Z80 Optimization

From WikiTI

Contents

Introduction

Sometimes it is needed some extra speed in ASM or make your game smaller to fit on the calculator. Examples: consuming graphics/data programs and graphics code of mapping, grayscale and 3D graphics.

If you are just looking for cutting some bytes go straight to small tricks in this topic.

Registers and Memory

Generally good algorithms on z80 use registers in a appropriate form. It is also a good practise to keep a convention and plan how you are going to use the registers.

General use of registers:

- a - 8-bit accumulator
- b - counter
- c,d,e,h,l auxiliary to accumulator and copy of b or a

- hl - 16-bit accumulator/pointer of a address memory
- de - pointer of a destination address memory
- bc - 16-bit counter
- ix - index register/pointer to table in memory/save copy of hl/pointer to memory when hl and de are being used
- iy - index register/pointer to table in memory (use when there is no other option or need optimal execution) (disable interrupts and on exit restore the original value because TI-OS uses)

8-bit vs. 16-bit Operations

The z80 processor makes faster operations on 8-bit values. Code dealing with 16-bit register tends to be bigger and slower because of the equivalent 16-bit instruction is slower or it does not exist and needs to be replaced with more instructions. And sometimes the equivalent 16-bit instruction is 1 more byte. If you use ix or iy registers operations are even slower and always are 1 byte bigger for each instruction. So try to convert your code to use hl and de instead of ix and iy.

In a practical example, imagine: - you pass through the accumulator a value to a routine - if the only valid values of the accumulator range from 0 to 63 and if in that routine you need to multiply the accumulator by, say 12, it has to be stored in a 16-bit pair register. - but you can multiply a by 4 before overflowing (63*4 = 252 which is smaller than 255) and take advantage of this to optimize

Now on the code:

```
; The most usual way is pass A (the accumulator) right in the start to HL
        ld h,0
        ld l,a
        add a,a
        ld d,h
        ld e,a
        add hl,de
        add hl,hl
        add hl,hl       ; hl=a*12
; 9 bytes, 56 clocks

; But given a is between 0 and 63 you can multiply by 4 without overflowing the 8-bit limit (255)
        add a,a
        add a,a         ; a*4
        ld l,a
        ld e,a
        ld h,0
        ld d,h          ; hl=a*4 and de=a*4
        add hl,hl       ; hl=a*8
        add hl,de       ; hl=a*12
; 9 bytes, 49 clocks

; Although this specific case could be even better as follows:
        ld l,a
        add a,a         ; a*2
        add a,l         ; a*3
        ld h,0
        ld l,a          ; hl=a*3
```

```
        add hl,hl       ; hl=a*6
        add hl,hl       ; hl=a*12
; 8 bytes, 45 clocks
```

In this example you both shaved a few clock cycles and saved some bytes, too. You can do this for other registers than A accumulator.

For example if passed in l and l is always lower than 64, you can do " sla l \ sla l \ ld h,0 " to multiply l by four and use hl for 16-bit operations. In this case you are exchanging size with speed increase. Each sla instruction is 2 bytes and add hl,hl is only 1 byte.

Mind this optimizations can produce bugs and somewhat hard code to follow, so comment them. I recommend to proceed to this optimization only when you really need speed and the code is bug free.

One common trick with multiplication by 256 is just load around the low byte register to the high byte register. This works because in binary a multiplication by 256 is like shifting 8 bits left, entering zeros. Examples:

```
; multiply a by 256 and store in hl
        ld h,a
        ld l,0
; multiply hl by 256 and store in ade (pseudo 24-bit pair register)
        ld a,h
        ld d,l
        ld e,0
```

If you are out of registers, try using ixh/ixl/iyh/iyl and even the i register for loop counters instead of maintaining a counter in memory or pushing/popping an already used register to the stack inside a loop. Using ixh/ixl/iyh/iyl will break compatibility with the TI-84+SE emulated by the Nspire. You can only use i register for other purposes if you disable interrupts first (di).

