Control Flow

• We know how to move data:
  – between registers
  – between registers and memory

• We have a basic template for how an
  assembly program should work

• What about when the code doesn’t run in a
  straight line?
  – if, else, loops, function calls

instruction pointer %eip

• %eip – contains address of next instruction
• doesn’t always increase sequentially:
  – when it goes up doesn’t always go up by fixed
    amount
    • opcodes, operands, different sizes
  – we may branch.

some things that you know in C

• assignment
• simple arithmetic operations
• functions
• conditionals
• loops

chapter 3 part 3
same thing in assembly

```assembly
  movl a, %eax
  movl b, %ebx
  cmpl %eax, %ebx
  je truestuff

  /* false stuff here */
  jmp DoneEitherWay

truestuff:
  /* if true code here */
DoneEitherWay:
  /* done whether true or false */
```

absdiff

```c
  int absdiff(int x, int y)
  {
    if (x<y)
      return y-x;
    return x-y;
  }
```

types of branches

- unconditional
  - jumps
  - calls
  - interrupts
- conditional jumps
  - "conditional", e.g. jump if \( x < y \)

general form of an if statement

```c
  if (a == b)
  {
    /* true stuff here */
  }
else
  {
    /* false stuff here */
  }
/* done either way (i.e., whether true or false) */
```
What does this code do?

absdiff

```c
int absdiff(int x, int y)
{
    int rval;
    if (x < y)
        goto less;
    rval = y - x;
    goto done;
    less:
    rval = y - x;
    done:
    return rval;
}
```

Returns the larger of the two arguments

```c
int absdiff(int x, int y)
{
    if (x < y)
        return y - x;
    return x - y;
}
```
loops

```c
while (a < b)
{
    /* loop body */
}
```

loop: same in assembly

```asm
while (a < b)
{
    /* loop body */
}
```

toupper

```c
int to_upper(int c)
{
    int to_upper:
    .section .text
    .globl to_upper
    .type to_upper, @function
    pushl %ebp
    movl %esp, %ebp
    xorl %eax, %eax
    movl 8(%ebp), %eax
    cmpl $122, %eax
    jg done
    cmpl $97, %eax
    jl done
    addl $-32, %eax
    done:
    movl %ebp, %esp
    popl %ebp
    ret
}
```

some things that you know in C

- assignment
- simple arithmetic operations
- functions
- conditionals
- loops
int sum(int A[], int len)

%eax - current sum
%edx - &A[0]
%ecx - i

reversing an array

int rev(int A[], int len) {
int *p=A,
q=A+len-1,
t1, t2;
while (p<q) {
    t1=*p;
    t2=*q;
    *q=t1;
    *p=t2;
    p++;
    q--;
}
}

eax - &A[0], then t1 ebx - t2 ecx - p edx - q

power

from the PGU book

the power function

int sum(int A[], int len) {
    .section .text
    .globl sum
    .type sum, @function

    sum:
    pushl %ebp
    movl %esp, %ebp
    movl $0, %ecx
    movl 8(%ebp), %edx
    jmp begin

    done:
    movl %ebp, %esp
    popl %ebp
    ret
}

int rev(int A[], int len) {
    .section .text
    .globl rev_array
    .type rev_array, @function

    rev_array:
    pushl %ebp
    movl %esp, %ebp
    movl 12(%ebp), %eax
    leal (%ecx,%edx,4), %edx
    begin:
        cmpl 12(%ebp), %ecx
        jge done
        movl (%ecx), %eax
        movl (%edx), %ebx
        movl %eax, (%edx)
        movl %ebx, (%ecx)
        addl $4, %ecx
        addl $-4, %edx
    jmp begin
    done:
    popl %ebx
    movl %ebp, %esp
    popl %ebp
    ret
loop instruction

