# Simpler Assembler Notation (SAN) for the 6502, 65c02, and 65816

Scot W. Stevenson

Version ALPHA, 10. Nov 2018

# Table of Contents

# Introduction

The 6502 MPU has traditionally used an assembler notation with characteristic three-character upper-case opcodes. This can be traced at least all the way back to the *MCS6500 Microcomputer Family Programming Manual* from 1976 [6500] and is still used by Western Design Center (WDC) today. [ENL] The notation was later expanded to the additional instructions of the 65C02 and 65816. Though alternative notations can be found, the `LDA #$01` style is the *de facto* standard for the 6502.

This "traditional" notation for the 6502, however, has drawbacks. When expanded to the 65816, these become glaring, and are made even worse by curious naming decisions for the new instructions such as `PEA`. There have been suggestions for new variants here as well, but these haven't taken hold.

This document introduces an alternative notation that attempts to rectify these drawbacks, make the assembler codes easier to read, and the mnemonics more logical. Variants are provided for both pre- and postfix notations, so it can be used for assemblers written in languages such as Forth. At the same time, it tries to make the code easier to read, easier to process and even faster to type — hence the name, Simpler Assembler Notation (SAN).

# What is wrong with the traditional notation

The most glaring problem is that a lot of the opcodes do not by themselves define which machine language instruction they code for. Put differently, the traditional notation violates the separation of opcode and operand: To figure out which instruction is meant, the operand must be analyzed as well.

Take `LDA`, an opcode that on the 65816 can refer to any one of these 15 modes:

```
immediate, absolute, absolute long, direct page, direct page indirect, direct
page indirect long, absolute X indexed, absolute long X indexed, absolute Y
indexed, direct page X indexed, direct page indirect X indexed, direct page
indirect Y indexed, direct page indirect long Y index, stack relative, stack
relative indirect Y indexed
```

The fixation with three-character mnemonics means that the operand has to pick up the slack. Instead of consisting just of a number, they must be decorated with an assortment of special characters to define the opcode. Also, the numbers themselves must have a fixed number of digits to distinguish between, say, Absolute and Zero Page modes. The final proof that the operand is not just a parameter, but mixed up with the mnemonic, is that the X is added to the mnemonic, not the opcode, in the indexed modes.

Still, with the 6502, this system is still manageable, and gets by with little more than a comma, the hash sign, and round brackets:

| | |
|---|---|
| `LDA #$00` | immediate |

| | |
|---|---|
| `LDA $1000` | absolute |
| `LDA $1000,X` | absolute X indexed |
| `LDA $10` | zero page / direct page |
| `LDA ($10)` | zero page / direct page indirect (65c02) |

If the 6502/65c02 were the only processors in the family, a new notation would hardly be needed. However, the additional instructions of the 65816 forced the system to adopt even more special characters. Also, since the modes cannot always be reliably determined from the mnemonic and operand combinations, even more special characters are required: `<` is used to force direct page addressing, `!` for absolute, `>` for long addressing mode, `[` and `]` for long indirect modes. Traditional 65816 assembler notation can end up looking like something from a bash shell script or a regex instruction.

| | |
|---|---|
| `LDA >$7512` | long addressing (forced) |
| `LDA [$17],Y` | direct page indirect long indexed with Y |
| `LDA !12000` | absolute (forced), decimal address |

The long list of special characters not only makes the code hard to read, but also hard to type, especially on non-American keyboards. For instance, on German keyboards, the `[` and `]` have to be entered with a special ALT key. Combined with the tradition of using upper case letters, entering code is slow, especially for ten-finger typists.

# What is *really* wrong with the traditional notation: The 65816

Where the traditional syntax falls apart is with the new instructions such as `PEA` and `BRL`. `PEA` is one offender: Formally called "Push Effective Absolute Address" with the syntax `PEA $2222`, what it actually does is push the operand on the stack, not an absolute address as the formatting would suggest. Both the instruction name and format are misleading. Even the standard reference [ENL] states:

> The assembler syntax is that of the absolute addressing mode, that is, a label or sixteen-bit value in the operand field. Unlike all other instructions that use this assembler syntax, the effective address itself, rather than the data stored at the effective address, is what is accessed (and in this case, pushed onto the stack).

