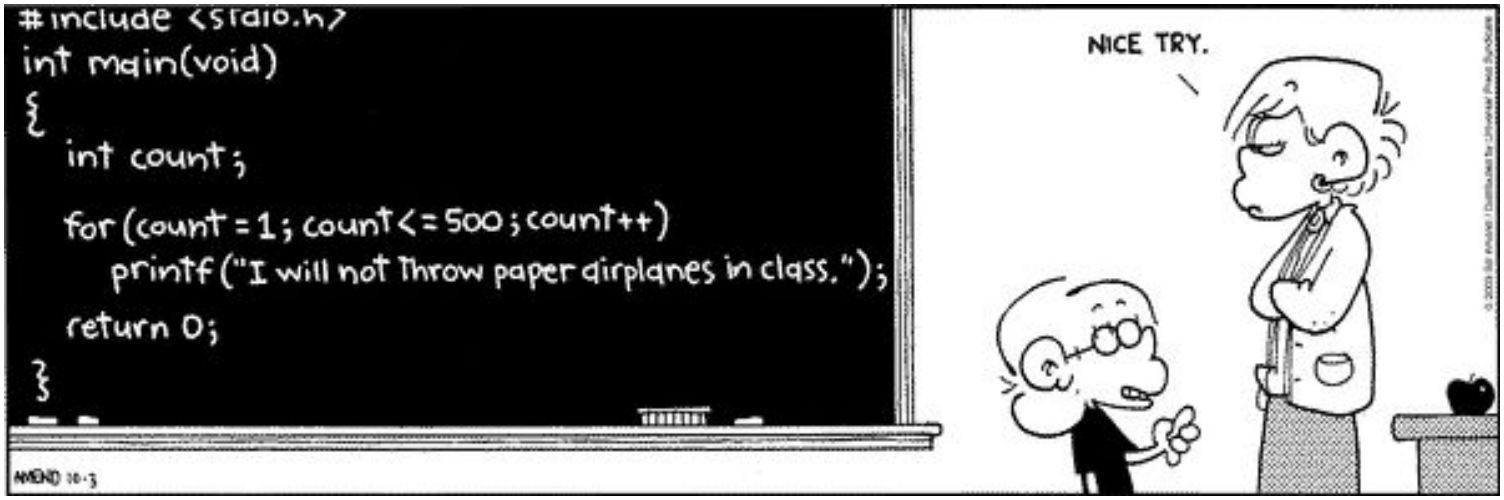
What would a compiler do?



Today we'll look at the assembly code that compiler's generate...

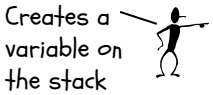
Code generation

A simple example written in C:

```
int array[10];
int total;

int main( ) {
    int i;

    total = 0;
    for (i = 0; i < 10; i++) {
        array[i] = i;
        total = total + i;
    }
}
```

Code we might write

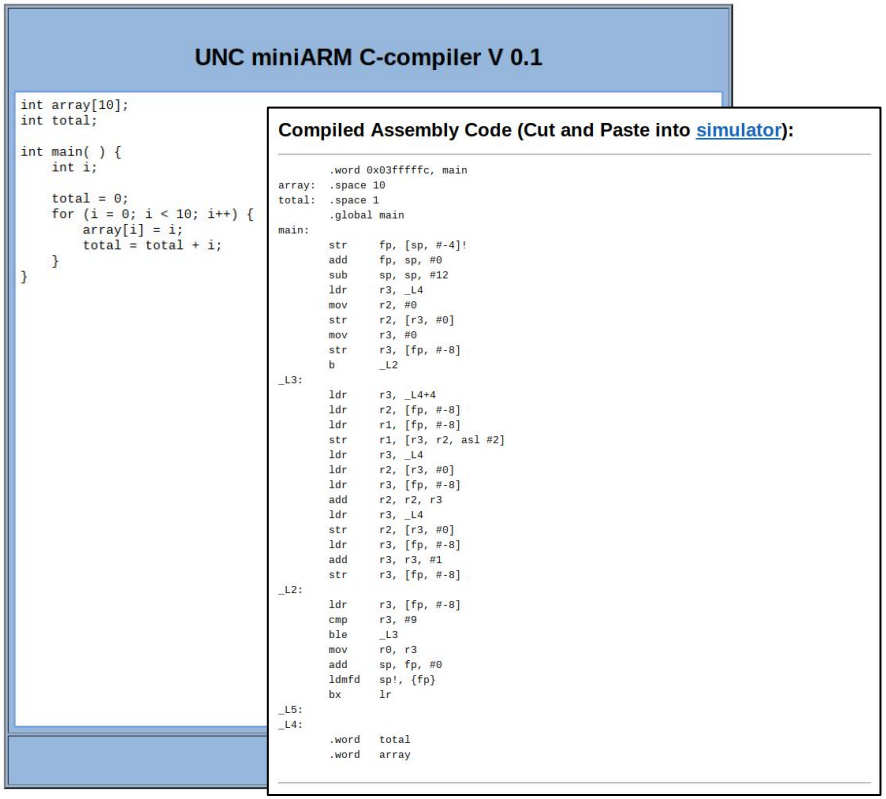
```
        .word 0x03fffffc, main

array:  .space  10                  ; int array[10];
total:  .space  1                   ; int total;

main:                               ; int main() {
        sub     sp,sp,#4            ;     int i;
        mov     r0,#0
        str     r0,total            ;     total = 0;
        str     r0,[sp]             ;     for (i = 0; i < 10; i++) {
        b       test
block:  mov     r1,#array
        str     r0,[r1,r0,lsl #2]   ;         array[i] = i;
        ldr     r1,total
        add     r1,r1,r0
        str     r1,total            ;         total = total + i;
        add     r0,r0,#1
        str     r0,[sp]
test:   cmp     r0,#10
        blt     block               ;     }
        add     sp,sp,#4
*       bx      lr
```