- uses %ecx for loop control
- basic form:
  ```
  mov counter val into %ecx
  lb:
  do stuff
  loop lb
  ```
- instruction also decrements %ecx

toupper
```
```c

int to_upper(int c) {
    if (c > 122 || c < 97)
        return c;
    return c-32;
}
```

str_to_upper
```
```c

void str_to_upper(char *str) {
    char *s = str;
    while (*s != '\0') {
        *s = to_upper(*s);
        s++;
    }
}
```
one solution

```assembly
/* file LoopInstrTst.s */

section .data

msg:
.asciz "ecx = %d\n"

section .text

.equ LINUX_SYSCALL, 0x80
.equ EXIT_SYSCALL, 1

.globl _start

_start:
    nop
    movl $30, %ecx

loop_begin:
    pushl %ecx
    pushl %ecx
    pushl $msg
    call printf
    addl $8, %esp
    popl %ecx

loop loop_begin

/* loop instruction uses ecx as a counter 
   don’t need to decl ecx. */

movl $EXIT_SYSCALL, %eax
movl $0, %ebx
int $LINUX_SYSCALL
```

another possibility

```assembly
/* file LoopInstrTst.s */

section .data

msg:
.asciz "ecx = %d\n"

section .text

.equ LINUX_SYSCALL, 0x80
.equ EXIT_SYSCALL, 1

.globl _start

_start:
    nop
    movl $30, %ecx

loop_begin:
    pushl %ebp
    movl %esp, %ebp

loop_loop_begin:
    /* use eax as counter */
    movl $0, %eax
    /* %edi now contains the address of the string */
    movl 8(%ebp), %edi

loop_beginning:
    /* have we found the null character? */
    cmpb $0, (%edi)
    je exit
    /* increment counter */
    incl %eax
    /* increment counter */
    incl %edi

jmp loop_beginning

exit:
    movl %ebp, %esp
    popl %ebp
    ret
```

loop instruction example

```assembly
/* file LoopInstrTst.s */

section .data

msg:
.asciz "ecx = %d\n"

section .text

.equ LINUX_SYSCALL, 0x80
.equ EXIT_SYSCALL, 1

.globl _start

_start:
    nop
    movl $30, %ecx

loop_begin:
    pushl %ecx
    pushl %ecx
    pushl $msg
    call printf
    addi $8, %esp
    popl %ecx

loop loop_begin
    /* loop instruction uses ecx as a counter 
       don’t need to decl ecx. */

movl $EXIT_SYSCALL, %eax
movl $0, %ebx
int $LINUX_SYSCALL
```

fake quiz

- implement the strlen( ) function in assembly

output same as:

```c
for (i=30; i>0; i--)
    printf("ecx = %d\n", i);
```
conditional jump

- **example**: `cmp l Y, X`

- `je`  jump if they were equal
- `jg`  jump if the 2nd was greater than the first
- `jge` jump if the 2nd was greater or equal than the first
- `jl`  jump if the 2nd was less than the first
- `jle` jump if the 2nd was less than or equal to the first

- be careful about the operand order

---

FLAGS

- many others. we care most about:

| CF | carry flag |
| ZF | zero flag  |
| SF | sign flag  |
| OF | overflow flag |

- automatically set for ops on other registers
- `addl src,dst`
  - `CF` set if carry past most significant bit (unsigned overflow)
  - `ZF` set if `src+dst == 0`
  - `SF` set if `src+dst<0`
  - `OF` set if signed overflow
    - `s>0, d>0, s+d<0 || s<0, d<0, s+d>=0`

---

How are conditional jumps implemented?