Shadow registers

In some rare cases, when you run out of registers and cannot to either refactor your algorithm(s) or to rely on RAM storage you may want to use the shadow registers : af', bc', de' and hl'

These registers behave like their "standard" counterparts (af, bc, de, hl) and you can swap the two register sets at using the following instructions :

```
ex af, af'  ; swaps af and af' as the mnemonic indicates

exx         ; swaps bc, de, hl and bc', de', hl'
```

Shadow registers are somewhat common for doing arithmetic operations on some big integers (16-bit to 32-bit) or BCD operations without rely on RAM storage or pushing and popping to the stack. Example:

```
MUL32:
        DI
        AND     A               ; RESET CARRY FLAG
        SBC     HL,HL           ; LOWER RESULT = 0
        EXX
        SBC     HL,HL           ; HIGHER RESULT = 0
        LD      A,B             ; MPR IS AC'BC
```

```
        LD      B,32            ; INITIALIZE LOOP COUNTER
MUL32LOOP:
        SRA     A               ; RIGHT SHIFT MPR
        RR      C
        EXX
        RR      B
        RR      C               ; LOWEST BIT INTO CARRY
        JR      NC,MUL32NOADD
        ADD     HL,DE           ; RESULT += MPD
        EXX
        ADC     HL,DE
        EXX
MUL32NOADD:
        SLA     E               ; LEFT SHIFT MPD
        RL      D
        EXX
        RL      E
        RL      D
        DJNZ    MUL32LOOP
        EXX

; RESULT IN H'L'HL
        RET
```

Shadow registers can be of a great help but they come with two drawbacks :

- they cannot coexist with the "standard" registers : you cannot use ld to assign from a standard to a shadow or vice-versa. Instead you must use nasty constructs such as :

```
; loads hl' with the contents of hl
push hl
exx
pop hl
```

- they require interrupts to be disabled since they are originally intended for use in Interrupt Service Routine. There are situations where it is affordable and others where it isn't. Regardless, it is generally a good policy to restore the previous interrupt status (enabled/disabled) upon return instead of letting it up to the caller. It's relatively easy to do (adding 5 bytes and 27/35 T-states to the routine), although this method is only reliable in CMOS Z80 CPUs (NMOS Z80 CPUs have an issue described at bottom left of page 3-130 here (http://www.z80.info/zip/ZilogProductSpecsDatabook129-143.pdf)):

```
 ld a, i  ; this is the core of the trick, it sets P/V to the value of IFF so P/V is set iff interrupts were enabled
 push af  ; save flags
 di       ; disable interrupts

 ; do something with shadow registers here

 pop af   ; get back flags
 ret po   ; po = P/V reset so in this case it means interrupts were disabled before the routine was called
 ei       ; re-enable interrupts
 ret
```

Notice that, in order to work on all Z80 CPUs (including NMOS Z80), it's necessary to check interrupt status twice within a short interval. This way, if an interrupt occurred exactly during the first test, it could cause a "false negative", but testing it again quickly before another interrupt could happen would ensure a reliable result, as follows:

```
 ld a, i  ; this is the core of the trick, it sets P/V to the value of IFF so P/V is set iff interrupts were enabled
 jp pe,label
 ld a, i  ; test again, to fix potential "false negative" from interrupt occurring at first test
label:
```

```
 push af  ; save flags
 di       ; disable interrupts

 ; do something with shadow registers here

 pop af   ; get back flags
 ret po   ; po = P/V reset so in this case it means interrupts were disabled before the routine was called
 ei       ; re-enable interrupts
 ret
```

> Note that this produces ugly and very hard code to follow, so comment it very well for understanding and debugging later.

SP register

This register is used in desperate situations generally during an interrupt loop demanding as much speed as possible and the normal registers are used. (remarkably used in James Montelongo 4 lvl grayscale interlace in graylib2.inc) You need to know these valid and not generally known instructions:

```
ld sp,6
add hl,sp
sbc hl,sp
inc sp
dec sp
```

Now an example of such situation:

```
 ld (saveSP),sp
;init hl,de,bc,a
 ld sp,6
loop:
;code
 add hl,sp  ;get next row of a table for example
;code using bc,de,ix,a
 ld a,b
 or c
 jp nz,loop:
;code
 ld sp,(saveSP)
 ret    ;finish interrupt
```

When you use sp in this way this means you can not push/pop registers and no calls are allowed. Mind again that this is only used as last resource. Don't forget to save and restore sp like the example shows.

Stack

When you run out of registers, stack may offer an interesting alternative to fixed RAM location for temporary storage.

