three bytes, one for the opcode and two for the address (Figure D-24).

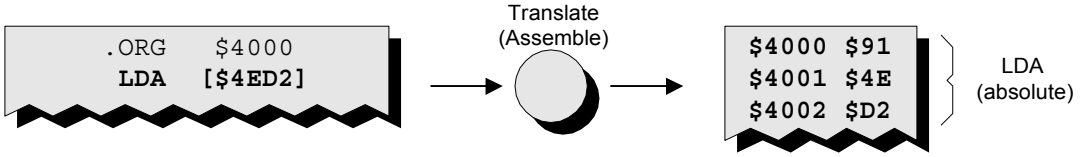


Figure D-24. The absolute addressing mode

- The absolute mode is distinguished from the immediate mode in the assembly source file by enclosing the operand in square brackets '`[`' and '`]`'. We decided to use square brackets because they visually imply a memory location. Thus, "`LDA [$4ED2]`" means "*Load the accumulator with the data value that will be found in the memory location at address $4ED2*."
- "Big" absolute instructions (such as `BLDX`) use the same format as do standard absolute instructions (such as `LDA`). In the case of the "big" instructions, however, the assembler – and the CPU – understands that the 2-byte address operand actually points to the first byte of data in a 2-byte field.
- Instructions that support the absolute (and "big" absolute) addressing mode are `ADD`, `ADDC`, `AND`, `BLDSP`, `BLDX`, `BLDIV`, `BSTSP`, `BSTX`, `CMPA`, `JC`, `JNC`, `JN`, `JNN`, `JO`, `JNO`, `JZ`, `JNZ`, `JMP`, `JSR`, `LDA`, `OR`, `STA`, `SUB`, `SUBC`, and `XOR`.

The indexed addressing mode: The indexed addressing mode is very similar to the absolute mode, not the least that these instructions also occupy three bytes, one for the opcode and two for the address (Figure D-25).

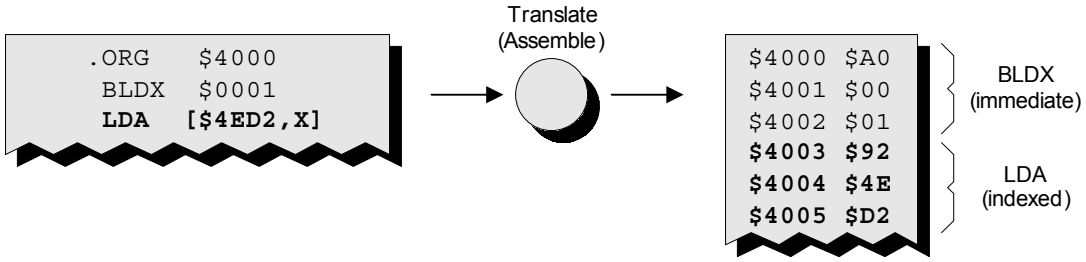


Figure D-25. The indexed addressing mode

- The indexed mode is distinguished from the absolute mode in the assembly source file by including the special keyword "X" in the square brackets; where the "X" – which is separated from the address by a comma – is a shorthand mnemonic for "inde**X** register." As you may recall, we previously noted that you couldn't use "X" as the name for a label because it was one of our reserved words ..... now you know "Y" ("why" – get it? – come on, work with us on this).

- Observe that the target address ($4ED2) in the machine code associated with the `LDA` using the absolute addressing mode in Figure D-24 is identical to the target address associated with the `LDA` using the indexed addressing mode in Figure D-25.

  However, when the CPU sees the $91 opcode in Figure D-24, it understands that this version of the LDA instruction is using the absolute addressing mode, which means the following two bytes contain the absolute address of the data to be loaded. By comparison when the CPU sees the $92 opcode in Figure D-25, it understands that – after reading the 2-byte address in the following two bytes – it has to internally add the current contents of the index register to this address, and to then use this modified address to point to the target location containing the data (or into which it should write the data in the case of a store instruction).

  Thus, the CPU understands "`LDA [$4ED2,X]`" to mean "*Load the accumulator with the data value that will be found in the memory location generated by adding the current contents of the index register to address $4ED2*."

- Instructions that support the indexed addressing mode are `ADD`, `ADDC`, `AND`, `CMPA`, `JMP`, `JSR`, `LDA`, `OR`, `STA`, `SUB`, `SUBC`, and `XOR`.

The indirect addressing mode: When we play with the indirect addressing mode, we're really starting to cook on a hot stove (Figure D-26).