- do some kind of comparison (testx, cmpx)
- bits set in EFLAGS register
- conditional branch instruction
  - reads state of relevant flags
setx instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Effect</th>
<th>Set Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete D</td>
<td>setz</td>
<td>D ← ZF</td>
<td>equal/zero</td>
</tr>
<tr>
<td>setne D</td>
<td>setnz</td>
<td>D ← ~ZF</td>
<td>not equal/not zero</td>
</tr>
<tr>
<td>seti D</td>
<td>seti</td>
<td>D ← SF</td>
<td>negative</td>
</tr>
<tr>
<td>setns D</td>
<td>setns</td>
<td>D ← ~SF</td>
<td>non-negative</td>
</tr>
<tr>
<td>setg D</td>
<td>setgle</td>
<td>D ← (SF ∧ OF) &amp; ~ZF</td>
<td>greater (signed &gt;)</td>
</tr>
<tr>
<td>setge D</td>
<td>setge</td>
<td>D ← (SF ∧ OF)</td>
<td>greater or equal (signed &gt;=)</td>
</tr>
<tr>
<td>setl D</td>
<td>setle</td>
<td>D ← SF ∧ OF</td>
<td>less (signed &lt;)</td>
</tr>
<tr>
<td>setle D</td>
<td>setl</td>
<td>D ← (SF ∧ OF)</td>
<td>less or equal (signed &lt;=)</td>
</tr>
<tr>
<td>seta D</td>
<td>setabe</td>
<td>D ← CF &amp; ~ZF</td>
<td>above (unsigned &gt;)</td>
</tr>
<tr>
<td>setae D</td>
<td>setae</td>
<td>D ← ~CF</td>
<td>above or equal (unsigned &gt;=)</td>
</tr>
<tr>
<td>setb D</td>
<td>setbe</td>
<td>D ← CF</td>
<td>below (unsigned &lt;)</td>
</tr>
<tr>
<td>setbe D</td>
<td>setbe</td>
<td>D ← CF</td>
<td>below or equal (unsigned &lt;=)</td>
</tr>
</tbody>
</table>

- sets destination register $D$ to 0 or 1 depending on state of flags

jumps based on %EFLAGS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Jump Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp label</td>
<td>label</td>
<td>1</td>
<td>direct jump</td>
</tr>
<tr>
<td>jmp *operand</td>
<td>label</td>
<td>1</td>
<td>indirect jump</td>
</tr>
<tr>
<td>je label</td>
<td>label</td>
<td>ZF</td>
<td>equal/zero</td>
</tr>
<tr>
<td>jne label</td>
<td>label</td>
<td>~ZF</td>
<td>not equal/zero</td>
</tr>
<tr>
<td>js label</td>
<td>label</td>
<td>SF</td>
<td>negative</td>
</tr>
<tr>
<td>jns label</td>
<td>label</td>
<td>~SF</td>
<td>non-negative</td>
</tr>
<tr>
<td>jge label</td>
<td>label</td>
<td>(SF ∧ OF) &amp; ~ZF</td>
<td>greater (signed &gt;)</td>
</tr>
<tr>
<td>jg label</td>
<td>label</td>
<td>~SF ∧ OF</td>
<td>greater or equal (signed &gt;=)</td>
</tr>
<tr>
<td>jl label</td>
<td>label</td>
<td>SF ∧ OF</td>
<td>less (signed &lt;)</td>
</tr>
<tr>
<td>jle label</td>
<td>label</td>
<td>(SF ∧ OF)</td>
<td>less or equal (signed &lt;=)</td>
</tr>
<tr>
<td>ja label</td>
<td>label</td>
<td>~CF &amp; ~ZF</td>
<td>above (unsigned &gt;)</td>
</tr>
<tr>
<td>jae label</td>
<td>label</td>
<td>~CF</td>
<td>above or equal (unsigned &gt;=)</td>
</tr>
<tr>
<td>jb label</td>
<td>label</td>
<td>CF</td>
<td>below (unsigned &lt;)</td>
</tr>
<tr>
<td>jbe label</td>
<td>label</td>
<td>CF</td>
<td>below or equal (unsigned &lt;=)</td>
</tr>
</tbody>
</table>

- note similarities to the set instructions

how are bits in %EFLAGS set?