Allocation

You can either allocate stack space with repeated push, which allows to initialize the data but restricts the allocated space to multiples of 2. An alternate way is to allocate uninitialized stack space (hl may be replaced with an index register) :

```
; allocates 7 bytes of stack space : 5 bytes, 27 T-states instead of 4 bytes, 44 T-states with 4 push which would have
ld hl, -7
add hl, sp
ld sp, hl
```

Access

The most common way of accessing data allocated on stack is to use an index register since all allocated "variables" can be accessed without having to use inc/dec but this is obviously not a strict requirement. Beware though, using stack space is not always optimal in terms of speed, depending (among other things) on your register allocation strategy :

```
; 4 bytes, 19 T-states
ld c, (ix + n)   ; n is an immediate value in -128..127

; 4 bytes, 17 T-states, destroys a
ld a, (somelocation)
ld c, a
```

If your needs go beyond simple load/store however, this method start to show its real power since it vastly simplify some operations that are complicated to do with fixed storage location (and generally screw up register in the process).

```
; 3 bytes, 19 T-states
cp (ix + n)

sub (ix + n)
sbc a, (ix + n)
add a, (ix + n)
adc a, (ix + n)

inc (ix + n)
dec (ix + n)

and (ix + n)
or (ix + n)
xor (ix + n)

; 4 bytes, 23 T-states
rl (ix + n)
rr (ix + n)
rlc (ix + n)
rrc (ix + n)
sla (ix + n)
sra (ix + n)
sll (ix + n)
srl (ix + n)
bit k, (ix + n)   ; k is an immediate value in 0..7
set k, (ix + n)
res k, (ix + n)
```

Again, choose wisely between hl and an index register depending on the structure of your data the smallest/fastest allocation solution may vary (hl equivalent instructions are generally 2 bytes smaller and 12 T-states faster but do not allow indexing so may require intermediate inc/dec).

Deallocation

If you want need to pop an entry from the stack but need to preserve all registers remember that sp can be incremented/decremented like any 16bit register :

```
; drops the top stack entry : waste 1 byte and 2 T-states but may enable better register allocation...
inc sp
inc sp
```

If you have a large amount of stack space to drop and a spare 16 bit register (hl, index, or de that you can easily swap with hl) :

```
; drop 16 bytes of stack space : 5 bytes, 27 T-states instead of 8 bytes, 80 T-states for 8 pop
ld hl, 16
add hl, sp
ld sp, hl
```

The larger the space to drop the more T-states you will save, and at some point you'll start saving space as well (beyond 8 bytes)

General Algorithms

Registers and Memory use is very important in writing concise and fast z80 code. Then comes the general optimization.

First, try to optimize the more used code in subroutines and large loops. Finding the bottleneck and solving it, is enough to many programs.

Do not forget that in z80 assembly vector tables (or look up tables) gives smaller and faster code than blocks of comparisons and jumps. Other times using a chunk of data for a task is better than a more usual programming method (notably in graphics screen effects). See Z80 Good Programming Practices for examples.

Look up in a complete instruction set for searching some instruction that can optimize somewhere in the code.

A list of things to keep in mind:

- Rework conditionals to be more efficient.
- Make sure the most common checks come first. Or said in other way, the more special and rare cases check in last.
- Get out of the main loop special cases check if they aren't needed there.
- Rearrange program flow
- When possible, if you can afford to have a bigger overhead and get code out of the main loop do it.
- When your code seems that even with optimization won't be efficient enough, try another approach or algorithm. Search other algorithms in Wikipedia, for instance.
- Rewriting code from scratch can bring new ideas (use in desperate situations because of all work needed to write it)
- Remember almost all times is better to leave optimization to the end. Optimization can bring too early headaches with crashes and debugging. And because ASM is very fast and sometimes even smaller than higher level languages, it may not be needed further optimization.
- Document wacky optimizations to understand the code later (z80 optimization leads to very hard code to understand)

Self Modifying Code

If your code is in ram, writes can be done to change the code. Having a instruction set that explains the opcodes is useful. Despite the self modifying code can be used in any instruction, it is very common with loading constants to registers.

Generally it is used to save any value to be used later (usually seen in masks). Examples:

```
 ld (savemask),a
;...code...
savemask = $+1
 ld a,$00   ; $00 is just a placeholder

 ld (something),hl
;... code
something = $+1
 ld de,$0000

 ld (saveSP),sp
;... code ...
saveSP = $+1
 ld sp,$0000  ; restore sp
```

SMC (Self Modifying Code) is quite used with unrolling and relative jumps. Example:

```
 ld (jpmodify),a
;...
jpmodify = $+1
 jr $00
 rrca
 rrca
 rrca
 rrca
 rrca
 rrca
 rrca
 rrca
```

Another SMC is modifying load instructions with (ix+0) and change the 0 to other values to really quickly read and write to the nth element of a list without using any extra registers.

