Structured Programming in Z80 Assembly

John Hardy Dec 18 '19 *Updated on Dec 26, 2019* · 9 min read

#z80 #macros #assembler #structuredprogramming

Structured Programming in Z80 Assembly

One of the great pains of writing assembly language for old-school microprocessors such as the Z80 is the complexity of implementing algorithms due to the lack of high-level control and looping structures. All you have are conditional jumps and labels and nothing to help you enforce the structure of your code.

It's not exaggerating when someone claims that GOTOs are considered harmful ...at least to your state of mental well-being! ;-)

This lack of structure is what often ends up driving programmers in the direction of high-level languages such as C which people often think of as "low level", "assembler++" or "close to the metal". On 8-bit CPUs, however, this far from the truth and C adds a lot of overhead to your machine-cycle and memory-cell constrained code. This is why assembly language is still the tool of choice for 8-bit programming despite it also being a major source of frustration.

Macros are a huge boon to writing assembly language. Recently I starting using a set inspired by a coding pattern invented by Garth Wilson and (quite separately by) Dave Keenan which enabled me to write structured programs in assembly.

See:

Program Structures with Macros

Adding Structured Control Flow to Any assembler

Both authors were heavily influenced by the Forth programming language and the way that it introduced high-level structured programming concepts to low level programming years before systems languages like C and Pascal became commonplace.

The examples I'm giving here were written using the asm80 macro system but I'm sure they could be easily adapted to your own favourite assembly's macro syntax.

See:

Asm80

Asm80: Macros

Create a stack with assembler variables.

Programming structures are recursive by nature and so it stands to reason that our first task should be to build a stack. Assemblers weren't really designed for meta-programming but you can implement a stack structure by using a bucket brigade of re-assignable assembler variables. I've made my stack twelve levels deep but you may decide to make this stack deeper which is an easy thing to do.

```
STRUC_COUNT .set 0

STRUC_12 .set 0
STRUC_11 .set 0
```

```
STRUC_10 .set 0
STRUC_9 .set 0
STRUC_8 .set 0
STRUC_7 .set 0
STRUC_6 .set 0
STRUC_5 .set 0
STRUC_4 .set 0
STRUC_3 .set 0
STRUC_2 .set 0
STRUC_TOP .set 0

.macro STRUC_PUSH, arg
    STRUC_COUNT .set STRUC_COUNT + 1

    STRUC_12 .set STRUC_11
    STRUC_11 .set STRUC_10
    STRUC_10 .set STRUC_9
    STRUC_9 .set STRUC_8
    STRUC_8 .set STRUC_7
    STRUC_7 .set STRUC_6
    STRUC_6 .set STRUC_5
    STRUC_5 .set STRUC_4
    STRUC_4 .set STRUC_3
    STRUC_3 .set STRUC_2
    STRUC_2 .set STRUC_TOP
    STRUC_TOP .set arg
.endm

.macro STRUC_POP
    STRUC_COUNT .set STRUC_COUNT - 1

    STRUC_TOP .set STRUC_2
    STRUC_2 .set STRUC_3
    STRUC_3 .set STRUC_4
    STRUC_4 .set STRUC_5
    STRUC_5 .set STRUC_6
    STRUC_6 .set STRUC_7
    STRUC_7 .set STRUC_8
    STRUC_8 .set STRUC_9
    STRUC_9 .set STRUC_10
    STRUC_10 .set STRUC_11
    STRUC_11 .set STRUC_12
    STRUC_12 .set 0
.endm
```

Also we need a utility macro `JUMP_FWD` with which we can use to go back and rewrite jump addresses that we can't resolve until later.

```
.macro JUMP_FWD
    CUR_ADR .set $
    org STRUC_TOP - 2
    dw CUR_ADR
    org CUR_ADR
.endm
```

Implement structured programming macros:

In assembly language, program logic is obscured by the way it gets expressed in terms of state flags and branches. Consider this simple example of inverting a binary value which we'll first express by using a structured language:

```
let a = input;
if (a == 0) {
  a = 1;
} else {
  a = 0;
}
let result = a;
```

In assembly language, we could do the same thing by loading the accumulator with the input and comparing it with zero. This may or may not set the zero flag. If the flag is false (test fails) we conditionally jump over the code in the "then" section and go straight to the "else" section. If the test succeeds we execute the "then" section but now we want to skip over the "else" section with an unconditional branch to the end of the if statement (i.e. to "endif").

```
    ld a,(input)
    cp 0
    jp nz, elseLabel
thenLabel:
    ld a, 1
    jp endIfLabel
elseLabel:
    ld a, 0
endIfLabel:
    ld (result),a
```

