```
entry:
```

Advanced Z-80 Programming

George Phillips
george@48k.ca

48k.ca/advz80.pdf

https://discord.gg/fDyTDAU

Free free to e-mail me if you want to talk about Z-80 or TRS-80s. Better still, go to that Discord and we can chat interactively.
The code examples in this talk come from many different places. The original TRS-80 BASIC ROMs, other machines such as the ZX Spectrum and MSX. Not to mention original TRS-80 programs.
Some of the tricks use ordinary or boolean math or the surprising connection between the two. I highly recommend the book "Hacker's Delight" by Henry S. Warren Jr. for a full treatment of the subject.

General Advice

For fast, small code use instructions in the 4 to 11 T-State range.

LDIR, IX, IY make programming much easier but are big and slow.

Keep as much as you can in registers. EXX, EX AF,AF' and IXL/IXH help.

8 bits often faster than 16.

PUSH/POP fastest way to memory by far.

> LDIR by itself is not big (but at 21 T-States is slow). It's more than if you use it from a standing start where you end up doing three 16 bit loads. That's 9 bytes; 11 including the LDIR. A lot can happen in 11 bytes.

```
T-States
 |
 4 8000 37              SCF
 4 8001 3F              CCF
        |
        Bytes


 4 8002 B7              OR      A

 4 8003 A7              AND     A

 4 8004 BF              CP      A
```

Clear Carry

There are a lot of code fragments. The first column is the number of T-States or cycles to execute the instruction. The second is the address and the third is the assembled bytes. So the first column gives you the time and the third column the size.
The Z-80 has instructions to set the carry and complement it but none to clear it. Most code I've seen uses "OR A", but "AND A" is common in some circles. Nobody uses "CP A" but it'll do the job.

```
7 8000 FE00            CP     0
```

```
4 8001 B7              OR     A

4 8002 A7              AND    A
```

Test A == 0 or A < 0 or A > 0

Here's two different ways to test if A is smaller, bigger or equal to zero. Most often just as a test if A equal to 0 or not. Notice that it saves 3 T-States and 1 byte compared to the obvious "CP 0" technique.

```
7 8000 3E00            LD     A,0
```

```
4 8001 AF              XOR    A

4 8002 97              SUB    A
```

A = 0

Here's a very well known trick for setting A register to 0. I've mostly seen "XOR A" used but "SUB A" is practically identical. Both of them do affect the flags which can be a problem depending on the situation. Otherwise, grab the byte and 3 free T-States.

```
4 8000 9F              SBC    A,A

4 8001 79              LD     A,C
4 8002 87              ADD    A,A
4 8003 9F              SBC    A,A
4 8004 47              LD     B,A
```

A = Carry

"SBC A,A" seems like "SUB A" except that with the carry involved it actually ends up being something like "LD A,carry". If the carry is not set then A = 0, otherwise A = $FF.

It can have its uses in time sensitive code as a way to use the carry flag without a branch. I've shown an example to take C as an 8 bit signed value and extend it to a 16 bit signed value which just amounts to replicating the sign bit of C into all bits of B.

```
8 8000 CB27            SLA    A
```

```
4 8002 87              ADD    A,A
```

A = A * 2

It's smaller and cheaper just to add A to itself rather than use the shift-left-arithmetic "CB" opcode. They are slightly different. "SLA A" sets the parity flag whereas "ADD A,A" sets overflow.

```
8 8000 CB17             RL      A
```

```
4 8002 8F               ADC     A,A
4 8003 17               RLA
```

Rotate Left A

A few shift and rotate instructions have shorter equivalents due to 8080 compatibility. Here's one, "RL A" can be done as "RLA". However, RLA only sets the Carry flag. The crazy thing is that the 8080 didn't even need RLA since ADC A,A was just sitting right there!

If nothing else I've found that I can remember the difference between RL and RLC by remembering that RL and ADC are the same.

```
8 8000 CB07           RLC    A
8 8002 CB1F           RR     A
8 8004 CB0F           RRC    A
```

```
4 8006 07             RLCA
4 8007 1F             RRA
4 8008 0F             RRCA
```

Other A Rotates

And there are 3 other rotate instruction with faster variants for just A register. Again, watch out if want to test if A is zero since the shorter ones don't do that. And adding an "OR A" just means the result is the same speed as the "CB" version.

```
 8 8000 CB25            SLA    L
 8 8002 CB14            RL     H
16
```

```
11 8003 29              ADD    HL,HL
```

HL = HL * 2

Generally you'll have to do 16 bit (and larger) shifts 8 bits at a time. But "ADD HL,HL" is much smaller and reasonably faster for a 16 bit shift. Unsurprisingly, it is rather like the hypothetical instruction "SLA HL". On a similar note, "ADC HL,HL" can be thought of as "RL HL" if you have a use for that.

You might wonder about "SBC HL,HL". 15 T-States but you can do it in 12 with "SBC A,A; LD H,A; LD L,A". But if you really need a byte or A is in use then it'd be fine.

```
10 8000 11E803         LD     DE,1000
 4 8003 B7             OR     A
15 8004 ED52           SBC    HL,DE
29

10 8006 1118FC         LD     DE,-1000
11 8009 19             ADD    HL,DE
21
```

Add Negative To Subtract

Don't underestimate the power of simple arithmetic to save T-States and bytes. We get the same result in HL adding -1000 as we do subtracting but with 8 fewer T-States and 2 fewer bytes.