- side effect of other ops
  - e.g., add, sub, xor
- test and cmp ops
  - testx A, B does A&B
  - cmpx A, B does B–A
  - where x can be b,w,l etc.
be careful about operand order

- gas operand order reverse of
  - Intel manuals
  - many other assemblers
- cmpl %ebx, %eax and then j1 to jump if:
  - %eax < %ebx NOT
  - %ebx < %eax

arrays

- book sec 3.8:
  - extended discussion about array layout in memory
- should remember this stuff from earlier in the semester

finding max

- if a is in %eax, and b is in %ebx

a < b ???
#multidimensional arrays

```c
#include <stdio.h>

#define NUMROWS 4
#define NUMCOLS 3

int main(int argc, char **argv)
{
    int A[NUMROWS][NUMCOLS], i, j;

    printf("\n\n|   int A[%d][%d] = \n|   %d   %d   %d
|   %d   %d   %d
|   %d   %d   %d
|   %d   %d   %d
|\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n}\n```

#example

```c
#include <stdio.h>

define LEN 5
typedef struct {
    char s[10];
    int i;
    float f;
} junk;

int main(int argc, char **argv)
{
    junk junkA[LEN];

    printf("\n\n|   junk A[%d] = \n|   %d   %d   %d
|   %d   %d   %d
|   %d   %d   %d
|   %d   %d   %d
|\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n```

#output

```c
0xbfffeb8 0xbfffebcc 0xbfffeb4d0 0xbfffeb4d4 0xbfffeb4d8

0xbfffeb4c3 0xbfffeb4c4 0xbfffeb4c5 0xbfffeb4c6 0xbfffeb4c7

0xbfffeb45c 0xbfffeb470 0xbfffeb484 0xbfffeb498 0xbfffeb4ac
```

an assembly example

```asm
; file SimpleArr.s
.data
msg1: .asciz "A[%d]=%d\n"
msg2: .asciz "Done.\n"
text
.globl _start
.equ SIZE, 10
_start:
push ebp
movl %esp, ebp
/* add space on stack for array of 10 ints */
subl $40, %esp
leal (%esp), %ebx
movl $0, %edi
init_loop_beginning:
movl %edi, (%ebx, %edi, 4)
incl %edi
cmpl $SIZE, %edi
jl init_loop_beginning
movl $0, %edi
print_loop_beginning:
pushl %edi
pushl (%ebx, %edi, 4)
pushl $msg1
call printf
addl $12, %esp
popl %ebx
movl $1, %eax
movl $0, %ebx
int $0x80
```
we already know, but how can we tell?

```c
#include "car.h"
#include <stdio.h>

void printCar(car *c)
{
    printf("make:%s\n", c->make);
    printf("model:%s\n", c->model);
    printf("year:%d\n", c->year);
    printf("city MPG:%d\n", c->cmpg);
    printf("highway MPG:%d\n", c->hmpg);
}

void printCarLong(car *c)
{
    printf("make[%p]:%s\n", (void*)&(c->make), c->make);
    printf("model[%p]:%s\n", (void*)&(c->model), c->model);
    printf("year[%p]:%d\n", (void*)&(c->year), c->year);
    printf("city MPG[%p]:%d\n", (void*)&(c->cmpg), c->cmpg);
    printf("highway MPG[%p]:%d\n", (void*)&(c->hmpg), c->hmpg);
}
```

---

a main( ) to test

```c
#include "car.h"

int main(int argc, char **argv)
{
    car c1 = {"Toyota", "Prius", 2004, 35, 50};
    car c2 = {"Ford", "Expedition", 2004, 8, 18};
    printCarLong(&c1);
    printCarLong(&c2);
    return 0;
}
```

---

output

<table>
<thead>
<tr>
<th>0xbffff438</th>
<th>0xbffff43c</th>
<th>0xbffff440</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbffff444</td>
<td>0xbffff448</td>
<td>0xbffff44c</td>
</tr>
<tr>
<td>0xbffff450</td>
<td>0xbffff454</td>
<td>0xbffff458</td>
</tr>
<tr>
<td>0xbffff45c</td>
<td>0xbffff460</td>
<td>0xbffff464</td>
</tr>
</tbody>
</table>

