asm341.py Documentation

asm341.py is an assembler for an assembly-like language for the 4-bit microcontroller developed in the assignments in CME341 at the University of Saskatchewan, written by Dylan Remus. It takes in a text file containing a program written in what I call 341 assembly, and outputs a .hex file containing the assembled machine code, which is ready to import into your program data in your Quartus microcontroller project and flashed onto your FPGA.

Syntax: “python asm341.py <input file>” or “python asm341.py <input file> <output file>”

If omitted from the command, the output file name defaults to out.hex.

341 Assembly Language Specification

341 Assembly (or 341 asm for short) is a language designed to be somewhat similar to other flavours of assembly, like x86 assembly or ARM assembly. Each line is of the form

<opcode> <param 1> <param 2> ... <param n> ; <comment>

where different opcodes accept different numbers of parameters, and where the comment field is optional. The semicolon is not necessary if the comment is omitted. While not mandatory, it is recommended that lines that contain assembler directives do not have leading spaces, and that lines that contain instructions have 2 to 4 spaces before the opcode. This is for the sake of code readability and ease of identifying block boundaries.

For a list of all accepted opcodes, their required parameters, and their functions, see the Opcodes section.

As well, asm341.py includes support for several different assembler directives. For more info, see the Assembler Directives section.

You can have a maximum of one instruction or assembler directive per line.

Because the target microcontroller only supports jumping to addresses which end in zero, it isn’t very useful to be thinking in terms of addresses or labels when writing in 341 asm. Instead, 341

asm programs are split into blocks. Each block is 16 instructions long, and there are 16 blocks in a program. Each instruction takes one byte, so in total a 341 asm program takes up 256 bytes of storage. If program execution reaches the end of a block, it moves to the next block automatically. If a program contains more than 16 instructions in a row without declaring the beginning of a new block, the excess instructions are automatically put into the next block. Care must be taken to ensure code written early in the assembly process is not overwritten later if code is allowed to overflow block boundaries. See the programs in the included Examples folder for examples of good and bad code.

Assembler Directives

Currently, asm341.py supports 3 assembler directives. More may be added in future if there is demand. The currently supported directives are .define, .undef, and .block.

.define

The .define directive is used to declare macros for code substitution, similar to #define in C-like languages. It currently only supports object-like macros (directly replacing any instances of a word with another). Function-like macros are unavailable.

Syntax:

.define <macro name> <replacement string>

For example. ".define foo bar" will replace all instances of the word "foo" with the word "bar" following the directive until a corresponding .undef directive. If a macro appears as part of a larger word, it is not expanded. For example. "foobar" would not be turned into "barbar", and instead would remain as "foobar".

Assembler directives and their arguments are never expanded, even if they match a macro.

For examples of .define in a program, see the Examples folder included with this document.

.undef

The .undef directive undoes a prior .define directive.

Syntax:

.undef <macro name>

For example, ".undef foo" undoes the .define written in the previous section. Once this directive is invoked, any instances of "foo" later in the file are no longer replaced with "bar".

Note that if this directive is invoked and its argument has not been previously .define'd, an assembler error occurs and the assembly cancels. For example, if using the code from the previous section, using ".undef xyz" would be an error and would cancel assembly.

.block

The .block directive causes the next instruction to be located in memory at the beginning of the given block.

Syntax:

.block <block number>

For example, .block 2 causes the next instruction to be located at the beginning of block 2. This is useful as a replacement for labels in other flavours of assembly, as in 341 asm you can only jump to block boundaries. As well, when there are too many instructions to fit in a block, a warning is shown. This directive can be used to suppress that warning.

Blocks do not have to be declared in order. For example,

.block 5
(code)
.block 4
(more code)

is entirely valid. However, one must be careful that the code intended to go in block 4 is less than 16 instructions long, as if it is longer then instructions in block 5 will be overwritten. For an example of this, and of other usages of .block, see the Examples folder included with this document.

Registers

The target microcontroller has 8 registers available for writing by the program. These registers are x0, x1, y0, y1, r, o_reg, i, m, and dm.

The x0, x1, y0, and y1 registers are general 4-bit data registers, and are also used as the operands for math operations.

The r register is where the result of a math operation is stored. This register is read only. If a write is attempted, the write is redirected to the o_reg register.

The o_reg register is the output of the microcontroller. This register is write only. If a read is attempted, the read is redirected to the r register.

The i register is the address that is used when reading or writing to RAM. This register is automatically incremented by the value in the m register whenever memory is accessed.

The m register contains the value that is automatically added to the i register after memory access.

The dm register is the RAM. By setting the i register to the desired address, a program can read or write to RAM by reading or writing this register.

Opcodes

341 asm has support for 13 different opcodes, each mapping to a different hardware instruction. These instructions are ld, mov, jmp, jnz, neg, not, add, sub, muh, mul, xor, and, and nop.

ld:

Syntax: ld <destination register> <value>

This instruction loads the given 4-bit hexadecimal number in value into destination register.

mov:

Syntax: mov <destination register> <source register>

This instruction copies the data in source register and puts it into destination register. The only exceptions are for when destination register is r, source register is o_reg, or when the destination and source are the same.

As r is read only and o_reg is write only, reading o_reg is interpreted as a read from r and a write to r is interpreted as a write to o_reg.

When the source and destination are the same, instead of doing nothing this instruction moves data from the input of the microcontroller to the given register. A program can also use “mov <destination register> i_pins” to achieve the same effect. Note that using i_pins as a register is only valid when it is the source register for the mov instruction. Using i_pins in any other instance is an assembler error and will cancel assembly early.

Jmp:

Syntax: jmp <block number>

This instruction causes execution to move to the beginning of the given block. The block number must be given in hexadecimal.

jnz:

Syntax: jnz <block number>

This instruction causes execution to move to the beginning of the given block, but only if the last math operation did not have an answer of zero. The block number must be given in hexadecimal.

neg:

Syntax: neg <x0 or x1>

This instruction takes the two's complement of the number in the x0 or x1 register, depending on the parameter, and stores the result in the r register. In other words, this instruction inverts the value in the given register, adds one, and stores the result in the r register.

not:

Syntax: not <x0 or x1>

This instruction takes the bitwise not of the number in the x0 or x1 register, depending on the parameter, and stores the result in the r register.

add:

Syntax: add <x0 or x1> <y0 or y1>

This instruction adds the values in the two chosen registers, and stores the result in the r register.

sub:

Syntax: sub <x0 or x1> <y0 or y1>

This instruction subtracts the values in the two chosen registers, and stores the result in the r register.

muh:

Syntax: muh <x0 or x1> <y0 or y1>

This instruction multiplies the values in the two chosen registers, and stores the high four bits of the result in the r register.

mul:

Syntax: mul <x0 or x1> <y0 or y1>

This instruction multiplies the values in the two chosen registers, and stores the low four bits of the result in the r register.

xor:

Syntax: xor <x0 or x1> <y0 or y1>

This instruction takes the bitwise xor of the values in the two chosen registers, and stores the result in the r register.

and:

Syntax: and <x0 or x1> <y0 or y1>

This instruction takes the bitwise and of the values in the two chosen registers, and stores the result in the r register.

nop:

Syntax: nop

This instruction does nothing. It may be overwritten in future.

Support

If you find a bug, need help, or have a feature request, please file an issue on the GitHub repository for this project.

www.github.com/dmrem/asm341