The flags are different, though so don't do this if you want to see if HL is equal to 1000. However, ADD HL,DE does set carry so you can use the technique to check if HL is bigger or smaller than some number. Just be careful as the sense of the carry will be reversed.

Because the Z-80 added SBC to the 8080 and not just SUB the main point is that 16 bit subtract is more expensive than 16 bit ADD. But 16 bit add with carry is the same as 16 bit subtract with carry.

```
; OK if 5 < A < 9
 7 8000 FE05          CP     5
12 8002 3815          JR     C,bad1
 7 8004 FE09          CP     9
12 8006 3011          JR     NC,bad1
-  8008       ok1:
-  ...
-  8019       bad1:

 7 8019 D605          SUB    5
 7 801B FE04          CP     9-5
12 801D 3011          JR     NC,bad2
-  801F       ok2:
-  ...
-  8030       bad2:
```

Range Check

So save a bit of time and space in doing ranged comparisons like the one shown, take advantage of modular arithmetic. And note my usage of "9-5". Many programmers will put in "4" but I like to "show my work" and let the assembler take care of it.

```
; HL = HL + A
 4 8000 4F              LD     C,A
 7 8001 0600            LD     B,0
11 8003 09              ADD    HL,BC
22
 4 8004 85              ADD    A,L
 4 8005 6F              LD     L,A
 4 8006 8C              ADC    A,H
 4 8007 95              SUB    A,L
 4 8008 67              LD     H,A
20
 4 8009 85              ADD    A,L
 4 800A 6F              LD     L,A
 7 800B 3001            JR     NC,noinc
 4 800D 24              INC    H
-  800E         noinc:
19 (or 20 if JR taken)
```

HL = HL + A

The Z-80 does have 16 bit operations but the instruction set is 8 bit at its heart (and 4 bit on the chip). Doing a hybrid operations like this (adding an 8 bit to a 16 bit value) can be done faster with just 8 bit math. Though it won't have the same flag results at all due to that trick of adjusting H to save a critical 3 T-States.

We can even do a little better by using a conditional branch. This averages to 19.5 T-States saving 0.5 from before. Note that if A is large a carry will happen frequently so using JP is better than JR as the usual case will be 4+4+10 or 18 cycles. But it'll be worse at 24 cycles if A is small and the JP isn't taken.

```
; HL'HL = HL'HL + $00800800
10 8000 110008          LD     DE,$0800
11 8003 19              ADD    HL,DE
 4 8004 D9              EXX
10 8005 118000          LD     DE,$0080
15 8008 ED5A            ADC    HL,DE
 4 800A D9              EXX
```

32 Bit Add

The "prime" registers available through EXX are really handy though they can be inconvenient as there isn't a fast facility for passing values between the two register banks. They're great for 32 bit arithmetic as seen in this example.

```
; A = (B & $F8) | (C & 7)
 4 8000 78                LD     A,B
 7 8001 E6F8              AND    $F8
 4 8003 47                LD     B,A
 4 8004 79                LD     A,C
 7 8005 E607              AND    $7
 4 8007 B0                OR     B
30

; A = ((B ^ C) & $F8) ^ C
 4 8008 78                LD     A,B
 4 8009 A9                XOR    C
 7 800A E6F8              AND    $F8
 4 800C A9                XOR    C
19
```

High 5 Bits of B Combined with Low 3 of C

A mastery of boolean algebra can speed up code. Consider this operation where we want to take the high 5 bits of B and combine them with the low 3 bits of C. The simpleminded solution masks both source values but we can use the properties of XOR to do the operation much more quickly.

And, as a bonus, the mask value could easily be a parameter in a register. The simple code would get much more complicated having to compute the mask and its inverse.

```
; A = A | B
 4 8000 B0                OR      B

; A = A & ~B
 4 8001 4F                LD      C,A
 4 8002 78                LD      A,B
 4 8003 2F                CPL
 4 8004 A1                AND     C

; A = (A | B) ^ B
 4 8005 B0                OR      B
 4 8006 A8                XOR     B
```

Set Bits and Clear Bits

Similar to the previous transformation, consider wanting to set all the bits in A where the corresponding bit is 1 in B register. That's trivial: just OR. But if you want to clear the corresponding bit in A then you might think you need to complement the mask value. Or just use more XOR magic.

```
 7 8000 2605           LD     H,5
 4 8002 2D             DEC    L
 4 8003 85             ADD    A,L

11 8004 DD2605         LD     IXH,5
 8 8007 DD2D           DEC    IXL
 8 8009 FD85           ADD    A,IYL
```

Split IX, IY are OK

A quick word on undocumented instructions. The Z-80 has a number of them including access to the high and low bytes of IX and IY. I consider these well-supported and have no qualms using them. They're all 1 byte and 4 T-States slower then their "HL" counterparts (DD means IX, FD means IY) but they can be very handy as extra registers when you run low.

```
 4 8000 EB               EX     DE,HL
11 8001 09               ADD    HL,BC
 7 8002 86               ADD    A,(HL)
 4 8003 EB               EX     DE,HL
```

EX DE,HL most Underrated

I think "EX DE,HL" is the most underrated Z-80 instruction (and its actually an 8080 instruction but let's not quibble). Since you can do so much with HL it is like having a second HL register (or, uh, third and fourth if you consider EXX).

