Mixed Language Programming

>> Very few applications written entirely in assembly

-- Almost all chips come with development environments that support C

-- Assembly instructions sets still highly chip specific

-- Straight assembly still typically faster but modern C cross-assemblers aren't bad

-- Often cheaper to throw HW at problem then extend SW development cycles to program in assembly

-- Go down to assembly for specific, judicious reasons

-- IO Drivers, interrupts,

-- Real-time computational constraints (DSP)
  (popular chips likely for have commercial math libraries)

Example: C code generates “fatter” assembly

```
.model small
.data
nums dw 9,19,29,29,49
cnt dw 5
res dw 0

.code
.8086

summer: ; Address of numbers passed in bx
    push cx
    push ax
    mov cx, cnt
    mov ax, 0
lp:    add ax, [bx]
    inc bx
    inc bx
    loop lp
    mov res, ax;
    ret

main:  mov bx, offset nums
        call summer
        mov ax, 4C00h
        int 21h
        end main
```
long summer(int* in_a, int cnt) // summer.c
{
  int i;
  long sum;

  sum = 0;
  for (i = 0; i < cnt; i++)
    sum += in_a[i];
  return sum;
}

main()
{
  int nums[5];
  int cnt;
  long result;

  cnt = 5;
  nums[0] = 9;
  nums[1] = 19;
  nums[2] = 29;
  nums[3] = 39;
  nums[4] = 49;
  result = summer(nums, cnt);
  return 0;
}

;summer.asm

ifndef ??version
?debug macro
endm
$comm macro name,dist,size,count
  comm dist name:BYTE:count*size
endm
else
$comm macro name,dist,size,count
  comm dist name[size]:BYTE:count
endm
endif
?debug S "summer.c"
?debug C E9654485330873756D6D65722E63
_TEXT segment byte public 'CODE'
_TEXT ends
DGROUP group_DATA,_BSS
  assume cs:_TEXT,ds:DGROUP
_DATA segment word public 'DATA'
d@ label byte
d@w label word
_DATA ends
_BSS segment word public 'BSS'
b@ label byte
b@w label word
_BSS ends
_TEXT segment byte public 'CODE'
;
; long summer(int* in_a, int cnt)

assume cs:_TEXT

_summer proc near
push bp
mov bp,sp
sub sp,4
push si
mov word ptr [bp-2],0
mov word ptr [bp-4],0
xor si,si
jmp short @1@114

@1@58:
mov ax,si
shl ax,1
mov bx,word ptr [bp+4]
add bx,ax
mov ax,word ptr [bx]
cwd
add word ptr [bp-4],ax
adc word ptr [bp-2],dx
inc si

@1@114:
cmp si,word ptr [bp+6]
jl short @1@58
mov dx,word ptr [bp-2]
mov ax,word ptr [bp-4]
jmp short @1@170

@1@170:
pop si
mov sp,bp
pop bp
ret

_summer endp

assume cs:_TEXT

_main proc near
push bp
mov bp,sp
sub sp,16

mov word ptr [bp-12],5
mov word ptr [bp-10],9
mov word ptr [bp-8],19
mov word ptr [bp-6],29
mov word ptr [bp-4],39
mov word ptr [bp-2],49
push word ptr [bp-12]
lea ax,word ptr [bp-10]
push ax
call near ptr _summer
pop cx
pop cx
mov word ptr [bp-14],dx
mov word ptr [bp-16],ax
In-line Assembly

>> Many C compilers have capability to assemble/compile in-line

>> Turbo C: asm
  -- Can use most 8086 instructions
  -- Must use C style comments

main()
{
  asm { /* Bracket MUST appear on same line */
    push cx
    mov cx, 123h
    sub cx, 57h
    pop cx
  }
  ....
  /* more main */
}
Can also use asm for just one line

```c
myfunc()
{
    int i;
    int x;

    if (i > 0)
        asm mov x,4
    else
        i = 7;
}
```

Can use labels and jumps

```c
int fun_x()
{
    a:    /* This is the goto label "a" */
        ...
    asm jmp a /* Goes to label "a" */
        ...
}
```

Visual C: _asm

```
void main()
{
    char my_a;

    _asm
    {
        mov ah, 8 ; read key no echo
        int 21h
        cmp al,'0'
        jb sm_big
        cmp al,'9'
        ja sm_big
        mov my_a,al ; save digit in my_a
        mov dl, al ; echo 0-9 only
        mov ah, 2
        int 21h
    sm_big:
    }
}```
Others compilers might not use standard 8086 assembly

-- GCC, the GNU C Compiler for Linux, uses AT&T\&sol; UNIX assembly syntax. Here's example using AT&T syntax

<table>
<thead>
<tr>
<th>INTEL</th>
<th>AT&amp;T</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov eax,1</td>
<td>movl $1,%eax</td>
</tr>
<tr>
<td>mov ebx,0ffh</td>
<td>movl $0xff,%ebx</td>
</tr>
<tr>
<td>int 80h</td>
<td>int $0x80</td>
</tr>
</tbody>
</table>

asm("movl %ecx %eax"); /* moves the contents of ecx to eax */

Assembly lines are “Interpreted Strings”

Linking Separately Assembled/Compiled Modules

Necessary assembler directives inside TASM

.model C (.model small, C)

Announce functions and variables public to other modules

public myfunction
public myVar

As well as variables and functions imported from other modules

extern otherFl : proc
extern externBBB : byte

Then write assembly modules

.data
myVar dw ?
bbb db 10 ; bbb is not Public

.code
.8086
myFunction: ; does something

... ret

Assemble with TASM using /mx to accommodate case-sensitivity

>> tasm /l /mx myModule.asm

-- This will generate .lst and .obj files

At top of C code modules or in header file

extern unsigned unsigned myVar;
extern unsigned long timer;
extern unsigned char font[256];

extern int key_pressed();
extern char read_key();

Compile with (and link) tcc

>> tcc myStuff.c myModule.obj

**Profiling C Code**

>> Many profilers available: gprof and Xprofiler (gcc)

GlowCode and others for Visual C++

>> Identifies location(s) of run-time bottlenecks

>> Target those areas for implementation in Assembly