So in a typical situation, we have two branches and at least two labels (the thenLabel is not really needed in this case). When logic gets nested, this can lead to difficult to read assembly code. Surely a more natural way to express this code would be a little more like this?

```
ld a,(input)
cp 0
_if z
  ld a, 1
_else
  ld a, 0
_endif
ld (result),a
```

In this code we are using macros. I am using an underscore in front to distinguish these macros from other uses of these words. You'll notice that we have no explicit branches and no labels. Each `_if`, `_else` and `_endif` macro expands out into code that is very similar to the hand-written assembly code.

```
.macro _if, flag
    jr flag, L_%%M
    jp $              ; placeholder jump to _else or _endif
```

```
    STRUC_PUSH $
L_%%M:
.endm

.macro _else
    jp $              ; placeholder jump to _endif
    JUMP_FWD
    STRUC_TOP .set $  ;reuse top of stack
.endm

.macro _endif
    JUMP_FWD
    STRUC_POP
.endm
```

Note: `L_%%M` is a local label which is unique to each macro invocation. \$ is the current assembler address.

The way this works is the `_if` sets up a jump on condition flag which, if the test fails, branches to an `_else` or the `_endif` depending on which occurs first.

The problem to solve is that `_if` cannot know where the `_else` or `_endif` will occur so it writes the jump instruction with a placeholder for the jump address. It then pushes the address of this jump on the assembler stack so it can be found again later.

When an `_else` is encountered, it looks on the stack for the address of the last `_if` occurrence. It goes back and fills in the placeholder address in the `_if` to point to the `_else` code. It then writes another jump instruction with a placeholder address to point to the `_endif`. Once again, it pushes the address of this jump onto the assembler stack.

When an `_endif` is encountered, it looks on the stack for the address of the last jump instruction pushed on the stack (it will be either an `_if` or `_else` occurrence). Then it goes back and fills in the placeholder address of the jump instruction to point to itself. **This means that the** `_else`

macro is completely optional and that you could just use `_if` and `_endif` without an `_else` if you wanted to. For example:

```
ld a,(input)
cp 0
_if z
  ld a, 1
_endif
ld (result),a
```

This rewriting magic is achieved by the `JUMP_FWD` macro which saves the current assembler address `$` in a temporary variable and then uses the `org` directive to move the current assembler address back towards the code it has already written. By backing up the assembler's address pointer it can then rewrite the branch placeholder address to the value of the saved value. It then restores from the saved assembler address and continues.

It will probably take you a few reads through to fully understand this logic, it's fiddly but not too complicated. If you are already familiar with the way the Forth language implements its control structures then you may grasp it immediately.

Now, with these macros in hand, you can easily write nested if...else...endif logic in your Assembly code without using a single jump or even a label!

```
ld A,1
or a     ; test for the zero condition just before the _if
_if z
    cmp $10  ; test for the not carry condition just before the _if
    _if nc
        nop
        nop
    _else
```

```
        nop
        nop
        nop
    _endif
_else
    nop
    nop
    nop
_endif
```

Note: I'm using `nop` here to stand in for any Z80 instruction

Avoiding deep nesting with Switch

When you have a lot of alternatives to deal with in your conditional logic, `_if` ... `_else` ... `_endif` can quickly become cumbersome and hard to read. For example, consider the following scenario in a structured language:

```
let a = input
if (a == 'a') {
    a = 'A';
} elseif (a = 'b') {
    a = 'B';
} elseif (a = 'c') {
    a = 'C';
} else {
    a = 'D';
}
```

Without some way to chain these "if" statements we end up with a heavily nested sequence like this:

```
ld A, input
cp 'a'
_if
```

```
    ld A,'A'
_else
    cp 'b'
    _if
        ld A,'B'
    _else
        cp 'c'
        _if
            ld A,'C'
        _else
            ld A,'D'
        _endif
    _endif
_endif
```

This is pretty ugly but you can solve this nesting by using the `_switch` macro.

```
ld A, input
_switch

    cp 'a'          ; test for 'a'
    _case z
        ld A,'A'
    _endcase

    cp 'b'          ; test for 'b'
    _case z
        ld A,'B'
    _endcase

    cp 'c'          ; test for 'c'
    _case z
        ld A,'C'
    _endcase

    ld A,'D'        ; default case

_endswitch
```

The way `_switch` works is that each case is tested in turn and, if a condition is met, the `_case` immediately following the test is executed. After that it jumps to the `_endswitch`.