This example shows storing A register into (DE+BC). It's not super compelling but it's hard to come up with a concise example. Probably why the instruction is so often overlooked. But I find it can help get out of tight spots where registers are running out and can avoid having to use IX or IY.

```
13 8000 3A00A0          LD     A,(var1)
 4 8003 3C              INC    A
13 8004 3200A0          LD     (var1),A
30

 7 8007 3E00            LD     A,0
-  8008         var2    equ    $-1
 4 8009 3C              INC    A
13 800A 320880          LD     (var2),A
24

10 800D 2100A0          LD     HL,var1
11 8010 34              INC    (HL)
21
```

Variable self-modify

The Z-80 is fairly slow to access memory. If you're not writing code for a ROM you can save a good number of cycles by storing the variable value in one of the LD instructions. That's obviously even better if that LD is in a loop. And that kind of thing is one of the chief reasons why self-modifying code can speed things up. 6 cycles vs. 13 if you modify a load rather than using an indirect load.

The Z-80's 8080 heritage shows through in that it really wants you to load HL with the address of a variable. Look at how the last code fragment is faster still despite the self-modifying cleverness.

```
; A = A / 3
 7 8000 2681            LD     H,high(div3)
 4 8002 6F              LD     L,A
 7 8003 7E              LD     A,(HL)
18

-   8100                org    ($+255) & -256
 4 8100 000000 div3:    DEFB   0/3,1/3,2/3
10 8103 010101          DEFB   3/3,4/3,5/3
 7 8106 020202          DEFB   6/3,7/3,8/3
 6 8109 030303          DEFB   9/3,10/3,11/3
-   ...
```

8 Bit Page Aligned Table (div 3)

For more complicated functions, nothing beats page (256 byte) aligned table lookups. A mere 18 T-States to to look up a value or only 11 if H already points to the table. Here we use a lookup table to divide A register by 3.

Fastest way to map an 8 bit input to an 8 bit value. Only one or two instruction functions will be faster.

Notice the "org" statement uses a cute expression to ensure the table is page aligned.

```
 7 8000 2681             LD     H,high(squared)
 7 8002 56               LD     D,(HL)
 4 8003 24               INC    H
 7 8004 5E               LD     E,(HL); DE=L*L

-  8100                  org    ($+255) & -256
-  8100         squared:
-  0000                  x = 0
-  8100 00000000         rept 256
-  8104 00000000           DEFB x * x / 256
-                          x++
-                        endm
-  0000                  x = 0
-  8200  00010409        rept 256
         10192431          DEFB x * x % 256
-                          x++
-                        endm
```

16 Bit Page Aligned Table (squares)

For 16 bit result tables you may be tempted to store them as 16 bit values. But in another way the Z-80 is 8 bit at heart you should split the values across into a page of high bytes and a page of low bytes. Then you can handle 256 inputs without 16 bit math (or any math at all). By having the pages adjacent a simple "INC" gets us there faster and shorter than another "LD H,high(squared+1)".

The "x = 0" and "x++" notation are something I added specifically to zmac for making table generation such as this more concise.

Going Places

```
 4 8000 37              SCF
 7 8001 30FE            JR     NC,$
12 8003 38FE            JR     C,$
12 8005 18FE            JR     $
10 8007 C30580          JP     $
```

```
Use JR for forward conditional branches.
Use JP on all others.

Specifically, conditional JR beats JP if
branch is taken less than 60% of the time.
```

JP vs JR

Obviously using a JR is more compact than a JP. The forward/back advice is based on the idea that backward jumps are typically for loops and forward jumps for exceptional cases. Obviously there are many counter-examples to this heuristic.

In fact, looping or not, for speed use JR is a branch is taken less than 60% of the time. A JR is faster if not taken so you should "JR" to the uncommon rather than common case.

```
12 8000 2803            JR    Z,incd1
 4 8002 15              DEC   D
12 8003 1801            JR    done1
 4 8005 14     incd1:   INC   D
-  8006        done1:

12 8000 2802            JR    Z,incd2
 4 8002 15              DEC   D
 7 8003 3E14            LD    A,$14
-  8004                 org   $-1
 4 8004 14     incd2:   INC   D
```

One byte jump

Here's a trick that the BASIC ROM uses to save code bytes. Suppose we want to add 1 to D if zero is set and subtract 1 otherwise. Normally that means having two jumps but if we don't care about the contents of A register we can effectively jump over the "INC D" by hiding it in a "LD A,n" instruction. It's pretty weird. And to be clear the second code segment is only 5 bytes long (look at the address repeating on the left).

The "LD A,n" is kind of a 1 byte jump forward.