Let's try the Compiler

The paste our C program into the miniARM C compiler at:

http://csbio.unc.edu/mcmillan/index.py?run=arm



UNC miniARM C-compiler V 0.1

```
int array[10];
int total;

int main( ) {
    int i;

    total = 0;
    for (i = 0; i < 10; i++) {
        array[i] = i;
        total = total + i;
    }
}
```

Compiled Assembly Code (Cut and Paste into simulator):

```
        .word 0x03fffffc, main
array:  .space 10
total:  .space 1
        .global main
main:
        str     fp, [sp, #-4]!
        add     fp, sp, #0
        sub     sp, sp, #12
        ldr     r3, _L4
        mov     r2, #0
        str     r2, [r3, #0]
        mov     r3, #0
        str     r3, [fp, #-8]
        b       _L2
_L3:
        ldr     r3, _L4+4
        ldr     r2, [fp, #-8]
        ldr     r1, [fp, #-8]
        str     r1, [r3, r2, asl #2]
        ldr     r3, _L4
        ldr     r2, [r3, #0]
        ldr     r3, [fp, #-8]
        add     r2, r2, r3
        ldr     r3, _L4
        str     r2, [r3, #0]
        ldr     r3, [fp, #-8]
        add     r3, r3, #1
        str     r3, [fp, #-8]
_L2:
        ldr     r3, [fp, #-8]
        cmp     r3, #9
        ble     _L3
        mov     r0, r3
        add     sp, fp, #0
        ldmfd   sp!, {fp}
        bx      lr
_L5:
_L4:
        .word   total
        .word   array
```

Paste and press [Compile].

Don't choose optimize yet.

Then take the generated assembly code and paste it into the miniARM assembler.

Unoptimized Compiler Output

Why is this code so bad?

Because it generated for debugging.
Essentially, each line is translated directly.

```
.word 0x03fffffc, main
array:  .space 10
total:  .space 1
        .global main
main:
        str     fp, [sp, #-4]!
        add     fp, sp, #0
        sub     sp, sp, #12
        ldr     r3, _L4
        mov     r2, #0                  total = 0;
        str     r2, [r3, #0]
        mov     r3, #0
        str     r3, [fp, #-8]           for(i = 0; ; )
        b       _L2
_L3:
        ldr     r3, _L4+4
        ldr     r2, [fp, #-8]           array[i] = i;
        ldr     r1, [fp, #-8]
        str     r1, [r3, r2, asl #2]
        ldr     r3, _L4
        ldr     r2, [r3, #0]            total += i;
        ldr     r3, [fp, #-8]
        add     r2, r2, r3
        ldr     r3, _L4
        str     r2, [r3, #0]
        ldr     r3, [fp, #-8]
        add     r3, r3, #1              for( ; ;i++ )
        str     r3, [fp, #-8]
```


```
_L2:
        ldr     r3, [fp, #-8]           for( ; i < 10; )
        cmp     r3, #9
        ble     _L3
        mov     r0, r3
        add     sp, fp, #0
        ldmfd   sp!, {fp}
        bx      lr
_L5:
_L4:
        .word   total
        .word   array
```

Optimized Code

```
                .word 0x03fffffc, main
                .global main
main:
                ldr     r2, _L4
                mov     r3, #0
_L2:
                str     r3, [r2, #4]!
                add     r3, r3, #1
                cmp     r3, #10
                bne     _L2
                mov     r2, #45
                ldr     r3, _L4+4
                str     r2, [r3, #0]
*               bx      lr
_L5:
_L4:
                .word   array-4
                .word   total
total:          .space 1
array:          .space 10
```

The compiler wrote code that is both smaller and faster than our version. Generally, moodern C-compiler optimizers favor speed over code size.