Small Tricks

Note that the following tricks act much like a peep-hole optimizer and are the last optimization step : remember to first optimize your algorithm and register allocation before applying any of the following if you really want the fastest speed and the smallest code.

Also note that near every trick turn the code less understandable and documenting them is a good idea. You can easily forgot after a while without reading parts of the code.

Be warned that some tricks are not exactly equivalent to the normal way and may have exceptions on its use, comments warn about them. Some tricks apply to other cases, but again you have to be careful.

There are some tricks that are nothing more than the correct use of the available instructions on the z80. Keeping an instruction set summary, help to visualize what you can do during coding.

Optimize size and speed

Loading stuff

```
;Instead of:
 ld a,0
;Try this:
 xor a    ;disadvantages: changes flags
;or
 sub a    ;disadvantages: changes flags
; -> save 1 byte and 3 T-states
```

```
;Instead of
        ld b,$20
        ld c,$30
;try this
        ld bc,$2030
;or this
        ld bc,(b_num * 256) + c_num             ;where b_num goes to b register and c_num to c register
; -> save 1 byte and 4 T-states
```

```
;Instead of
  ld a,$42
  ld (hl),a
;try this
  ld (hl),$42
; -> save 1 byte and 4 T-states
```

```
;Instead of
        xor a
        ld (data1),a
        ld (data2),a
        ld (data3),a
        ld (data4),a
        ld (data5),a    ;if data1 to data5 are one after the other
;try this
        ld hl,data1
        ld de,data1+1
        xor a
        ld (hl),a
        ld bc,4
        ldir
; -> save 3 bytes for every ld (dataX), after passing the initial overhead
```

```
;Instead of
        ld a,(var)
        inc a
        ld (var),a
;try this       ;Note: if hl is not tied up, use indirection:
        ld hl,var
        inc (hl)
        ld a,(hl) ;if you don't need (hl) in a, delete this line
; -> save 2 bytes and 2 T-states
```

```
; Instead of :
 ld a, (hl)
 ld (de), a
 inc hl
 inc de
; Use :
 ldi
 inc bc
; -> save 1 byte and 4 T-states
```

```
;Never use:
    push BC
;    ...
    pop BC
    ld D,B
    ld E,C
;Use instead:
    push BC
;    ...
    pop DE      ;we only want to DE hold pushed BC (no need for a copy of DE in BC)
; -> save 2 bytes and 8 T-states
```

Math and Logic tricks

```
;Instead of:
 cp 0
;Use
 or a
; -> save 1 byte and 3 T-states
```

```
  cp 1
; >
  dec a   ;changes a!
; -> save 1 byte and 3 T-states
```

```
  xor %11111111
; >
  cpl
; -> save 1 byte and 3 T-states
```

```
;Instead of
    ld de,767
    or a       ;reset carry so sbc works as a sub
    sbc hl,de
;try this
    ld de,-767 ;negation of de
    add hl,de
; -> 2 bytes and 8 T-states !
```

```
;Instead of
    ld de,-767
    add hl,de
;try this
    dec h  ; -256
    dec h  ; -512
    dec h  ; -768
    inc hl  ; -767
;Note that works in many other cases
; -> save 3 T-states
```

```
;Instead of
        srl a
        srl a
        srl a
;try this
        rrca
        rrca
        rrca
```

```
        and %00011111
; -> save 1 byte and 5 T-states
```

```
;Instead of
        neg
        add a,N   ;you want to calculate N-A
;Do it this way:
        cpl
        add a,N+1    ;neg is practically equivalent to cpl \ inc a
; -> save 1 byte and 4 T-states
```

```
;Never use:
    ld A,B
    neg
;Instead use:
    xor A
    sub B
; -> save 1 byte and 4 T-states
```

```
;Never use:
    ld A,D
    sub $D3
    neg
;Instead use:
    ld A,$D3
    sub D
; -> save 2 bytes and 8 T-states
```

```
  sla l
  rl h         ; I've actually seen this!
; >
  add hl,hl
; -> save 3 bytes and 5 T-states
```