The takeway here is to remember how the processor works. You can go ahead and jump into the so-called middle of an instruction and it'll just to what you tell it.

```
10 8000 11DD23          LD     DE,$23DD
-  8002                 org    $-1
10 8002 DD23   lp:      INC    IX
13 8003 10FC            DJNZ   lp
```

Two byte jump

LD DE,nn and other 16 bit loads can act as a 2 byte jump forward. Again, with a side effect of "trashing" a register pair. Here in this artificial example we want to add B-1 to IX. The first time through we do nothing. The load of DE skips the INC IX. Subsequent iterations do the INC IX so it happens B - 1 times.

```
 4 8000 E9              JP     (HL)
 5 8001 D8              RET    C        ;skip
 7 8002 38FE            JR     C,$      ;skip
 8 8004 FDE9            JP     (IY)
 8 8006 10FE            DJNZ   $        ;skip
10 8008 C30880          JP     $
10 800B C9              RET
10 800C DC1B80          CALL   C,PRINT ;skip
11 800F D0              RET    NC
11 8010 CF              RST    8
12 8011 18FE            JR     $
13 8013 10FE            DJNZ   $
17 8015 CD1B80          CALL   PRINT
```

Relative Speed of flow change

Here's a quick cheat sheet demonstrating the relative speed of change of flow instructions. Nobody beats JP (HL), conditional return not taken is a bit faster than conditional JR not taken. So while a CALL is expensive it can win back that time if there are many conditional returns not taken in it compared to JR's not taken. JP (IY) or (IX) is just a bit faster than JP so use that to grab a couple extra T-States in a loop.

```
; case of A 0,1,2,3
 4 8000 87              ADD    A,A
13 8001 320580          LD     (jroff),A
12 8004 18FE            JR     $
-  8005         jroff   equ    $-1
12 8006 1826            JR     case0
41
12 8008 1835            JR     case1
12 800A 1844            JR     case2
-               case3:  ...
-                       ...
-               case0:  ...
-               case1:  ...
-               case2:  ...
```

Simple Jump Table

Switch statements or "jump tables" where a different bit of code is executed depending on a register value are common. One easy and compact way to aproach this is to self-modify the offset of a JR instruction.

Note how we avoid a "JR case3" by staring that case at offset 6.

If you can control the values of a case (say by them being 0, 2, 4, 6) then the "ADD A,A" isn't required.

```
 4 8000 6F              LD     L,A
 7 8001 2681            LD     H,high(jump)
 7 8003 6E              LD     L,(HL)
 4 8004 E9              JP     (HL)
22
-  8100                 org    ($+255) & -256
 6 8100 03      jump:   defb   low(do_1)
 4 8101 14              defb   low(do_2)
 4 8102 25              defb   low(do_3)
-  8103         do_1:   ...
-  8114         do_2:   ...
-  8125         do_3:   ...
```

Fast Jump Table

For pure speed you can't beat JP (HL) at a mere 4 T-States. As with ordinary table lookup the best approach is to split 16 bit values into two pages. But if the possibilities are small then a single byte can be used as the destination as long as the jump table and the destinations are all on a single page.

Here I've given them the most possible space with full page alignment but that is more than is required.

We could lose a bit of time and have the table not start on a page boundary. And we could even do a full 16 bit add but it'd be slower.

Subroutines

Great way to simplify programming.

Arguments and return values in registers.
Otherwise fixed memory locations (not stack).

Zero flag is best for boolean (yes/no) return.

Carry flag is best for single bit return value.

Always have one subroutine named "ME".

Not sure if that's a Blondie or Carly Rae Jepsen joke.

Blondie: CALL ME
Carly: CALL Z,ME ; Call ME, maybe.

```
11 8000 29       mul8:   ADD    HL,HL
11 8001 29               ADD    HL,HL
11 8002 29               ADD    HL,HL
10 8003 C9               RET

17 8004 CD0080           CALL   mul8
```

HL = HL * 8

Small subroutines are not good for speed and size reasons. This routine to multiply HL by 8 could be done in-line faster and would reduce the size of the entire program by 4 bytes because the 3 "ADD HL,HL" instructions would fit in the 3 bytes the CALL requires.

```
 4 8000 7C      hexHL:  LD     A,H
17 8001 CD0E80          CALL   hexA
 4 8004 7D              LD     A,L  ;CALL hexA;RET
11 8005 F5      hexA:   PUSH   AF
 4 8006 0F              RRCA
 4 8007 0F              RRCA
 4 8008 0F              RRCA
 4 8009 0F              RRCA
17 800A CD0E80          CALL   hex
10 800D F1              POP    AF   ;CALL hex;RET
 7 800E E60F    hex:    AND    15
 7 8010 FE0A            CP     10
 7 8012 DE69            SBC    $69
 4 8014 27              DAA         ;CALL putch;RET
10 8015 C300B0          JP     putch
```

Fall Though to Tail and DAA Magic

A very common space and time-saving technique it to have subroutines "fall though" rather than having a "CALL to-subroutine; RET" sequence at their end. Here the routine to convert HL to hexadecimal does a call to output H and then falls-through to print L. Similarly, hexA falls through to do the lower nybble of A.