`PEI` suffers from similar problems. The `BRA` instruction syntax is inconsistent: Whereas the difference between "short" and "long" forms for `LDA` is signaled by the length of the operand as shown above, the "long" form of the branch instruction is given its own mnemonic, `BRL`.

# Constructing an improved syntax

When setting out to get rid of these problems, we have to remember that any new system must remain easy to read for coders who see it for the first time, even if they never use it themselves. Because of this, we cannot stray too far from the traditional syntax. This is especially true for the original 6502 mnemonics that have been in widespread use for decades. We have a bit more leeway with the new 65816 instructions, because the MPU is not so well known. In fact, we can try to make the more advanced processor easier to use with choice of mnemonics.

With that in mind, the new syntax has the following features:

**There is a one-to-one relationship between mnemonics and machine code instructions.** Put differently, every mnemonic represents one and only one opcode, not up to 15. No analysis of the operand is required to figure out what instruction we are dealing with. Besides removing the requirement for various special characters, this makes building assemblers and disassemblers far easier.

**All mnemonics are lower case.** This way, `LDA` becomes `lda`. This is a simple way to improve coding speed immediately, especially for ten-finger typists.

**We keep (almost) all of the original mnemonics as the three-character "stems" of the new opcodes.** This allows the code to be read by those programmers who have never even heard the name of the new syntax. So even though you could argue that `STA` should be named `stc` for a 16-bit accumulator when running in native mode on the 65816, this would make it too confusing. So we stick with `sta`.

**The addressing modes are coded as part of the opcode body, separated by a dot from the stem.** This is the opcode's "suffix". The tail suffix signals whether the instruction is direct page, immediate, X indexed etc. For example, `LDA $10` becomes `lda.z 10` and `STA $1000,X` becomes `sta.x $1000` under the new system. We go into detail below.

**The operand is pure parameter and not used for identifying the instruction.** This simplifies the writing of assemblers, because

```
        lda.z 0
        lda.z 00
        lda.z 0000
        lda.z 000000
        lda.z 00:0000           ; syntactic sugar
```

all result in the same machine language instruction, loading the accumulator with the content of zero page address 00.

**Some 65816 instructions are reorganized and renamed.** For example, `BRL` is a "long" form of `BRA`, so we keep `bra` for the short "base" form and `bra.l` for the long version. `PEA`, `PEI`, and `PER` are folded into one family with the common stem `phe` (PusH Effective address) and different suffixes. These new versions are discussed below.

# Complete list of addressing modes

The mnemonic suffixes follow the names of the addressing modes. The "naked" stem without a suffix is used for either for Implied instructions (such as `dex`) or Absolute Mode (`sta $1000`), following their traditional use. Where there is a possible conflict with Accumulator modes (e.g. `INC` and `INC A`), the suffix `a` is used for the Accumulator version. Indexed modes receive the letter for the register they are indexed with (e.g. `sta.x 1000`).

Indirect modes are marked with an `i` that is placed where the bracket would be in traditional notation. This way, `LDA ($10,X)` becomes `lda.dxi` and `LDA ($10),Y` becomes `lda.diy`.

We keep the hash symbol () for Immediate mode because though it is a special character, at this point it is too deeply ingrained to change without major disruption (e.g. `lda. 33`).

These and other variants give us the following complete list of modes (for the 65816):

| Mode | Traditional Notation | Simpler Notation |
|---|---|---|
| Implied | `DEX` | `dex` |
| Absolute | `LDA $1000` | `lda $1000` |
| Accumulator | `INC A` | `inc.a` |
| Immediate | `LDA #$00` | `lda.# $00` |
| Absolute X indexed | `LDA $1000,X` | `lda.x $1000` |
| Absolute Y indexed | `LDA $1000,Y` | `lda.y $1000` |
| Absolute indirect | `JMP ($1000)` | `jmp.i $1000` |
| Indexed indirect | `JMP ($1000,X)` | `jmp.xi $1000` |
| Absolute long | `JMP $101000` | `jmp.l $101000` |
| Absolute long X indexed | `JMP $101000,X` | `jmp.lx $101000` |
| Absolute indirect long | `JMP [$1000]` | `jmp.il $1000` |
| Direct page (DP) | `LDA $10` | `lda.z $10` |
| Direct page X indexed | `LDA $10,X` | `lda.zx $10` |
| Direct page Y indexed | `LDX $10,Y` | `ldx.zy $10` |
| Direct page indirect | `LDA ($10)` | `lda.zi $10` |
| DP indirect X indexed | `LDA ($10,X)` | `lda.zxi $10` |
| DP indirect Y indexed | `LDA ($10),Y` | `lda.ziy $10` |
| DP indirect long | `LDA [$10]` | `lda.zil $10` |
| DP indirect long Y index | `LDA [$10],Y` | `lda.zily $10` |
| Relative | `BRA <LABEL>` | `bra <LABEL>` |
| Relative long | `BRL <LABEL>` | `bra.l <LABEL>` |
| Stack relative | `LDA 3,S` | `lda.s 3` |
| Stack rel ind Y indexed | `LDA (3,S),Y` | `lda.siy 3` |
| Block move | `MVP 0,0` | `mvp 0 0` |

