6502 ASSEMBLY INSTRUCTION SET

Load and Store Group

Load Register LDA, LDX, LDY

Copies the data referred to by the operand into the specified register.

The data cannot be processed until it has been fetched from the memory into the processor.

Store Register STA, STX, STY

Copies the data in the specified register into the memory location referred to by the operand.

Once the data has been processed, it will usually have to be stored back in the memory to make room in the processor for the next data item to be processed.

Arithmetic Group

Add with Carry ADC

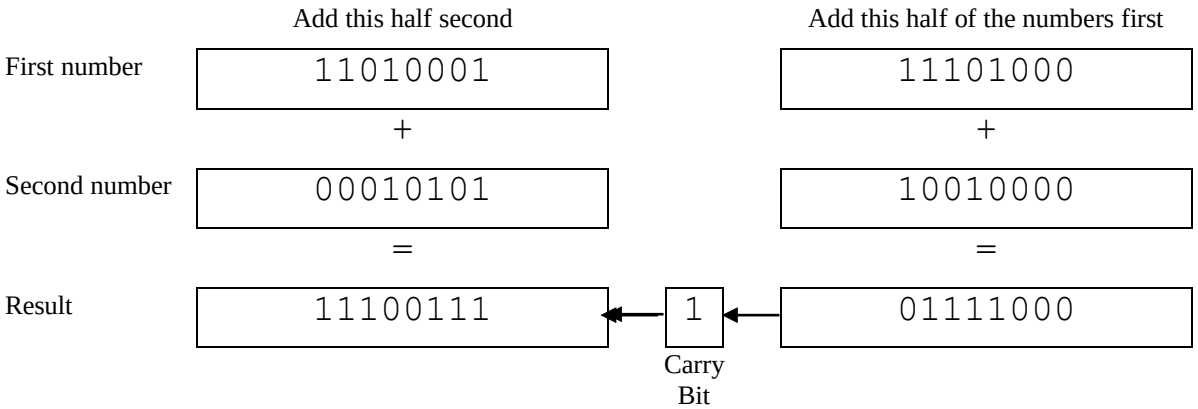
Adds the data item referred to by the operand to the contents of the Accumulator register, along with the value of the carry flag.

The value of the carry flag is always included in the addition. This is why it is good practice to clear the carry flag to zero before doing an addition, unless you are doing multi-byte addition.

Multi-byte addition is carried out when you want to add two numbers which are longer than the word length of the processor.

Let's say that there was a data type called HugeInt, which is an integer, held in two's complement form, which is 16 bits long. The 6502 processor has a wordlength of 8 bits.

On our processor, a number of this type would have to be stored in two consecutive memory locations, with the least significant half (little end) first. This means that when we wanted to add two such numbers together, we would have to do it in two stages. First we would add the two least significant halves of each number (which is why those bytes usually always stored first in memory) and then the two most significant halves of the number:



As with the example above, there can be a problem if a one is carried from the result of adding the two most significant (left most) bits of the least significant (right most) halves of the numbers. To get a correct result, this will have to be added to the least significant bits of the most significant halves of the numbers. This means that we need to store it somewhere, while we load the second half of each number into the registers. We use the carry flag for this purpose.

Subtract with Carry SBC

Subtracts the data item referred to by the operand from the contents of the Accumulator register.

NOTE: In order to get the correct result from a subtraction, the carry flag must be set to one beforehand.

This is because the circuits were designed as an exact inverse of the addition circuits. Otherwise, the carry flag is used in exactly the same way when doing multi-byte subtraction. It is sometimes called the borrow flag in this context.

Increment and Decrement Group

Increment / Decrement INC, DEC

Add one to (Subtract one from) the data item referred to by the operand.

These instructions allow you to quickly add one or subtract one from a number which is held in memory, without having to load the number into the Accumulator first.

Increment / Decrement Register INX, INY, DEX, DEY

Add one to (Subtract one from) the contents of the specified register.

Due to these instructions, an index register is the best place to store a count value. For example, you may need to count how many times you go round a loop. All you need to do is initialise the contents of the index register before you enter the loop, and use the increment instruction inside the loop.

Register Transfer Group

Transfer Register to Register TAX, TAY, TXA, TYA

Copy the contents of one register into another register.

You may need to transfer the contents of one of the index registers into the accumulator, to do some arithmetic on it. Or you may just want to use an index registers as a temporary storage area for a value, to save you having to store it in memory while you use the accumulator to process other data. Other processors have more than three registers, so that they can process large amounts of data without having to constantly store them in memory in between calculations.

Logical Group

Logical AND AND

Perform a bit by bit AND operation between the data item referred to by the operand, and the contents of the Accumulator. Put the result in the Accumulator.