It even relaid out the variables so that all writes are sequential,

r3 = i;
r2 = &total;

It precomputed total!

Let's try our old friend

C

```
int gcd(a,b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}

int x = 35;
int y = 55;
int z;

void main() {
    z = gcd(x, y);
}
```

Human

```
main:   ldr     r0,x
        ldr     r1,y
        bl      GCD
        str     r0,z
halt:   b       halt

x:      .word 35
y:      .word 55
z:      .word 0

GCD:    cmp     r0,r1
        bxeq    lr
        subgt   r0,r0,r1
        sublt   r1,r1,r0
        b       GCD
```

The compiler wrote code that is both more compliant and portable than our version.
Still a five-instruction loop, but test is moved to the end.
A push, but we win on code size.

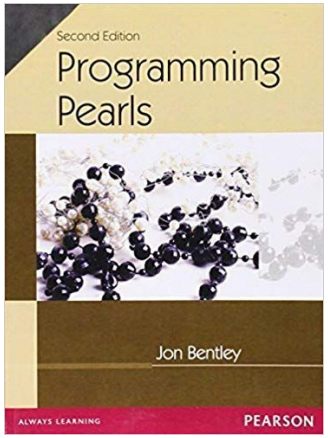
Compiler

```
        .word 0x03fffffc, main
        .global gcd
gcd:
        cmp     r0, r1
        bxeq    lr
_L6:
        cmp     r0, r1
        rsbgt   r0, r1, r0
        rsble   r1, r0, r1
        cmp     r1, r0
        bne     _L6
        bx      lr
        .global main
main:
        stmfd   sp!, {r3, lr}
        ldr     r3, _L9
        ldr     r0, [r3, #0]
        ldr     r1, [r3, #4]
        bl      gcd
        ldr     r3, _L9+4
        str     r0, [r3, #0]
        ldmfd   sp!, {r3, lr}
        bx      lr
_L10:
_L9:
        .word   _LANCHOR0
        .word   z
z:      .space 1
        .global y
        .global x
_LANCHOR0:
x:
        .word   35
y:
        .word   55
```