Notice how "hex" converts binary 0 .. 15 to ASCII '0' .. '9', 'A' ... 'F'. I won't go into it (nor do I fully understand it), but considering DAA's job is to take results like 5 + 6 = 11 and convert them to BCD "11" you can kind of see the connection with hexadecimal.

```
-  0000        no_err  equu    0
-  0002        sn_err  equ    2
-  0004        tm_err  equ    4

 7 8000 3E04           LD     A,tm_err
12 8002 1811           JR     done
                       ...
 7 8010 3E02           LD     A,sn_err
 4 8012 B7     done:   OR     A
10 8013 C9             RET
```

Set Z flag for Non-zero Error Codes

The Z flag is pretty convenient if A register contains an error code. If any non-zero code is an error then "OR A" suffices to set the flag. This means the caller does not have to do the test themselves but can immediate branch to test for failure. Or conditionally return on not-zero to propagate the error back to their caller.

```
12 8000 1825            JR     s_ok
-   ...
12 8013 1811            JR     s_fail
-   ...
 7 8026 F6AF   s_fail: OR     $AF
-   8027                org    $-1
 4 8027 AF     s_ok:   XOR    A
10 8028 C9              RET
```

Boolean Tail Overlap

The overlapping instruction technique can be used to create compact good/bad exit points in a subroutine. Since $AF is non-zero the "OR $AF" will guarantee Z is not set. But the JR to "s_ok" will be "XOR A" which ensures Z is set.

```
17 8000 CD1B80          CALL   print
 4 8003 48656C6C        ascii  'Hello!',0
        6F2100
10 800A 010A00          LD     BC,10
-  ...
19 801B E3      print:  EX     (SP),HL
 7 801C 7E      plp:    LD     A,(HL)
 6 801D 23              INC    HL
 4 801E B7              OR     A
12 801F 2805            JR     Z,pdn
17 8021 CD00B0          CALL   putchar
12 8024 18F6            JR     plp
19 8025 E3      pdn:    EX     (SP),HL
10 8026 C9              RET
```

Arguments After CALL

One very convenient trick is to put CALL parameters directly after the CALL instruction. It saves having to load (and use) a register and annoys reverse engineers as a side effect. It also shatters the myth that a Z-80 will return to the next instruction after a CALL.

Here we swap the return address and HL with the EX (SP),HL. Then read over the nul terminated string as we print it and simply return back to the next instruction after the string.

```
; Model I/III     BASIC
11 8000 C7   RST $0  ; reboot
11 8001 CF   RST $8  ; JP $4000 syntax check
11 8002 D7   RST $10 ; JP $4003 next char
11 8003 DF   RST $18 ; JP $4006 CP HL,DE
11 8004 E7   RST $20 ; JP $4009 get type
11 8005 EF   RST $28 ; JP $400C break/sys
11 8006 F7   RST $30 ; JP $400F unused/debug
11 8007 FF   RST $38 ; JP $4012 IM 1 intr
```

RST is compact but slow (Model I/III)

The Z-80 features a fast call instruction called RST. It is 6 T-States faster and 2 bytes smaller than a CALL. But it can only go to 8 fixed locations which then generally involve an additional JP either because the locations are vectored from ROM or 8 bytes is not sufficient for a subroutine. As a result a RST isn't usually faster than a CALL but it certainly is very compact.

On the Model I/III only one RST vector is completely unused. RST 0 will just warm start the machine and RST $38 is the interrupt.

```
; Model 4
11 8000 C7   RST $0  ; reboot
11 8001 CF   RST $8  ; unused
11 8002 D7   RST $10 ; unused
11 8003 DF   RST $18 ; unused
11 8004 E7   RST $20 ; unused
11 8005 EF   RST $28 ; syscall
11 8006 F7   RST $30 ; debug
11 8007 FF   RST $38 ; IM 1 intr
```

RST is compact and maybe fast (Model 4)

The Model 4 has 4 unused RST vectors, one shared by the interrupts and the others availble if you don't need TRS-DOS.

```
; Model II
11 8000 C7   RST $0  ; possible (TRSDOS return)
11 8001 CF   RST $8  ; syscall
11 8002 D7   RST $10 ; debug
11 8003 DF   RST $18 ; unused
11 8004 E7   RST $20 ; unused
11 8005 EF   RST $28 ; unused
11 8006 F7   RST $30 ; unused
11 8007 FF   RST $38 ; unused
```

RST is compact and maybe fast (Model II)

The Model II uses mode 2 interrupts (IM 2) so RST $38 is available as are others. At least when using TRS-DOS. If it is running LS-DOS then it is the same as the Model 4 except that RST $38 is free.

Looping

```
13 8000 10FE            DJNZ   $

 4 8002 05              DEC    B
12 8003 20FD            JR     NZ,$-1
16

 4 8002 15              DEC    D
10 8003 C20280          JP     NZ,$-1
14
```

DJNZ is small but slightly fast

For 8 bit loops you can't beat DJNZ for size and speed. But its speed advantage is surprisingly slim. Just 1 cycle savings over a decrement and JP and you can use other conditions besides != 0.