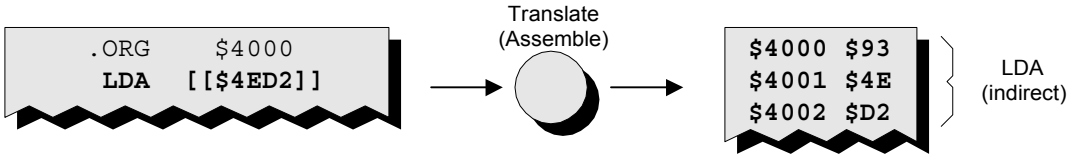


Figure D-26. The indirect addressing mode

- The whole point of an instruction using the indirect mode is that its operand is not an address that points to the data, but rather an address that's pointing to a second address, where this second address is the one that points to the data. Purely for aesthetic reasons, we decided to describe this in our source file using pairs of square brackets ("`[[`" and "`]]`"), which we understand to mean "*an address pointed to by an address*." Thus, "`LDA [[$4ED2]]`" means

"*Load the accumulator with the data value that will be found in the memory location that's pointed to by a 2-byte address whose first byte is at address $4ED2*" (try saying that ten times quickly without taking a breath).

- Instructions that can support the indirect addressing mode are `JMP`, `JSR`, `LDA`, and `STA`. Note that there's no reason why other instructions such as `ADD`, `SUB`, `AND`, and `OR` should not be equipped with an indirect addressing mode (other than the fact that when we were designing the *DIY Calculator's* CPU, we made the decision to not implement this mode for these instructions).

The pre-indexed indirect addressing mode: (Don't worry, we're almost done). As its name might suggest, the pre-indexed indirect addressing mode is a combination of the indexed and indirect addressing modes that we've already seen (Figure D-27).

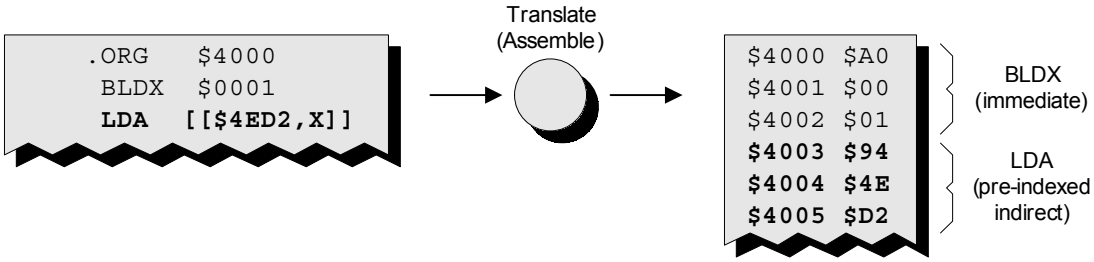


Figure D-27. The pre-indexed indirect addressing mode

- In this case, the CPU first adds the current contents of the index register to the address it finds in the operand bytes, and it uses this generated address to point to a second address that actually points to the data. Thus, "`LDA [[$4ED2,X]]`" means "*Load the accumulator with the data value that will be found in the memory location that's pointed to by a 2-byte address, whose first byte is at the address generated by adding the current contents of the index register to address $4ED2*".
- Instructions that support the pre-indexed indirect addressing mode are: `JMP`, `JSR`, `LDA`, and `STA`.

The indirect post-indexed addressing mode: Settle down and cease your whimpering and whining ... this is the very last one. Once again, as its name might suggest, the indirect post-indexed addressing mode is a combination of the indirect and indexed addressing modes that we've already seen (Figure D-28).

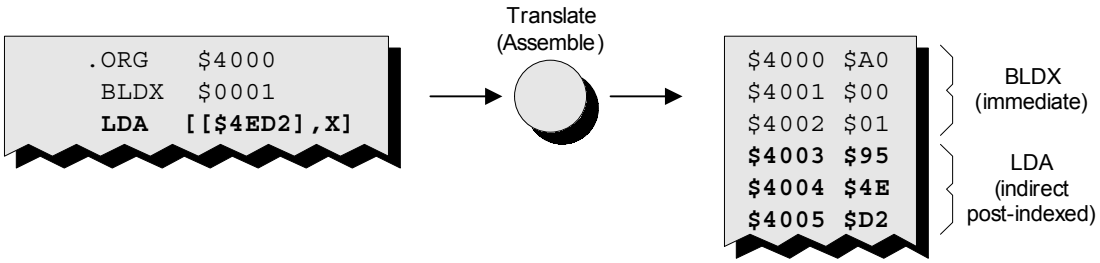


Figure D-28. The indirect post-indexed addressing mode

- The pre-indexed indirect mode and this indirect post-indexed mode are distinguished in the assembly source file by the positioning of the "X" keyword. In the post-indexed case, the CPU uses the address it finds in the instruction's operand bytes to point to a second address, it internally adds the current contents of the index register to this second address, and it uses the