Conditionals

```
  and 1
  cp 1
  jr z,foo
; >
  and 1         ;and sets zero flag, no need for cp
  jr nz,foo
; -> save 2 bytes and 7 T-states
```

```
  and 1
  cp 1         ;a not needed after this
  jr z,foo
; >
  rra
  jr c,foo
```

```
  bit 0,a
  call z,foo
; >
  rra
  call nc,foo
```

```
  bit 7,a
  jr z,foo
; >
  rla
  jr nc,foo
```

```
  bit 2,a
  ret nz
  xor a
; >
  and %100
  ret nz
```

```
; Instead of:
  cp 9        ;if a<=9 then goto label
  jp c,label
  jp z,label

; Use this:
  cp 9+1      ;if a<10 then goto label
  jp c,label

; -> save 3 bytes and 10 T-states
```

Code Flow

Almost never call and return...

```
;Instead of
 call xxxx
 ret
;try this
 jp xxxx
;only do this if the pushed pc to stack is not passed to the call. Example: some kind of inline vputs.
; -> save 1 byte and 17 T-states
```

```
;Never use:
    dec B
    jr NZ,loop    ;I have seen this...
;Use:
    djnz loop
; save 1 byte and 3 T-states
```

Fallthrough looping

If you need to repeat a routine several times but can't spare registers for a loop counter or unroll the routine, try structuring the routine so it can call itself several times and fall through at the end. For example:

```
foo:
  ld hl, data
  call bar      ; Run routine once
  call bar      ; .. twice
  call bar      ; .. three times
bar:
  ld a, (hl)    ; .. fourth and final time
  inc l
  and $0F
```

```
  out (c), a
  ret
```

Although this specific case would be even better (same size but shorter) as follows:

```
foo:
  ld hl, data
  call bar2     ; Run routine four times
bar2:
  call bar      ; Run routine twice
bar:
  ld a, (hl)    ; Run routine once
  inc l
  and $0F
  out (c), a
  ret
```

Toggling values in loops

Consider a board game that needs to alternate between players 1 and 2 at every turn:

```
;Instead of
 ld a,(hl)      ; a=1 or 2
 inc a          ; a=2 or 3
 cp 3
 jr nz,label
 ld a,1         ; a=2 or 1
label:
; 8 bytes, 30 or 32 clocks

;Better
 ld a,(hl)      ; a=1 or 2
 dec a          ; a=0 or 1
 jr nz,label
 ld a,2         ; a=2 or 1
label:
; 6 bytes, 23 or 23 clocks

;Even better
 ld a,(hl)      ; a=1 or 2
 cpl            ; a=-2 or -3
 add a,4        ; a=2 or 1, same as calculating 3-a
; 4 bytes, 18 clocks

;Best
 ld a,(hl)      ; a=1 or 2
 xor 3          ; a=2 or 1
; 3 bytes, 14 clocks
```

The trick is xor logic make a register alternate between two values.

Look up Table

```
; Instead of
 ld a,(Number)
 cp 0
 jp z,A_is_0
 cp 1
 jp z,A_is_1
 cp 2
 jp z,A_is_2
 cp 3
 jp z,A_is_3
 cp 4
```

```
 jp z,A_is_4
 cp 5
 jp z,A_is_5

; This is a little better
 ld a,(Number)
 or a
 jp z,A_is_0
 dec a
 jp z,A_is_1
 dec a
 jp z,A_is_2
 dec a
 jp z,A_is_3
 dec a
 jp z,A_is_4
 dec a
 jp z,A_is_5

; Even better
 ld a,(Number)
 add a,a   ; a*2 (limits Number to 128)
 ld h,0
 ld l,a
 ld de,VectorTable
 add hl,de
 ld a,(hl)
 inc hl
 ld h,(hl)
 ld l,a
 jp (hl)
VectorTable:
 .dw A_is_1
 .dw A_is_2
 .dw A_is_3
 .dw A_is_4
 .dw A_is_5

; Best
 ld a,(Number)
 add a,a   ; a*2 (limits Number to 128)
 add a,VectorTable%256
 ld l,a
 adc a,VectorTable/256
 sub l
 ld h,a
 ld a,(hl)
 inc hl
 ld h,(hl)
 ld l,a
 jp (hl)
VectorTable:
 .dw A_is_1
 .dw A_is_2
 .dw A_is_3
 .dw A_is_4
 .dw A_is_5
```

If you use an aligned table (see section "Table Alignment" below), this code can be optimized even further:

```
; Using 256-byte table alignment
 ld a,(Number)
 add a,a   ; a*2 (limits Number to 128)
 ld (addr+1),a
addr:
 ld hl,(VectorTable)
 jp (hl)
VectorTable:
 .dw A_is_1
 .dw A_is_2
 .dw A_is_3
 .dw A_is_4
 .dw A_is_5
```

Also see Z80 Good Programming Practices

Size vs. Speed

The classical problem of optimization in computer programming, Z80 is no exception. In ASM most frequently size is what matters because generally ASM is fast enough and it is nice to give a user a smaller program that doesn't use up most RAM memory.