Use it when you can but if it doesn't fit easily into the code then it may not be saving any time.

```
-               loop1:  ...
 6 8011 0B              DEC    BC
 4 8012 78              LD     A,B
 4 8013 B1              OR     C
10 8014 C20080          JP     NZ,loop1
24

-               loop2:  ...
16 8028 EDA1            CPI
10 802A EA1780          JP     V,loop2
26

 6 802D 23              INC    HL
```

16 bit BC loop using CPI

For loops with 16 bit counters the easy approach is to use a register pair. To make register pairs more convenient as pointers the 16 bit INC/DEC instruction does not set flags. So extra work must be done to test for 0.

There is an instruction, CPI, which will decrement BC and set the overflow flag. It is a byte shorter than the obvious code but slightly slower. It also has the side effect of reading (HL) and incrementing HL so be careful.

However, if your code happens to need HL incremented anyways then it is 4 T-States faster as a bonus.

Because CPI sets the usual flags for other reasons the parity/overflow flag is used to indicate BC != 0. My mnemonic is "overflow means we keep flowing through the loop".

And don't forget CPD if you want HL to decrement.

```
; BC to B inner, C outer
 4 8000 AF              XOR    A
 4 8001 B9              CP     C
 4 8002 88              ADC    A,B
 4 8003 41              LD     B,C
 4 8004 4F              LD     C,A
-               loop3:  ...
13 8016 10ED            DJNZ   loop3
13
 4 8018 0D              DEC    C
10 8019 C20580          JP     NZ,loop3
```

Change 16 bit BC loop to two 8 bit Nested

The best way to go faster is to break the loop into an inner loop and outer loop with 8 bit counters. If the loop count is known in advance you can do this by hand. But if the loop count is a parameter then here's a little bit of code to transform the 16 bit loop count into the necessary two 8 bit loop counts. This works for all values of BC including 0 == 65536.

Each iteration only takes 13 rather than the 24 of the straightforward approach. The outer loop iterations are relatively infrequent so don't add much to the total.

The first run through the loop will be the remainder. After that C counts off how many times we do 256 iterations of the loop taking advantage of the fact that B == 0 after the DJNZ completes.

```
; DE to E inner, D outer
 6 8000 1B              DEC    DE
 4 8001 14              INC    D
 4 8002 1C              INC    E
-               loop4:  ...
 4 8014 1D              DEC    E
10 8015 C20380          JP     NZ,loop4
14
 4 8018 15              DEC    D
10 8019 C20380          JP     NZ,loop4
```

Change any pair to nested

If you don't need that extra T-State per iteration DJNZ gives you then there's an even more compact transformation. It's a good one to sit down and think through how it works.

```
10 8000 21003C          LD     HL,$3C00
 4 8003 AF              XOR    A
 7 8004 77     loop5:   LD     (HL),A
 6 8005 23              INC    HL
 8 8006 CB74            BIT    6,H
12 8008 28FA            JR     Z,loop5
```

Drop Counter and Test Pointer

Another way to speed up loops is to use the pointer to memory as a counter. Thus not having to maintain both a pointer and a counter. This can be difficult and slow in general on a Z-80 as "SBC" is the only 16 bit comparison we have and it is destrutive (though EX DE,HL can help). This example shows clearing the screen on a Model III ($3C00 .. $3FFF). It's not actually faster than when broken into two 8 bit loops but it is quite compact (and never mind how much slower it is than PUSH).

```
10 8000 21003C          LD     HL,$3C00
 4 8003 7D              LD     A,L
 7 8004 77      loop6:  LD     (HL),A
 4 8005 2C              INC    L
10 8006 C20480          JP     NZ,loop6
 4 8009 24              INC    H
 8 800A CB74            BIT    6,H
12 800C 28F6            JR     Z,loop6
```

Nested Counter Drop

This clear screen is faster yet as it not only uses an 8 bit inner loop but gets rid of the inner counter by using L for both purposes. We're running through memory without using a 16 bit increment!

This is like the previous loop but basically the "INC HL; BIT 6,H" is replaced with an "INC L" dropping 10 T-States out of the loop.

Looping over memory a page at a time like this is a very fast approach. Combined with the split values we say with the table lookups you can see how this can lead to some compact and fast code.

```
; Almost the same as LDIR if BC is even
; (loses 5 cycles in total)
16 8000 EDA0   loop7:  LDI
16 8002 EDA0           LDI
10 8004 EA0080         JP     V,loop7
42 (21 / byte)

; faster than LDIR if BC is a multiple of 4
16 8007 EDA0   loop8:  LDI
16 8009 EDA0           LDI
16 800B EDA0           LDI
16 800D EDA0           LDI
10 800F EA0780         JP     V,loop8
74 (18.5 / byte)
```

Unrolling LDIR

A general technique that can be used anywhere is loop unrolling and the Z-80 is no exception.