result to point to the data. Thus, "`LDA [[$4ED2],X]`" means "*Load the accumulator with the data value that will be found in the memory location that's pointed to by a 2-byte address, where this address is determined by adding the current contents of the index register to the address whose first byte is located at address $4ED2*".

☞ Instructions that support the indirect post-indexed addressing mode are `JMP`, `JSR`, `LDA`, and `STA`.

These last two modes are fairly esoteric and you might not find much use for them in the average program. In fact, you can actually write any program using just the implied, immediate, and absolute addressing modes (or just the implied and absolute modes at a pinch) – the other modes are simply there to make our lives easier and our programs smaller and faster. On the other hand, should you become enamored of creating your own programs, you might be pleasantly surprised to discover just how frisky the more sophisticated modes can be.

Appendix E

Assembly Language
In Backus-Naur Form

Backus-Naur Notation

Introduced by John Backus and Peter Naur in the late 1950s and early 1960s, Backus-*Naur Form (BNF)* is a technique for recursively defining the grammar – the words, symbols, and tokens – associated with a computer language. BNF allows us to represent a language using a combination of syntactic entities and meta-syntactic symbols, where the meta-syntactic symbols are used to define how the syntactic entities can be combined. An individual syntactic definition is composed of a name in *italic font*, followed by the definition symbol '≡' (meaning "*is defined as*"), followed by the definition itself; for example:

alpha_char ≡ *'A' through 'Z' and 'a' through 'z'*
num_char ≡ *'0' through '9'*
alphanum_char ≡ *alpha_char* | *num_char*

These statements define three syntactic entities: *alpha_char*, *num_char*, and *alphanum_char*. The approach we employ is bottom-up, with fundamental items being defined in advance of any constructs that use them. Note that syntactic entities are only ever defined once, but they may be used multiple times in the definitions of subsequent entities. Also note the use of the | ("vertical bar") symbol in the third example. This is one of the meta-syntactic symbols, which are used to define how syntactic entities can be combined. The meta-syntactic symbols are as follows:

| Indicates alternative choices; for example:

alphanum_char ≡ *alpha_char* | *num_char*

This means that an *alphanum_char* can be either an *alpha_char* or a *num_char*.

{ … } Indicates part of a definition that can be repeated zero or many times; for example:

dec_literal ≡ *num_char*{*num_char*}

The first part of this definition means that a *dec_literal* ("decimal literal", which basically means a number) must consist at least one *num_char*, while the second part enclosed by the {} symbols means that it may also contain zero or more additional instances of *num_char*.

(…) Indicates part of a definition that contains options separated by the | symbol; only one of the options can be selected; also one of the options must be selected; for example:

address_label ≡ (_ | *alpha_char*){*label_char*}

The first part of this definition enclosed by the ( ) symbols means that an *address_label* must commence with either an underscore character or an *alpha_char*, while the second part enclosed by the {} symbols means that it may also contain zero or more instances of *label_char*.

[…] Indicates part of a definition that is optional; for example:

implied_instruction ≡ [*address_label:*] *implied_mnemonic* [*comment*]

This means that an *implied_instruction* consists of an optional *address_label*; followed by a mandatory *implied_mnemonic*; followed by an optional *comment*.

Note that you've got to be careful with some of the definitions, because in addition to being meta-syntactic, symbols such as |, [ ], and ( ) are also part of the assembly language itself; for example:

absolute_instruction ≡
[*address_label:*] *immediate_mnemonic [integer_expression]* [*comment*]

Here we see that the *[ ]* characters used to delimit the *integer_expression* are part of the syntax of our assembly language, and are therefore presented in *11-point italic font*. This distinguishes them from their meta-syntactic cousins [ ], which are presented in 14-point normal font.

Formal syntax summary

The remainder of this appendix is devoted to a syntactic summary of the *DIY Calculator's* assembly language described in the Backus-Naur representation introduced above. In an ideal world, our Backus-Naur description would fully and completely define our language without requiring any augmentation; but in practice we discover that a few well-placed notes tend to ease the way considerably, and these notes are indicated by pointing finger characters (☞).