For the sake of size

- Use relative jumps (jr label) whenever possible. When relative jump is out of reach (out of -128 to 127 bytes) and there is a jp near, do a relative jump to the absolute one. Example:

```
;lots of code (more that 128 bytes worth of code)
somelabel2:
 jp somelabel
;less than 128 bytes
 jr somelabel2   ;instead of a absolute jump directly to somelabel, jump to a jump to somelabel.
```

- Relative jumps are 2 bytes and absolute jumps 3. In terms of speed jp is faster when a jump occurs (10 T-states) and jr is faster when it doesn't occur.

```
;Instead of
 dec bc
 ld a,b
 or c
 ret z
;try this
 cpi              ;increments HL
 ret po
; save 1 byte at the cost of 2 T-states
```

Passing inline data

When you call, the pc + 3 (after the call) is pushed. You can pop it and use as a pointer to data. A very nifty use is with strings. To return, pass the data and jp (hl).

```
Instead of:
 ld hl,string
 bcall(_vputs)
 ret
;Try this:
  call Disp
  .db "This is some text",0
  ret
;Not a speed optimization, but it eliminates 2-byte pointers, since it just uses the call's return address.
;It also heavily disturbs disassembly.
Disp:
  pop hl
  bcall(_vputs)
  jp (hl)
; -> save 2 bytes for each use, but 4 bytes of overhead (Disp routine)
```

This routine can be expanded to pass the coordinates where the text should appear.

Wasting time to delay

There are those funny times that you need some delay between operations like reads/writes to ports ***and there is nothing useful to do***. And because nop's are not very size friendly, think of other slower but smaller instructions. Example:

```
;Instead of
 ld a,KEY_GROUP
 out (1),a
 nop
 nop
 in a,(1)
;Try this:
 ld a,KEY_GROUP
 out (1),a
 ld a,(de)    ;a doesn't need to be preserved because it will hold what the port has.
 in a,(1)
; -> save 1 byte and 1 T-state (well 1 T-state less is almost the same time)
```

When you need to delay and cannot afford to alter registers or flags there are still ways to delay that waste less size than nop's :

```
; 2 bytes, 8 T-states
 nop
 nop

; 2 bytes, 12 T-states
 inc hl
 dec hl

; 2 bytes, 12 T-states
 jr $+2

; 2 bytes, 21 T-states
 push af
 pop af

; 2 bytes, 38 T-states
 ex (sp), hl
 ex (sp), hl
```

If you need a small adjustable delay:

```
;4 bytes, b*13+2 T-states (variable)
        ld b,255        ; initial delay
        djnz $          ; do it
;b=0 on exit
```

Notes:

- There are many other instructions that you can use
- Beware that not all instructions preserve registers or flags
- For delay between frames of games or other longer delays, you can use the 'halt' instruction if there are interrupts enabled. It make the calculator enter low power mode until an interrupt is triggered. To fine-tune the effect of this delay mechanism you can alter interrupt mask and interrupt time speed beforehand (and possibly restore their values afterwards).

Unrolling code

General Unrolling You can unroll some loop several times instead of looping, this is used frequently on math routines of multiplication. This means you are wasting memory to gain speed. Most times you are preferring size to speed.

Unroll commands

```
; "Classic" way : ~21 T-states per byte copied
 ld hl,src
 ld de,dest
 ld bc,size
 ldir

; Unrolled : (16 * size + 10) / n -> ~18 T-states per byte copied when unrolling 8 times
 ld hl,src
 ld de,dest
 ld bc,size  ; if the size is not a multiple of the number of unrolled ldi then a small trick must be used to jump app
loopldi:    ;you can use this entry for a call
 ldi
 ldi
 ldi
 ldi
 ldi
 ldi
 ldi
 ldi
 jp pe, loopldi    ; jp used as it is faster and in the case of a loop unrolling we assume speed matters more than siz
; ret if this is a subroutine and use the unrolled ldi's with a call.
```

This unroll of ldi also works with outi and ldr.