This is a way of combining the corresponding bits of two binary codes which is based on the circuitry built in to the ALU.

First Input	Second Input	Output
0	0	0
0	1	0
1	0	0
1	1	1

The AND operation is often used to mask off particular parts of a binary code.

Note in the example on the right how only half of the first code is transferred to the result register.

```
10011011
AND
00001111
=
00001011
```

Logical Inclusive OR ORA

Perform a bit by bit OR operation between the data item referred to by the operand, and the contents of the Accumulator. Put the result in the Accumulator.

The OR operation is another of the logical operations, which allow you to combine two binary codes.

First Input	Second Input	Output
0	0	0
0	1	1
1	0	1
1	1	1

The OR operation can also be used to mask off certain parts of a binary code, as shown in the example on the right.

```
10011011
  ORA
00001111
   =
10011111
```

Exclusive OR EOR

Perform a bit by bit exclusive OR operation between the data item referred to by the operand, and the contents of the Accumulator. Put the result in the Accumulator.

The EOR operation is especially useful for testing to see if two codes are equal or not.

First Input	Second Input	Output
0	0	0
0	1	1
1	0	1
1	1	0

The XOR operation is sometimes called Not Equal, because if two identical codes are XOR'ed together, the result will always be zeroes – which will set the zero flag.

```
10011011
  EOR
10011011
   =
00000000
```

Compare and Bit Test Group

Compare register CMP, CPX, CPY

The compare operation **pretends** to subtract the data item referred to by the operand from the contents of the register, and sets the flags according to what the result would have been. The contents of the register are left unchanged.

The compare operation is used when you want to find out what is in a register. For example:

```
        CMP #'X'
        BEQ EOP
         :
EOP:    BRK
```

The CMP operation will subtract the ASCII code for the character X from the code in the accumulator. If the accumulator contains the code for X as well, the result of the subtraction would be zero, and so the zero flag will be set. This will then be tested by the branch instruction.

Instead of actually doing the subtraction, and leaving zero in the accumulator, the original code is left intact in the accumulator, for later use.

Bit Test BIT

The BIT operation **pretends** to perform an AND operation on the data item referred to by the operand, and the contents of the Accumulator, and sets the flags according to what the result would have been. The contents of the accumulator register are left unchanged.

To be more precise, the flags are set as follows:

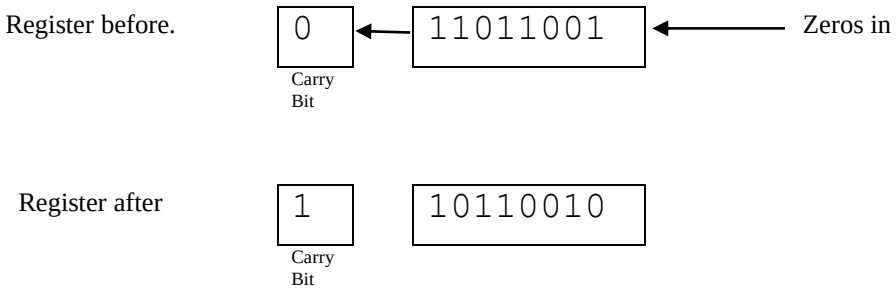
- Negative flag (N) is made equal to the initial state of bit 7 (MSB) of the data item in memory.
- Overflow flag (V) is made equal to the initial state of bit 6 of the data item in memory.
- Zero flag (Z) is set to 1 if the result of the pretend AND operation would have been an accumulator full of zeros.

Like the Compare operation, this is a 'flag setter operation', which allows you to see what is in the memory location or accumulator without actually changing its contents.

Shift and Rotate Group

Arithmetic Shift Left ASL

Shift the contents of the location specified by the operand 1 bit to the left. The most significant bit is transferred to the carry flag.

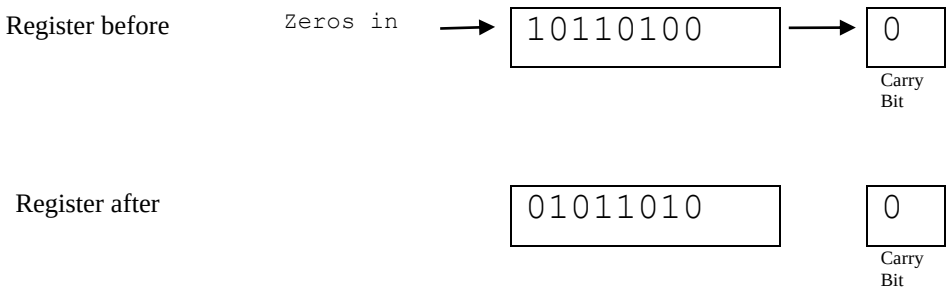


In general, shifting a binary number to the left has the effect of multiplying it by two. In the example above, we can test the carry flag and discover that we should not trust the result, as it was too large to fit into the accumulator register.