Names

alpha_char ≡ *'A' through 'Z' and 'a' through 'z'*

num_char ≡ *'0' through '9'*

alphanum_char ≡ *alpha_char* | *num_char*

label_char ≡ *alphanum_char* | _

constant_label ≡ (_ | *alpha_char*){*label_char*}

address_label ≡ (_ | *alpha_char*){*label_char*}

any_label ≡ *constant_label* | *address_label*

☞ Both types of labels (*address_labels* and *constant_labels*) must commence with either an underscore '_' character or an alpha character.

☞ Although this syntax doesn't impose a limit on the length of labels, the *DIY Calculator's* current assembler implementation restricts these entities to a maximum of eight characters including any underscores. Also, labels are not allowed to be one of the language's keywords (which are listed at the end of this appendix).

☞ Although the syntactic entity *alpha_char* includes both uppercase and lowercase characters, the *DIY Calculator's* assembler internally converts all characters to uppercase for the purposes of its machinations. For example, *fred*, *Fred*, *FrEd*, and *FRED* will be considered to be identical.

Text, comments, and blank lines

horizontal_whitespace ≡ *<tab>* | *<space>*

vertical_whitespace ≡ *<cr>* | *<lf>* | *<ff>* {Carriage return, Line feed, Form feed}

blank_line ≡ {*horizontal_whitespace*} *vertical_whitespace*

extra_char ≡ ` | ‘ | “ | ~ | *!* | @ | # | *$* | *%* | ^ | *&* | * | *(* | *)* | - | _ | = | + | \ | / | | | *[* | *]* | *{* | *}* | *;* | *:* | . | , | < | > | *?*

text_char ≡ *alphanum_char* | *extra_char* | *horizontal_whitespace*

comment ≡ #{*text_char*} *vertical_whitespace*

- The *horizontal_whitespace* characters *<tab>* and *<space>* are equivalent to the keyboard keys of the same name, and they equate to the ASCII codes $09 and $20, respectively.
- The *vertical_whitespace* characters *<cr>*, *<lf>*, and *<ff>* (“*carriage return*”, “*line feed*”, and “*form feed*”) equate to the ASCII codes $0D, $0A, and $0C, respectively. Note that *<cr>* may be called “*return*” or “*enter*” on your keyboard, while the other *vertical_whitespace* characters typically don’t have keyboard equivalents.
- Comments can commence anywhere on an input line. A comment starts with a ‘#’ character followed by any *text_chars*, and is terminated by any *vertical_whitespace* character.

Literals

bin_num_char ≡ *‘0’ through ‘1’*

hex_num_char ≡ *‘0’ through ‘9’ and ‘A’ through ‘F’ (or ‘a’ through ‘f’)*

dec_literal ≡ *num_char*{*num_char*}

bin_literal ≡ *%bin_num_char*{*bin_num_char*}

hex_literal ≡ *$hex_ num_char*{*hex_num_char*}

integer_ref ≡ *dec_literal* | *bin_literal* | *hex_literal* | *any_label*

- An *integer_ref* is either a *dec_literal* (decimal literal), *bin_literal* (binary literal), *hex_literal* (hexadecimal literal), or a label that’s been assigned to such a literal or which equates to an address.
- Decimal literals are formed from one or more decimal digits with no spaces or commas between them (e.g. 32565 and not 32,565). Binary literals are formed from one or more binary digits, which must be prefixed by a ‘%’ character (e.g. %00110101). Hexadecimal literals are formed from one or more hexadecimal digits, which must be prefixed by a ‘$’ character (e.g. $E0A2). Note that no whitespace characters are permitted between the ‘%’ and ‘$’ characters and their associated arguments.
- All *integer_refs* are automatically padded with zeros to occupy four bytes within the assembler. An *integer_ref* (or the result of an expression using *integer_refs*) can be used as a data value, an address value, or both. In these cases the assembler will issue a warning if the value is too large for its intended purpose (this point is discussed in more detail in *Appendix D*).

Expressions

arithmetic_operator ≡ + | – | * | / {Add, Subtract, Multiply, Divide}

logical_operator ≡ & | | | ^ {AND, OR, Exclusive-OR}

unary_ operator ≡ – | ! {Unary minus, Unary NOT)

any_operator ≡ *arithmetic_operator* | *logical_operator*

integer_primary
≡ *integer_ref* | *integer_expression* | @ | *unary_operator integer_primary*

integer_expression ≡ [( ] *integer_primary* {*any_operator integer_primary*} [ )]

- Note that *integer_primaries* are self-referential, in that they may be generated as a combination of a *unary_operator* (– or *!*) and another *integer_primary*. Also note that *integer_primaries* and *integer_expressions* are recursively defined in that they each refer to the other.