Looping with 16 bit counter

There are two ways to make loops with a 16bit counter :

- the naive one, which results in smaller code but increased loop overhead (24 * n T-states) and destroys a

```
  ld bc, ...
loop:
  ; loop body here

  dec bc
  ld  a, b
  or  c
  jp  nz,loop
```

- the slightly trickier one, which takes a couple more bytes but has a much lower overhead (approximately 13 * n + 9 * (n / 256) T-states)

```
  ld  b, e
  dec  de
  inc  d
loop2:
  ; loop body here

  djnz loop2
  dec  d
  jp  nz,loop2
```

The rationale behind the second method is to reduce the overhead of the "inner" loop as much as possible and to use the fact that when b gets down to zero it will be treated as 256 by djnz.

You can therefore use the following macros for setting proper values of 8bit loop counters given a 16bit counter in case you want to do the conversion at compile time :

```
 #define inner_counter8(counter16) ((counter16) & 0xff)
 #define outer_counter8(counter16) (((counter16) - 1) >> 8) + 1
```

Preserve Registers

```
; description: both routines compare b to 0, same size and speed but the second preserves accumulator
; remarks: - inc/dec doesn't affect carry flag
;          - inc/dec doesn't affect any flags on 16-bit registers, so do not extrapolate to 16-bit registers.
        ld a,b
        or b
        jr z,label
; >
        inc b
        dec b
        jr z,label
```

```
; description: add a to hl without using a 16-bit register
;normal way:
        ld d,$00
        ld e,a
        add hl,de
;4 bytes and 22 clock cycles
; >
        add a,l
        ld l,a
        jr nc, $+3
        inc h
;5 bytes, 19/20 clock cycles
```

Setting flags

In some occasion you might want to selectively set/reset a flag.

Here are the most common uses :

```
; set Carry flag
 scf

; reset Carry flag (alters Sign and Zero flags as defined)
 or a

; alternate reset Carry flag (alters Sign and Zero flags as defined)
 and a

; set Zero flag (resets Carry flag, alters Sign flag as defined)
 cp a

; reset Zero flag (alters a, reset Carry flag, alters Sign flag as defined)
 or 1

; set Sign flag (negative) (alters a, reset Zero and Carry flags)
 or $80

; reset Sign flag (positive) (set a to zero, set Zero flag, reset Carry flag)
```

```
 xor a
```

Other possible uses (much rarer) :

```
;Set parity/overflow (even):
 xor a

;Reset parity/overflow (odd):
 sub a

;Set half carry (hardly ever useful but still...)
 and a

;Reset half carry (hardly ever useful but still...)
 or a

;Set bit 5 of f:
 or %00100000
```

As you can see these are extremely simple, small and fast ways to alter flags which make them interesting as output of routines to indicate error/success or other status bits that do not require a full register.

Were you to use this, remember that these flag (re)setting tricks frequently overlap so if you need a special combination of flags it might require slightly more elaborate tricks. As a rule of a thumb, always alter the carry last in such cases because the scf and ccf instructions do not have side effects.

More advance ways of manipulating flags follow:

```
;get the zero flag in carry
        scf
        jr z,$+3
        ccf

;Put carry flag into zero flag.
        ccf
        sbc a, a
```

Tools of the job

Want to try test your optimization or test new ones? Then you have to check this:

- Keep a z80 instruction set to not forget a useful instruction and flags affected. (see Z80_Instruction_Set)
- Use an assembler that has ".echo" directive and use this in the source to count size: (see Assemblers)

```
SomeCodeorData:
;code or data goes here
End:
 .echo "size of the code/data:"
 .echo End-SomeCodeorData
```

- Get a nice IDE of z80 that counts code (IDE's)
- Make use of the counting capabilities of an emulator (Emulators) (see wabbitemu)

Table alignment

Indexing aligned tables

If you align tables to a 256-byte boundary, you can access the contents by placing the index in a register such as l and the table address in h. This is faster than loading the full unaligned 16-bit address and adding a 16-bit index to it, and makes accessing tables with a size of 256 bytes or less very convenient:

```
; With 256-byte table alignment
 ld h, (sineTable >> 8) & $FF    ; Get MSB of table
 ld a, (frame_count)             ; Get index
 ld l, a
 ld a, (hl)                      ; Look up value
; 7 bytes, 31 clocks
```