If the condition fails it falls through to the next case and so on. If none of these cases execute then it falls through to the final "default" case just before the `_endswitch`.

The implementation of the switch macro is as follows:

```
.macro _switch
    jr L_%%M
    jp $            ; placeholder jump to endswitch
    STRUC_PUSH $
L_%%M:
.endm

.macro _case, flag
    jr flag, L_%%M
    jp $            ; placeholder jump to endcase
    STRUC_PUSH $
L_%%M:
.endm

.macro _endcase
    jp STRUC_2 - 3  ; jump to placeholder jump to endswitch
    JUMP_FWD
    STRUC_POP
.endm

.macro _endswitch
    JUMP_FWD
    STRUC_POP
.endm
```

Loops

Looping is another pain point for assembly language and another potential source of bugs.

Consider following code written with a structured language:

```
let a = 0;
while (a < 10) {
    ; do something here
    a++;
}
```

In conventional assembly code you would normally use a conditional branch to a label

```
    ld A, 0
LOOP1:
    cp 10
    jp nc, LOOP_EXIT
    nop                 ; do something here
    inc A
    jp LOOP1
LOOP_EXIT:
```

In this case I'm testing for the *opposite* condition to the structured version. I'm deciding whether to exit the loop rather than to stay inside it. I then do some work before incrementing the counter and jumping back to do the test.

This code with two jumps and two labels is somewhat confusing and the situation only gets more confusing when loops are nested within one another.

The structured macro way to do the same thing is to use the `_do` macro.

```
ld A, 0
_do
```

```
    cp 10           ; test
    _while c
    nop             ; do something here
    inc A
_enddo
```

The code between `_do` and `_enddo` is repeated and a test is conducted just before the `_while` which will jump to the `_enddo` if the test fails.

Sometimes it's more convenient to terminate on the success of a test.

```
ld A, 0
_do
    cp 10           ; test
    _until nc
    nop             ; do something here
    inc A
_enddo
```

You can unconditionally terminate a loop by using the `_break` macro.

An alternative to terminating a loop is to `_continue` a loop, that is, tp unconditionally jump to the start of a loop.

```
ld B, 0
_do
    ld A,B
    and $01          ; test
    _if nz
        nop
        _continue
    _endif
                     ; get here only on even values
    inc B            ; test
    _until z
_enddo
```

Note: both `_while`, `_until`, `_break` and `_continue` are optional and may appear zero or more times inside a loop.

Loops can be nested easily as long as the values of counter variables are preserved.

```
ld A, 0
_do
    cp 2      ; test
    _while z
    push AF
    ld A, 0
    _do
        cp 5   ; test
        _while z

        nop    ; do something here
        inc A
    _enddo
    pop AF
    inc A
_enddo
```

A loop can also built using the the Z80's own `djnz` instruction. This assumes that the counter value is stored in the B register which is decremented automatically each time through the loop. When B reaches zero the loop terminates.

```
ld B, 10
_do
    nop             ; do something here
_djnz
```

Note: `_while`, `_until`, `_break` and `_continue` all work inside `_do` ... `_djnz` loops exactly the same way as they do in `_do` ... `_enddo` loops.

The implementation of macros for looping are as follows:

```
.macro _do
    jr L_%%M
    jp $                    ; placeholder jump to enddo
    STRUC_PUSH $
L_%%M:
.endm

.macro _while, flag
    jr flag L_%%M
    jp STRUC_TOP - 3        ; jump to jump to enddo
    JUMP_FWD
L_%%M:
.endm

.macro _until, flag
    jp flag, STRUC_TOP - 3  ; jump to jump to enddo
    JUMP_FWD
.endm

.macro _continue
    jp STRUC_TOP            ; start of loop
.endm

.macro _break
    jp STRUC_TOP - 3        ; jump to jump to enddo
    JUMP_FWD
.endm

.macro _enddo
    jp STRUC_TOP
    JUMP_FWD
    STRUC_POP
.endm

.macro _djnz
    djnz STRUC_TOP
    JUMP_FWD
```

```
    STRUC_POP
.endm
```

So there you have it, a pretty painless way to improve the readability of your code and increase your productivity as an assembly language programmer. The best thing is that if you examine the generated assembly code you'll see that it doesn't look weird or add overhead to the way you might have written this code natively.

Anyway, if you do give it a try let me know how it goes!

A repo of all the macros discussed here can be found here:

struct-z80

Contact me: jh@lagado.com GitHub: jhlagado Twitter: @jhlagado

John Hardy **+ FOLLOW**

@jhlagado jhlagado jhlagado github.com/jhlagado