## Code example

We can compare the two notations with a 6502 code fragment:

```
        LDA #00                 lda.# 00
        STA $10                 sta.z $10
        TAX                     tax
LOOP1:                  loop1:
        STA $1000,x             sta.x $1000
        DEX                     dex
        BNE LOOP1               bne loop1
```

If the changes seem minor, remember this is intentional: The code must remain readable for people not familiar with the new syntax.

# Modified mnemonics for the 65816

The changes to the 65816 mnemonics mainly involve defining a common stem and adding suffixes instead of creating new mnemonics as in the traditional variant.

| | | |
|---|---|---|
| `BRL` | `bra.l` | Branch long |
| `JML` | `jmp.l` | Jump absolute long |
| `JSL` | `jsr.l` | Jump subroutine long |
| `PEA` | `phe.#` | Push effective absolute address |
| `PEI` | `phe.d` | Push effective indirect address |
| `PER` | `phe.r` | Push effective relative address |
| `RTL` | `rts.l` | Return subroutine long |

The only really difficult one is `phe.d`, which reflects the actual workings of the instruction better than the misleading "indirect" description.

Note we have not added suffixes for new suffixes' sake. The relative branches could have received a `r` tail in keeping with `phe.r`, but the short form is more familiar, and there is no other addressing mode for the branch instructions anyway. Also, a more complete systematic revision of the opcodes might also suggest `CMP` should be changed into `cpa` in keeping with `cpx` and `cpy`. However, `cmp` is the far more familiar form. The same holds true for `INC A` vs `INA`, which is why we stick with `inc.a`.

# Disadvantages of the new notation

Apart from the obvious initial unfamiliarity, both the listing of all addresses and the small code fragment show that we give up the column formatting always present in three-letter operands.

Adding a tail expands some of the lesser-used mnemonics to a ridiculous length, such as `lda.dily` — in this case, the suffix wags the stem, so to speak.

Using `z` for direct mode on the 65816 is 6502/65c02 makes using code from one on processor on the other harder.

# Suggested conventions for assemblers

While we're at it, we might as well define a set of conventions for assemblers. Realistically, these are notes for myself.

## Number formatting

Traditionally, `$` has been used to designate hexadecimal and `%` binary numbers. This convention is too strong to be changed, though `0x` is the more common marker for hexadecimal numbers with modern computer languages. It should be accepted by assemblers for SAN.

As syntactic sugar, a colon `:` should be legal in 24 bit numbers to separate the bank byte from the rest of the address (e.g. `$10:0000`)

## Directives

Directives should all start with a dot (`.`) and be the first word on the line.

## Comments

Comments begin with `;` on the line.

Since `(` and `)` are not used for the mnemonics any more, this opens the door to using them for Forth-like in-line comments. However, there seems to be little use for them in Assembler.

## Indentation

Indentation is handled by spaces, not tabs, with eight spaces per indentation level. Labels and high-level comments start at the beginning of the line, directives on indentation in, and instructions two indentations in.

## Postfix notation math

Since the `[` and `]` symbols are not used for the mnemonics any more, they can be used to designate a math term in postfix notation during assemble time. Postfix math is easier to adapt to miniature stacks for primitive assemblers to run on a 6502/65c02/65816 system itself. Example: `[ 2 3 + ]`

# Tools for SAN

Unsurprisingly, there are currently few tools for SAN.

## Assemblers for SAN

The disassembler of Tali Forth 2 for the 65c02 (https://github.com/scotws/TaliForth2) outputs SAN. An assembler is being worked on that will accept SAN notation.