---

structs

- should remember from earlier in semester
- How would a car be laid out in memory?
unions

3.10 alignment

- store data at address that’s a multiple of some number of bytes (e.g., the word size)
- required by certain machines
  - different processors, OSs, compilers have different rules
- why?
  - suppose CPU can fetch word size (e.g. 4 bytes) at a time
  - I want to fetch a single int
  - If data is word-aligned, it requires 1 fetch
  - Otherwise 2 fetches

the output

```c
1 make[0xbffff4b4]:Toyota
2 model[0xbffff4c3]:Prius
3 year[0xbffff4d4]:2004
4 city MPG[0xbffff4d8]:35
5 highway MPG[0xbffff4dc]:50
6 make[0xbffff488]:Ford
7 model[0xbffff497]:Expedition
8 year[0xbffff4a8]:2004
9 city MPG[0xbffff4ac]:8
10 highway MPG[0xbffff4b0]:18
```

- is this what we’d expect???

structs and assembly

- Example from Programming from the Ground Up
- struct containing:
  - first name – 40 bytes
  - last name – 40 bytes
  - address – 240 bytes
  - age – 4 bytes
for primitive data types (i.e., not structs)

• depends on size of data type.

<table>
<thead>
<tr>
<th>size</th>
<th>type</th>
<th>lowest bits of address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>char</td>
<td>no restrictions</td>
</tr>
<tr>
<td>2</td>
<td>short</td>
<td>lowest bit=02</td>
</tr>
<tr>
<td>4</td>
<td>int, float, pointer</td>
<td>lowest two bits=002</td>
</tr>
<tr>
<td>8</td>
<td>double</td>
<td>Windows - lowest 3 bits=0002</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Linux - lowest 2 bits=002</td>
</tr>
</tbody>
</table>

alignment with structs

• elements within a struct:
  – must satisfy element’s alignment requirement
  – i.e., the requirements from the previous table

• placement of entire struct
  – structure itself has alignment requirement $K$
    • where $K$=largest alignment of any element
  – Initial addr, struct length multiples of $K$

how?

• If my data isn’t naturally aligned properly?
  – compiler inserts padding

alignment rules

• Windows:
  – All types aligned to address equal to its size (e.g., double aligned to 8 byte boundary)
inline assembly

- embed assembly in your C programs
- why?
  - access processor features not available to C
  - assembly code in OS
    - in Linux kernel source, many examples under arch (duh), and drivers

two forms

```c
void someCFunc( )
{
  ...
  asm("assembly here");
  /* OR */
  __asm__("assembly here");
}
```

- __asm__ form useful if “asm” conflicts with something else in your program

example. what do we expect?

```c
/* file sizeofstst.c */
#include <stdio.h>

typedef struct {
  int x;
  char c[5];
} foo;

typedef struct {
  int x;
  char c;
} goo;

int main(int argc, char **argv)
{
  foo F[10];
  goo G[10];

  printf("sizeof foo = %lu
",
         sizeof(F[0]));

  printf("sizeof F[] = %lu
",
         sizeof(F));

  printf("sizeof goo = %lu
",
         sizeof(G[0]));

  printf("sizeof G[] = %lu
",
         sizeof(G));

  return 0;
}
```

output

```c
/* file sizeofstst.c */
#include <stdio.h>

typedef struct {
  int x;
  char c[5];
} foo;

typedef struct {
  int x;
  char c;
} goo;

We get:

  sizeof foo = 12
  sizeof F[] = 120
  sizeof goo = 8
  sizeof G[] = 80
```
big problem

• what happens if your inline assembly changes registers that the C compiler is using?