LDIR is very handy but if BC is even we can unroll LDIR into two LDI instructions and be almost as fast. If we unroll 4 times (BC must be divisible by 4) then the average per byte drops from 21 - epsilon to 18.5

If the amount to copy is known then you can unroll fully for 16 T-States/byte.

```
 7 8000 0610           LD      B,64/4
 4 8002 AF             XOR     A
 7 8003 86     loop9:  ADD     A,(HL)
 4 8004 2C             INC     L
 7 8005 86             ADD     A,(HL)
 4 8006 2C             INC     L
 7 8007 86             ADD     A,(HL)
 4 8008 2C             INC     L
 7 8009 86             ADD     A,(HL)
 6 800A 23             INC     HL
13 800B 10F6           DJNZ    loop9
```

Unrolling and using 8 bit INC

Since 8 bit increment is two T-States less than 16 bit increment you can also use unrolling to save on the update of the memory pointer. As long as HL 4 byte aligned in this case. And if you know HL doesn't cross a page boundary you can use INC L for all the updates. If we're feeling really clever we could make the page boundary the end of loop condition and drop the counter.

```
20 8000 ED730C82        LD     (spsave),SP
 4 8004 F3              DI
10 8005 112020          LD     DE,' '*$101
10 8008 310040          LD     SP,screen+$400
   800B D5D5D5D5        rept 1024/2
11 800F D5D5D5D5          PUSH DE
                        endm
10 820B 310000          LD     SP,0
-  820C         spsave  equ    $-2
 4 820E FB              EI
```

```
 7 8000 77              LD     (HL),A
 4 8001 2C              INC    L
```

Unroll All the Way (fast clear)

When it comes to reading a writing memory there is nothing faster on the Z-80 than POP and PUSH. POP can read two bytes in 10 T-States (5 per byte) and PUSH can write two bytes in 11 T-States (5.5 per byte). And you even get to update the destination pointer in that.

Here's a super fast, fully unrolled screen clear. Interrupts must be disabled because we're using the stack. This example is for a TRS-80 Model I, III or 4 but you can do similar things on a Model II but DI won't stop NMI so you'll need to turn that interrupt source off for safety.

Just for comparison the fastest an unrolled conventional loop could manage is 11 cycles/byte.

Examples

Three Useful

One Disturbing

```
ldv     macro   reg,var
        LD      REG,(IX+var-varbase)
endm

stv     macro   reg,var
        LD      (IX+var-varbase),reg
endm

decv    macro   var
        DEC     (IX+var-varbase)
endm

addv    macro   var
        ADD     A,(IX+var-varbase)
endm
```

Zero Page IX Macros

Programmer convenience isn't nearly as compelling as small and fast programs but it is important. There's a lot of code to write where speed and size are minor concerns but it is complicated enough that juggling registers becomes difficult. These macros allow you to set up 256 byte region in memory of very convenient 8 bit variables. You can pull the variables into any register and even use them as loop counters.

```
10 8000 01     step     defb  1
-  8001        varbase:
 4 8001 00     count    defb  0
 4 8002 00     pos      defb  0

14 8003 DD210180        LD    IX,varbase
-                       ...
19 8018 DD7E00          ldv   A,count
 7 801B C605            ADD   A,5
19 801D DD86FF          addv  step
19 8020 DD7700          stv   A,count
-              lp:      ...
23 8031 DD3500          decv  count
12 8034 20EA            JR    NZ,lp

10 8036 210180          LD    HL,count
11 8039 35              DEC   (HL)
21
```

Zero Page IX Usage

Here's a quick example of the macros in use. One thing to note is that you can still access the variables directly by their name if that is easier. The cycle counts and byte sizes make it clear that the technique is not winning any speed or size contests. In fact, look how we can decrement a variable using HL as a pointer FASTER than using IX indexing using only one byte extra.

```
; Just load A with 0 to start up.
; Use SLA r and RL r for other reg.
 4 8000 87     getbit: ADD    A,A
11 8001 C0             RET    NZ
17 8002 CD0080         CALL   getbyt
 4 8005 37             SCF
 4 8006 8F             ADC    A,A
10 8007 C9             RET
```

Stream Bits

Oftentimes you need to read data a bit at a time for compression or anything where compact data representation is important. Here's a very efficient approach that doesn't require a register to keep track of the number of bits remaining in the current byte.

The trick is to ensure that the bit buffer (A register in this case) will always be non-zero when there are bits remaining. So in the normal case the next bit put into the carry flag and the routine terminates. If not, we get another byte from the input stream, put the top bit into carry and set the lowest bit. That's what guarantees the value will be non-zero as more bits remain. After 7 more calls that bit will end up in the carry but it was just a marker of sorts, A will be all 0's so we'll get another byte.

The whole thing starts out by loading A register with 0. And you can easily use another register besides A if slightly slower.

```
 4 8000 2C      getbyt: INC    L
 7 8001 7E              LD     A,(HL)
11 8002 C0              RET    NZ
17 8003 CD1980          CALL   fillbuffer
 7 8006 7E              LD     A,(HL)
10 8007 C9              RET
-                       ...
-  8019         fillbuffer:
-                       ...
10 8029 C9              ret
10 802A 21FF81 setup:   LD     HL,buffer+255
-                       ...
-  8100                 org    ($+255) & -256
-  8100         buffer: defs   256
```

Stream Bytes

"getbyt" may just step through memory. But if we'd like to stream from floppy then we can use the page-aligned tricks to make another efficient routine for reading another block of 256 bytes at a time. As long as L doesn't "wrap around" to the start of the page.