### Assemblers for TAN

Typist's Assembler Notation (TAN) was an early version of SAN (see below). There are a number of assemblers for TAN that should be easy to port to SAN, as the code is almost identical:

- A Tinkerer's Assembler for the 6502/65c02/65816 in Python (https://github.com/scotws/tinkasm)
- A Typist's Assembler for the 65c02 CPU in Forth (https://github.com/scotws/tasm65c02)
- A Typist's Assembler for the 65816 CPU in Forth (https://github.com/scotws/tasm65816)

The last two use postfix notation.

Liara Forth for the 65816 was written in Typists' Assembler Notation (TAN), a proto-version of SAN (https://github.com/scotws/LiaraForth).

## Editor plugins

There is a vim plugin for TAN which will be converted to SAN and included in this repository. See https://github.com/scotws/Typist-VIM-Syntax for the original plugin.

## Automatic conversion

A program for automatic conversion from SAN to traditional assembler for the Ophis assembler is planned.

There is a primitive tool to aid in converting traditional format to TAN named typ65conv (https://github.com/scotws/type65conv). Currently, it converts instructions only, not directives.

## Formatting aids

The Go (golang) language introduced the principle of having all formatting handled by a tool (`gofmt`) that provides the official variant. Such a tool is planned for SAN once the specification is stable.

# Appendix

# Earlier versions

The first reworking of the traditional syntax resulted in Typist's Assembler Notation (TAN). It included more radical — in the end, too radical — departures when dealing with the operand, for instance using hex numbers as the default. It also defined separate suffixes for Direct mode on the 65816 and Zero Page on the 6502/65c02. The parts were called "body" and "tail" instead of "stem" and "suffix". See https://docs.google.com/document/d/16Sv3Y-3rHPXyxT1J3zLBVq4reSPYtY2G6OSojNTm4SQ for the specification. TAN is deprecated.

# Postfix variants (Forth)

Reducing the operands to pure parameter data makes it easy to use the same system with postfix formatted assemblers such as those written in Forth: Opcode and operand merely have to be switched. This way, `lda.# 10` becomes `10 lda.#` with no further modifications necessary. Since there is a one-to-one relationship between mnemonics and opcodes, the mnemonics can be simply defined as Forth words.

Forward jumps and branches are a bit more complicated, because internally the assembler has to create a list of unresolved "future symbol" references until the actual location of the label is known. This is a well-defined problem with single-pass assemblers. In these cases, we deal with the problem by assigning special directives to the unresolved labels.

```
            <j frog jsr
           <jl frog jsr.l
            <b dogs bra

   -> dogs           brk
   -> frog           inc.a
               dogs bra
                     rts
```

There are four directives to use with forward references in single-pass postfix assemblers:

| | |
|---|---|
| `<j` | jump absolute (for `jmp`, `jsr`); 2 byte operand |
| `<jl` | jump absolute long (for `jmp.l`, `jsr.l`); 3 byte operand |
| `<b` | branch relative (for `bra`, `bne`, etc); 1 byte operand |
| `<bl` | branch relative long (for `bra.l`); 2 byte operand |

In this version → is used as a label directive. === Further information and help

For all things to do with the 6502/65c02/65816, see http://www.6502.org/ Very nice, very helpful people.

# References and further reading

[ENL] *Programming the 65816. Including the 6502, 65C02 and 65802*, David Eyes and Ron Lichty

[SWS] "Typist's Assembler Notation. An Alternative Syntax for the 6502, 65C02, and 65816 MPUs", Scot W. Stevenson, Dec 2016, https://docs.google.com/document/d/16Sv3Y-3rHPXyxT1J3zLBVq4reSPYtY2G6OSojNTm4SQ/

[6500] "MCS6500 Microcomputer Family Programming Manual", 2nd edition, Jan 1976, http://wdc65xx.com/wp-content/uploads/2013/07/6500-50A_MCS6500pgmManJan76.pdf

[PJ] Microchess source code listing, Peter Jennings, 1976 http://users.telenet.be/kim1-6502/microchess/microchess.html

[RH] "A Proposed Assembly Language Syntax For 65c816 Assemblers", Randall Hyde http://fms.komkon.org/comp/CPUs/65816.1.txt

[DS] *Assemblers And Loaders*, David Salomon, 1993, http://www.davidsalomon.name/assem.advertis/AssemAd.html