- The unary minus operator (which would appear in the form –*integer_primary*) generates the two's complement of the *integer_primary* with which it is associated. The unary NOT operator (which would appear in the form *!integer_primary*) generates the one's complement of the *integer_primary* with which it is associated (that is, it inverts all of the bits in the *integer_primary*). Note that no space characters are allowed between a *unary_operator* and its *integer_primary*.

- Expressions in *declaration_statements* (see the directive statements section below) may not employ forward-referencing, but may employ only literals (binary, decimal, hexadecimal) and previously declared *constant_labels*. By comparison, expressions in *reserve_statements* or *instruction_statements* may use any type of label, including forward-references to *address_labels* (these points are discussed in more detail in *Appendix D*).

- The *DIY Calculator's* assembler supports only a simple expression syntax, in which the binary operators +, –, *, /, &, |, and ^ all have equal precedence (the unary operators ! and – have a higher precedence than the binary operators). By default, expressions are evaluated from left to right, so you have to use parenthesis ( ) to force particular ordering of any sub-expression evaluations.

- Although not apparent from the syntax above, parenthesis must be balanced. That is, any open bracket '(' must have a matching close bracket ')'.

- The results from any integer divisions will be truncated without warning (e.g., 8 / 3 will return 2); however, any attempt to divide by zero will cause the assembler to flag an error.

- The '@' character directs the assembler to substitute whatever value the program counter will contain at this point in the program. For example, consider the statement "*JMP [@ + 6]*", in which the "*[@ + 6]*" will be translated into the address of the `JMP` instruction's opcode plus 6.

Instruction mnemonics

implied_mnemonic ≡ *CLRIM* | *SETIM* | *INCA* | *DECA* | *INCX* | *DECX* | *HALT* | *NOP* | *PUSHA* | *POPA* | *PUSHSR* | *POPSR* | *SHL* | *SHR* | *ROLC* | *RORC* | *RTI* | *RTS*

immediate_mnemonic ≡ *ADD* | *ADDC* | *SUB* | *SUBC* | *AND* | *OR* | *XOR* | *CMPA* | *LDA*

big_immediate_mnemonic ≡ *BLDIV* | *BLDSP* | *BLDX*

absolute_mnemonic ≡ *ADD* | *ADDC* | *SUB* | *SUBC* | *AND* | *OR* | *XOR* | *CMPA* |*LDA* | *STA* | *BLDIV* | *BLDSP* | *BLDX* | *BSTSP* |*BSTX* | *JC* | *JNC* | *JN* | *JNN* | *JO* | *JNO* | *JZ* | *JNZ* | *JMP* | *JSR*

indexed_mnemonic ≡ *ADD* | *ADDC* | *SUB* | *SUBC* | *AND* | *OR* | *XOR* | *CMPA* | *LDA* | *STA* | *JMP* | *JSR*

indirect_mnemonic ≡ *LDA* | *STA* | *JMP* | *JSR*

preindexed_indirect_mnemonic ≡ *LDA* | *STA* | *JMP* | *JSR*

indirect_postindexed_mnemonic ≡ *LDA* | *STA* | *JMP* | *JSR*

☞ We distinguish between *immediate_mnemonics* and *big_immediate_mnemonics*, because the former only require a single byte of data while the latter require two bytes of data. Thus, differentiating between these two groups allows our assembler to perform additional error checking.

Instruction statements

instruction_statement ≡ *implied_instruction* | *immediate_instruction* | *big_immediate_instruction* | *absolute_instruction* | *indexed_instruction* | *indirect_instruction* | *pre-indexed_indirect_instruction* | *indirect_post-indexed_instruction*

implied_instruction ≡ [*address_label:*] *implied_mnemonic* [*comment*]

immediate_instruction ≡ [*address_label:*] *immediate_mnemonic* *integer_expression* [*comment*]

big_immediate_instruction ≡ [*address_label:*] *big_immediate_mnemonic* *integer_expression* [*comment*]

absolute_instruction ≡ [*address_label:*] *immediate_mnemonic* *[integer_expression]* [*comment*]

indexed_instruction ≡ [*address_label:*] *indexed_mnemonic* *[integer_expression , X]* [*comment*]

indirect_instruction ≡ [*address_label:*] *indirect_mnemonic*
[[integer_expression]] [*comment*]

pre-indexed-indirect_instruction ≡ [*address_label:*] *preindexed_indirect_mnemonic*
[[integer_expression , X]] [*comment*]

indirect-post-indexed_instruction ≡ [*address_label:*] *indirect_postindexed_mnemonic*
[[integer_expression] , X] [*comment*]