• bad things happen

solution. extended asm

• tell the C compiler what you’re doing
  – regs being used
  – hints about what variables should be stored in which registers

effects

• asm(“movl $25, %eax”);
• asm(“movl $25, %eax\n\ntcmpl %eax, ebx”);
• asm (“pushl %eax\n\nt”
  “movl $0, %eax\n\nt”
  “popl %eax”);

from the Linux kernel

• from linux-2.6.29.1
  – /arch/x86/boot/tty.c

```c
unsigned char c = ch;
if (c == \n‘\n’)
  putchar(‘\r’);
/* \n \n */
/* int $0x10 is known to have bugs involving touching registers
  it shouldn’t. Be extra conservative... */
asm volatile(“pushal; pushw %%ds; int $0x10; popw %%ds; popal"
: : “b” (0x0007), “c” (0x0001), “a” (0x0000|ch));```

examples
input/output list syntax

- "<constraint>" (<variable>)
- where <constraint> could be, for example
  - which register to use
  - not to use a register, but use memory instead
  - use a general purpose register
  - etc.
- <variable> name of a C variable to which reg corresponds

common constraints

<table>
<thead>
<tr>
<th>char</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>use any general purpose register</td>
</tr>
<tr>
<td>'b'</td>
<td>%eax, %ax, or %al</td>
</tr>
<tr>
<td>'c'</td>
<td>%ebx, %bx, or %bl</td>
</tr>
<tr>
<td>'d'</td>
<td>%ecx, %cx, or %cl</td>
</tr>
<tr>
<td>'e'</td>
<td>%edx, %dx, or %dl</td>
</tr>
<tr>
<td>'S'</td>
<td>%esi</td>
</tr>
<tr>
<td>'D'</td>
<td>%edi</td>
</tr>
<tr>
<td>'m'</td>
<td>use memory, not a register</td>
</tr>
<tr>
<td>'0', '1', ..., '9'</td>
<td>use some variable that you've defined earlier in some list, e.g. asm(&quot;addl %0, %0&quot;: &quot;a&quot; (var) : &quot;0&quot; (var)). In this case '0' refers to the first variable, 'var', in the output list.</td>
</tr>
</tbody>
</table>

extended asm: some syntax

- asm("assembly template": <output>: <input>: <ClobberedRegs>)
  - output: comma separated list of output operands
  - input: comma separated list of input operands
  - ClobberedRegs: comma separated list of registers which you modify in the assembly

assembly “template” syntax

- placeholders: %0, %1, %2, etc. map to first, second, ... variables regs in the output, input lists.
  - think of printf format characters
- because we're using '%' in the placeholders, if we want a '%' character to appear in the assembly, we write "%%", e.g., we'd write "% %eax" in the assembly string.
CPUID from C using extended asm

```c
char *getCPUID()
{
    int func=0, bx, cx, dx;

    char *cpuidstr;
    if ((cpuidstr = (char*)calloc(CPUID_STRLEN, 1))==NULL)
        return NULL;

    asm("cpuid"
        : "=b" (bx),
           "=c" (cx),
           "=d" (dx)
        : "a" (func));

    memcpy(cpuidstr, &bx, 4);
    memcpy(&cpuidstr[4], &dx, 4);
    memcpy(&cpuidstr[8], &cx, 4);
    cpuidstr[CPUID_STRLEN-1] = '\0';
    return cpuidstr;
}
```

maybe some useful information

- GCC inline assembly howto
- GCC manual
  - http://gcc.gnu.org/onlinedocs/gcc-4.4.0/gcc/Extended-Asm.html#Extended-Asm
- Article on IBM developerworks
- Brennan’s Guide to Inline Assembly

callback our CPUID code

```c
1 /* file cpuid2.s */
2 .section .data
3 .asciz "The processor Vendor ID is '%s'\n"
4 .section .bss
5 .lcomm buffer, 12
6 .section .text
7 .globl _start
8 _start:
9     movl $0, %eax
10    cpuid
11    movl $buffer, %edi
12    movl %ebx, (%edi)
13    movl %edx, 4(%edi)
14    movl %ecx, 8(%edi)
15    pushl $buffer
16    pushl $output
17    call printf
18    addl $8, %esp
19    pushl $0
20    call exit
```