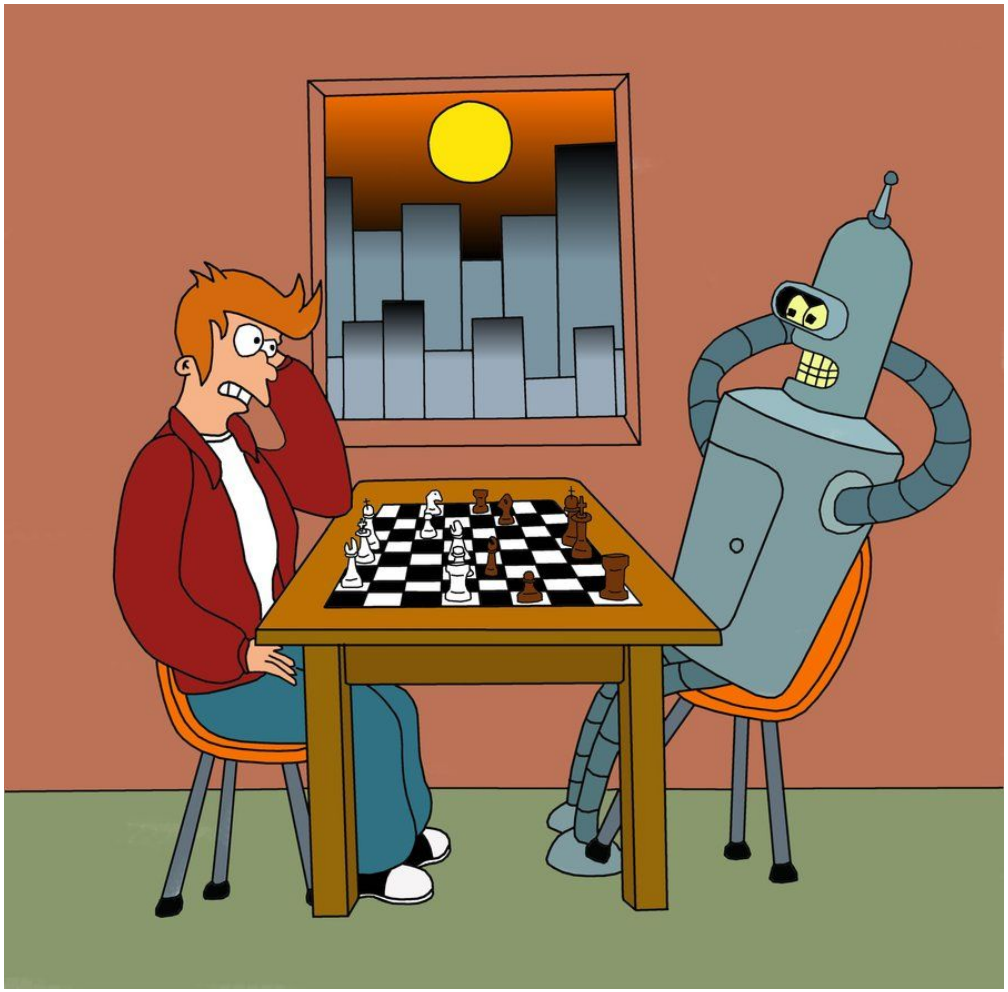
More of a Challenge

```
void swap(int *x, int i, int j) {
   int t = x[i];
   x[i] = x[j];
   x[j] = t;
}

void quicksort(int *x, int lo, int hi) {
    int i, pivot;
    if (lo >= hi) return;
    pivot = lo;
    for (i = lo+1; i <= hi; i++)
        if (x[i] < x[lo]) {
            pivot += 1;
            swap(x, pivot, i);
        }
    swap(x, lo, pivot);
    quicksort(x, lo, pivot-1);
    quicksort(x, pivot+1, hi);
}

int array[10] = {7,29,19,61,12,3,19,68,42,0};

void main() {
    quicksort(array,0,9);
}
```

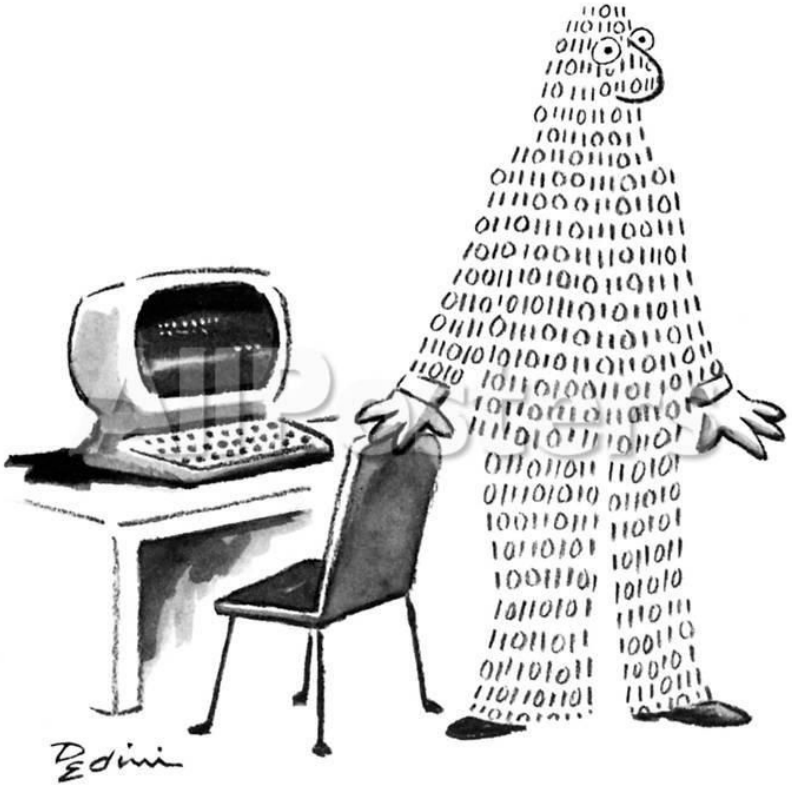
One of the most elegant pieces of code I've ever seen written by Jon Bentley, a UNC Alum, and featured in his book "Programming Pearls"



Man vs Machine



Next Time



We look into the hardware