- In the absence of a comment, any instruction can have trailing *horizontal_whitespace* characters. Also, every instruction is terminated by a *vertical_whitespace* character.
- An *integer_expression* in an *instruction_statement* may employ forward-referencing. That is, these expressions may employ both *constant_labels* and *address_labels* (and, of course, literal values).
- The '*X*' character in the *indexed*, *pre-indexed-indirect*, and *indirect-post-indexed* instruction statements is a special keyword that is understood by the assembler to refer to the index register, thereby causing the assembler to select the appropriate opcode for that instruction (see also the "*Reserved words*" section at the end of this appendix).

Directive statements

origin_statement ≡ *.ORG integer_ref* [*comment*]

end_statement ≡ *.END* [*comment*]

declaration_statement ≡ *constant_label: .EQU integer_expression* [*comment*]

reserve_statement ≡ [*address_label:*] (*.BYTE* | *.2BYTE* | *.4BYTE*)
[**integer_expression* | *integer_expression*
{*, integer_expression*}] [*comment*]

- An *integer_expression* in a *declaration_statement* may not employ any forward-referencing, but may only employ previously declared *integer_refs*. To put this another way, these expressions may only employ literal values or previously declared *constant_labels*.
- A *.BYTE* statement without an associated operand will reserve a single byte of memory for the program's future use. By comparison, a *.BYTE *n* will reserve 'n' bytes, where 'n' can either be a literal value (for example, *.BYTE *10*, which will reserve ten bytes), or an integer expression (for example, *.BYTE *(FRED + 3)*, which, assuming *FRED* equals 7, will also reserve ten bytes).
- By default, any locations set aside by *reserve_statements* will be initialized by the assembler to contain zero values. Alternatively, they may be explicitly initialized as part of the statement; for example, the statement *.BYTE 2, 32, 14, 42* will cause the assembler to reserve four bytes and initialize these bytes with the values 2, 32, 14, and 42, respectively. Additionally, any of the values in this comma-separated list could be full-blown *integer_expressions*.
- The *.2BYTE* and *.4BYTE* versions of reserve_statements work in a similar way to their *.BYTE* counterpart, except that (not surprisingly) they reserve two and four bytes, respectively. For example, the statement *.2BYTE $A42* will reserve a single 2-byte field and initialize that field to contain $0A42 (the assembler will automatically zero-extend the value to fit the 2-byte field). Note that our assembly language is based on a "big-endian" approach, in that multi-byte numbers are stored with their most-significant byte in the lowest address.

File structure

declaration_section ≡ {*declaration_statement* | *blank_line* | *comment*}

body_statement ≡ *reserve_statement* | *instruction_statement*

body_section ≡ *origin_statement* {*body_statement* | *blank_line* | *comment*} *end_statement*

assembly_source_file ≡ *declaration_section* *body_section* [{*comment* | *blank_line*}]

Reserved words

Although all of the reserved words are shown below in uppercase, they are in fact case-insensitive. Our assembler internally converts all characters to uppercase for the purposes of its machinations, and will therefore consider reserved words such as *add*, *Add*, and *ADD* to be identical.

Directive keywords

.ORG .END .EQU .BYTE .2BYTE .4BYTE

Instruction keywords

ADD	ADDC	AND	BLDSP	BLDX	BLDIV	BSTSP	BSTX	CLRIM
CMPA	DECA	DECX	HALT	INCA	INCX	JC	JNC	JN
JNN	JO	JNO	JZ	JNZ	JMP	JSR	LDA	NOP
POPA	POPSR	PUSHA	PUSHSR	ROLC	RORC	RTI	RTS	SETIM
SHL	SHR	STA	SUB	SUBC	XOR			

Special keywords

The only special keyword is ‘X’, which is used when an instruction is employing one of the indexed addressing modes. There is no theoretical reason why a label could not be named ‘X’, or indeed any of the instruction keywords. In practice, however, prohibiting such label names reduces confusion on the part of both the user and the assembler.