Initialization is as simple as loading HL with buffer+255 guaranteeing the first call will fill the buffer. If reading from a floppy "fillbuffer" will jump to an error handler if it fails (i.e., an exception). If there is not a full 256 bytes remaining in the file then it only need put them at the end of the buffer and point HL to them.

"INC L" sets the sign bit. By self-modifying the "RET" condition we could have fillbuffer be called after the first or second 128 bytes of the buffer have been consumed. That could give us some parallelism where we kick off a DMA for the consumed part of the buffer so it may be filled while we continue processing.

```
 7 8000 1E08            LD     E,od_err
10 8002 C32F80          JP     error
-                       ...
10 8016 C32D80          JP     e_syn
-                       ...
 7 802A 1E04   e_fc:    LD     E,4
10 802C 01              defb   1
 7 802D 1E02   e_syn:   LD     E,2
10 802F 213880 error:   LD     HL,errtab
 7 8032 1600            LD     D,0
11 8034 19              ADD    HL,DE
10 8035 C31B80          JP     print
 7 8038 4E46   errtab:  ascii  'NF'
 4 803A 534E            ascii  'SN'
 7 803C 4643            ascii  'FC'
```

How BASIC Signals Errors

Here's an example of space optimization that might well offend your sensibilities and is something I did whilst fooling around trying to make a smaller version of the Model III BASIC ROM.

BASIC handles errors by loading E register with the error number (always a multiple of two) and jumped to the "error" routine which prints the 2 letter error code and so on.

Each error check can take as much as 5 bytes (load and JP) though common ones are reduced by 3 by jumping to a common "load E" location which can then fall through directly or with "LD BC," jumps.

Short of using RST I couldn't see any way to better this. And even then a "LD E,n; RST" is still 3 bytes so not doing much better than the usual case.

```
10 8000 CD2F80         CALL   e_od
-                      ...
10 8016 CD2C80         CALL   e_syn
-                      ...
 7 802A 00     e_od:   NOP
10 802B 00     e_fc:   NOP
 7 802C 00     e_syn:  NOP
; Use return address to find what was
; called and calculate E based on that.
10 802F 213880 error:  LD     HL,errtab
 7 8032 1600           LD     D,0
11 8034 19             ADD    HL,DE
10 8035 C31B80         JP     print
 7 8038 4E46   errtab: ascii  'NF'
 4 803A 534E           ascii  'SN'
 7 803C 4643           ascii  'FC'
```

NOP Slide For Errors

So for every error we have an entry in this series of NOPs (which won't just 3 but 23 long). To signal an error just call the right routine. We can then pop the return address, look back two bytes to see what the actual entry point of the routine was and then use that to compute E.

I've not shown the actual calculation because I wasn't terribly satisified with the result. It saved some bytes but not a whole lot one the size of the computation code was factored in. After all, the old system handled most error codes with only 3 bytes.

And those NOPs seemed like such a waste. 23 bytes of zeros. Could they at least be doing some work? Well, not especially. You can't stick a subroutine in there because the code has to fall through.

But then I thought, hey, what if we put the error strings where the NOPs are. Could that work?

```
7 8000 4E            LD    C,(HL) ; 'N'
7 8001 46            LD    B,(HL) ; 'F'
4 8002 53            LD    D,E    ; 'S'
7 8003 4E            LD    C,(HL) ; 'N'
7 8004 46            LD    B,(HL) ; 'F'
4 8005 43            LD    B,E    ; 'C'
```

Executing ASCII!?

However, let's consider the two letter error messages themselves. It's a strange question, but what if we were to run them as code? It turns out that mostly it would mean random registers loaded into other registers. Sometimes it would mean reading (HL) though not writing since those are lower-case letters.

Executing them would be harmless since HL normally points to the BASIC code being interpreted. And H and L load instructions are out of range of the upper-case letters.

```
17 8000 CD1980           CALL   err_sn
17 8003 C41780           CALL   nz,err_nf
-   ...
 7 8017 4E46   err_nf:   ascii  'NF'
 4 8019 534E   err_sn:   ascii  'SN'
 7 801B 4643   err_fc:   ascii  'FC'
-   ...
10 802E E1     error:    POP    HL
 6 802F 2B               DEC    HL
 7 8030 56               LD     D,(HL)
 6 8031 2B               DEC    HL
 7 8032 5E               LD     E,(HL)
 4 8033 7B               LD     A,E
 7 8034 D617             SUB    low(err_nf)
 4 8036 EB               EX     DE,HL
 4 8037 5F               LD     E,A
10 8038 C31B80           JP     print
```

Tighter BASIC Errors

So this insane solution can work. Instead of loading E register anywhere we just CALL the error string. This harmlessly falls through to "error" routine (which is no longer called directly). The CALL isn't because we can return from an error (we can't). Instead, the "error" routine looks at the return address which is just after the CALL to the error string. By backing up two bytes we get the low address of the error string which we can adjust into the error code E. And we were given the address of the string already so that's that.

My actual code was a little more complicated and safer. It had to allow for "/0" and "L3" errors, too, which are less convenient opcodes. There was also more bookkeeping BASIC which I've left out.

Saved 35 bytes!

```
        end     entry
```

End of Class

A small joke here. We have the entry point on the first slide so we must make sure to tell the assembler that we have finished and where to start.