Logical Shift Right LSR

Shift the contents of the location specified by the operand 1 bit to the right. The least significant bit is transferred to the carry flag.

If the data represents a number, this has the effect of dividing it by 2.



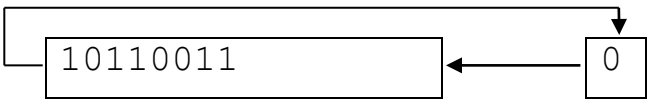
Rotate Accumulator Left **ROL**

Rotate Accumulator Right **ROR**

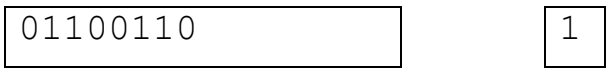
The action of these instructions is best shown with a diagram. The contents of the location specified by the operand are moved 1 bit to the right, or left, via the carry bit:

Rotate left

Register and carry bit before.

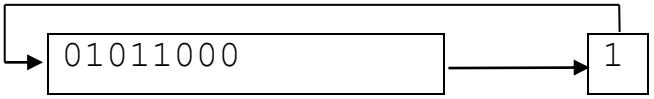


Register and carry bit after

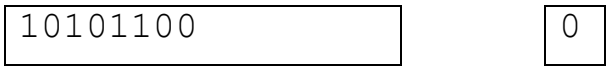


Rotate right

Register and carry bit before



Register and carry bit after



When you are using low level languages, you have to know what format your data is in, at the level of bits and bytes. If you don't, you won't know how to manipulate it.

Other processors have variations on these shift and rotate instructions that bypass bit 7 (MSB). That is to say, they treat it as a special case, because if the code is a number code, it may be acting as a sign bit, and you may want to multiply the number by 2 without changing its sign.

Jump and Branch Group

Unconditional Jump **JMP**

This will transfer control to the instruction referred to by the operand.

Conditional Branch **BCC, BCS, BEQ, BMI, BNE, BPL, BVC, BVS**

Transfer control to the instruction referred to by the operand, if the specified flag is set / clear.

The thing to remember with the branches is that the conditions (branch if zero, branch if greater than zero etc) do not refer to the contents of the accumulator register. They refer to the condition of the flag bits in the status register, which may have been set as the result of an operation that has been carried out on the accumulator - or on one of the other registers. It is always worth checking that the operation that you are testing does indeed set the flags in the way that you are expecting.

Why have a jump instruction and branch instructions?

Remember how we use branch instructions in machine code: we specify the number of bytes that we wish to jump to relative to the location of the branch instruction itself. So, for example, the conditional branch instruction, $F0 (BEQ – Branch if zero flag is set) is used like this:

```
F0 0A
```

which means jump to the instruction which is 0A (ten) bytes further on in memory than this one.

A jump instruction is used like this:

```
4C  58  4F
```

which means jump to the instruction in location 4F58.

The largest number of bytes we can specify in a branch is therefore FF, which is 255 decimal. So with branch instructions, we can only jump to places that are in the same area of memory as the branch instruction. With the jump instruction, we can specify any memory location in the entire memory, so the jump instruction is more flexible, but it turns out that 95% of the jumps in assembly language programs are no more than 100 bytes either way, so the branch instructions are not as restrictive as it may appear. And they take up less memory space and are therefore faster to execute.

Stack Manipulation Group

Push Register onto stack **PHA, PHP**

Pull word at top of stack into Register **PLA, PHP**

Copy the data in a specified register onto the top of the stack.

Copy the data on the top of the stack into the specified register.

The first page of memory ($0100 to $01FF) is used for the stack. The stack pointer register is used to store the address (i.e. point to) the next available memory location at the top of the stack. The stack grows downwards. i.e., it starts at a high address, and grows towards the low addresses.

The **Push** operation involves:

- Copying the word from the specified register to the empty memory location addressed by the stack pointer register.
- Decrementing the contents of the stack pointer register by one.

The **Pull** (aka Pop) operation involves:

- Incrementing the contents of the stack pointer register by one.
- Copying the byte addressed by the stack pointer into the specified register.

The stack is used for temporarily storing data items, and also for storing parameters, when you jump to a subroutine.

It is also used to store the current state of the processor (Flags register and contents of Accumulator) when an interrupt signal or a break signal is received, so that the relevant operating system routine can be loaded and executed. After the interrupt has been serviced, the state of the original program can be restored and execution can carry on from where it left off.