Instead of:

```
; Without 256-byte table alignment, simpler version
 ld hl, sineTable                ; Get address of table
 ld d, 0                         ; Set index high byte to zero
 ld a, (frame_count)
 ld e, a                         ; Set index low byte
 add hl, de                      ; Add offset to base
 ld a, (hl)                      ; Look up value
; 11 bytes, 52 clocks

; Without 256-byte table alignment, optimized version
 ld a, (frame_count)             ; Get index
 add a, sineTable%256
 ld l,a
 adc a, sineTable/256
 sub l
 ld h,a                          ; Add address of table to index
 ld a, (hl)                      ; Look up value
; 11 bytes, 46 clocks
```

Incrementing within aligned tables

Use an aligned address on memory such as $8000 (theoretical example) and if you will only use 256 bytes ($8000 to $80FF), to get the next byte use inc l instead of inc hl (2 clocks faster).

Crazy, "magick", hacks and obscure optimization's tricks

These are not normally recommend for use because some disturb disassembly and even coders understanding the code.

Better else

So you normally have an if-else-endif block like this:

```
    jr nz,else    ; the IF condition
    ;some code
    jr endif
else:
    ;some code
endif:
```

But here's a crazy trick for when the ELSE code is a single 2-byte instruction: You use the first byte of a 3 byte instruction with no side effects instead of the "jr endif" line! So if you had code like this:

```
    cp 7
    jr nz,else
    ld a,3      ; the IF code
    jr endif
else:
    ld a,4      ; the ELSE code
endif:
; 10 bytes, 33 T-states (for IF) or 26 T-states (for ELSE)
```

You could replace it with this:

```
    cp 7
    jr nz,else
    ld a,3      ; the IF code
    .db $C2  ;jp nz,xxxx
else:
    ld a,4      ; the ELSE code
endif:
; 9 bytes, 31 T-states (for IF) or 26 T-states (for ELSE)
```

Instead of branching over the ld a,4 instruction, it now executes a jp nz,XXXX instruction where the XXXX is the two bytes of the next instruction. You already know what the flags will be here, so you can make the jump never taken. You can use this to skip the next two bytes of execution! Who needs to branch over it?

This only takes 31 T-states for if. A small saving of 2 T-states, but could be useful in tight loops, and saves 1 byte! The only reason not to use this for 1-byte or 2-bytes instructions would be code readability and bug safety. Watch those flags!

However, when the ELSE code is a single 2-byte instruction as above, it's usually better to simply execute the ELSE part in all cases, then just skip the IF part depending on a certain condition. Although this option won't be always possible, obviously:

```
    cp 7
    ld a,4      ; the ELSE code
    jr nz,endif
    ld a,3      ; the IF code
endif:
; 8 bytes, 28 T-states (for IF) or 26 T-states (for ELSE)
```

In this particular example, the code could be optimized even further:

```
    cp 7
    ld a,4      ; the ELSE code
    jr nz,endif
    dec a       ; the IF code
endif:
; 7 bytes, 25 T-states (for IF) or 26 T-states (for ELSE)
```

Conditional rst

For a smaller conditional rst $38, use jr cc, -1. This will cause a conditional jump to the displacement byte ($FF) which is the rst $38 opcode.

DAA trick

Normally DAA instruction is used for BCD math but can be used for converting (?) ASCII integer.

```
cp 10
ccf
adc a, 30h
daa
```

Related topics

- MaxCodez TI-ASM optimization (http://www.junemann.nl/maxcoderz/viewtopic.php?f=5&t=675)
- ticalc archives: 1 (http://www.ticalc.org/archives/files/fileinfo/108/10821.html) 2 (http://www.ticalc.org/archives/files/fileinfo/285/28502.html)
- Balley Alley Z80 Machine Language Documentation (http://www.ballyalley.com/ml/z80_docs/z80_docs.html)
- Fast loops in MSX Assembly Page (http://map.grauw.nl/articles/fast_loops.php)
- Shiar z80 optimization page (http://shiar.nl/calc/z80/optimize)
- SMS Power! dev wiki z80 Techniques (http://www.smspower.org/Development/Z80ProgrammingTechniques)

Acknowledgements

- fullmetalcoder
- Galandros
- Dwedit for sharing in MaxCoderz the "Better else" trick with JP NZ
- MaxCoderz participants in assembly optimizing topic (Jim e,CoBB,...)
- SMS Power wiki
- lunarul
- Einar Saukas
- Alvin (Alcoholics Anonymous)
- Metalbrain