Transfer Stack Pointer to Index Register X **TSX**

Transfer Index Register X to Stack Pointer **TXS**

It is sometimes useful to store the contents of the stack pointer register in another location, temporarily. The index register can be used for this, provided it is not being used for another purpose.

Status Flag Change Group

Clear Flag **CLC, CLD, CLI, CLV**

Clear the specified flag to zero.

Set Flag **SEC, SED, SEI**

Set the specified flag to one.

Decimal Mode Flag D

The decimal mode flag (D) can be set by using SED and cleared by using CLD. When it is set to one, all arithmetic operations (ADC and SDC) will behave in a different way. Instead of treating the contents of the accumulator as an 8 bit binary number, the arithmetic unit will treat it as two 4 bit binary coded decimal integers. Some calculations are better done in this way, to avoid rounding errors.

Interrupt Disable Flag I

The Interrupt flag is used to disable (SEI) or enable (CLI) maskable interrupts coming in from a device external to the processor.

If your program is performing a delicate operation which must not be interrupted, you can set this flag so that it will ignore requests from other parts of the system for help.

The 6502 will set this flag automatically when an interrupt signal is received, and restore it to its prior status when the interrupt service routine has completed. It is not a good idea to have your interrupt service routine interrupted by yet another interrupt service routine.

Overflow Flag V

After an add or subtract operation, this will be set if the result of the operation was too large to fit in the accumulator. This is determined by seeing if the carry flag is different to the negative flag, which can be calculated by doing an exclusive OR on those two flags. See attached sheet for further details.

Subroutine and Interrupt Group

Jump to subroutine JSR

Saves return address on stack and jumps to the subroutine referred to by the operand.

This will cause control to be passed to a different part of the program. In other words, the address of the instruction being jumped to will be placed in the program counter register. Before we can do that, the processor needs to save the address of the current instruction, so that it knows where to jump back to, after the subroutine has finished executing. The section of the memory known as the stack is where the processor stores the return address.

When a JSR instruction is executed, the following steps are carried out:

1. The High End half of the address of the last byte of the JSR instruction is pushed onto the stack.
2. The stack pointer is decremented, so that it now points to the next free space.
3. The Low End half of the address of the last byte of the JSR instruction is pushed onto the stack.
4. The stack pointer is decremented, so that it now points to the next free space.
5. The address specified in the JSR instruction is loaded into the program counter register.

The instruction at the location specified by the operand will therefore be the next one to be fetched and executed.

Return from subroutine RTS

Pop return address off stack and jump back to the instruction immediately following the subroutine call.

This will cause control to be passed back to the main program, to the instruction immediately after the JSR instruction (the subroutine call). This means that the return address has to be retrieved from the stack.

When a RTS instruction is executed, the following steps are carried out:

1. The stack pointer is incremented.
2. The Low End of the return address is loaded into the low end of the program counter register.
3. The stack pointer is incremented.
4. The High End of the return address is loaded into the high end of the program counter register.
5. The contents of the program counter are incremented by one, so that instead of referring to the last byte of the JSR instruction, the program counter now contains the address of the first byte of the next instruction.

The stack can be used by the subroutine, but it must ensure that the return address is at the top of the stack when the subroutine ends.

Dealing with return addresses in this way means that subroutines may contain calls to other subroutines, or even calls to themselves, a technique known as recursion. The use of the stack ensures that the return addresses are recovered in the reverse order to that of the subroutine calls.

Force an Interrupt BRK

Jump to the interrupt service routines of the operating system.

We have been using the BRK instruction as a convenient halt instruction, to mark the end of the program. In fact, it is how the 6502 implements software interrupts. When a BRK instruction is executed, the 6502 carries out the following operations:

1. Push contents of Program Counter (return address) onto stack.
2. Push contents of Flags register onto stack.
3. Set Break Flag (B) to one, to indicate BRK command is being responded to.
4. Load address stored at locations $FFFE and $FFFF into program counter and continue execution from there.

The operating system in use must ensure that the address that is stored in the reserved locations $FFFE and $FFFF is the location of the start of the interrupt service routines

Return from Interrupt RTI

Return to the program that was being run when the interrupt occurred.

This is the last instruction of every interrupt service routine. When a RTI instruction is executed, the 6502 carries out the following operations:

1. Pop data off stack and copy into Flags register.
2. Pop return address off stack and copy into Program Counter register.

Execution of the original program can then continue.

No operation NOP

Do nothing.

A no-op instruction has no effect whatsoever on the processor, other than the fact that it takes a set number of clock cycles to execute. It is often used to create a delay, by padding out a loop.