chapter 3 part 3
some things that you know in C

• assignment
• simple arithmetic operations
• functions
• conditionals
• loops
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.*
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) */
```
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;
}
```
int absdiff(int x, int y)
{
    if (x < y)
        return y - x;
    return x - y;
}
int absdiff(int x, int y) {
    if (x < y)
        goto less;
    return y - x;
    return x - y;
}

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;
}
What does this code do?

```assembly
.section .text
.globl max
.type max, @function
max:
    pushl %ebp
    movl %esp, %ebp
    movl $0, %eax
    movl 8(%ebp), %ecx
    movl 12(%ebp), %edx
    cmpl %ecx, %edx
    jge L1
    movl %edx, %eax
    jmp done
L1:
    movl %edx, %eax
done:
    movl %ebp, %esp
    popl %ebp
ret
```
What does this code do?

Returns the larger of the two arguments

```
1 .section .text
2 .globl max
3 .type max, @function
4 max:
5 pushl %ebp
6 movl %esp, %ebp
7 movl $0, %eax
8 movl 8(%ebp), %ecx
9 movl 12(%ebp), %edx
10 cmpl %ecx, %edx
11 jge L1
12 movl %ecx, %eax
13 jmp done
14 L1:
15 movl %edx, %eax
16 done:
17 movl %ebp, %esp
18 popl %ebp
19 ret
```
```c
int to_upper(int c) {
    if (c > 122 || c < 97) {
        return c;
    }
    return c - 32;
}
```

```assembly
.section .text
.globl to_upper
.type to_upper, @function
to_upper:
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
loops

1    while (a < b)
2        {
3            /* loop body */
4        }

15
loop: same in assembly

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

movl a, %eax
movl b, %ebx
loop_beginning:
cmpl %ebx, %eax
jge loop_ending

loop_body:

loop_ending:

jmp loop_beginning
.section .data
.section .text
.globl _start

_start:
    pushl $3          #push second argument
    pushl $2          #push first argument
    call power        #call the function
    addl $8, %esp     #move the stack pointer back

    pushl %eax        #save the first answer before
                        #calling the next function

    pushl $2          #push second argument
    pushl $5          #push first argument
    call power        #call the function
    addl $8, %esp     #move the stack pointer back

    popl %ebx         #The second answer is already
                        #in %eax. We saved the
                        #first answer onto the stack,
                        #so now we can just pop it
                        #out into %ebx

    addl %eax, %ebx   #add them together
                        #the result is in %ebx

    movl $1, %eax     #exit (%ebx is returned)
    int $0x80
the
power
function

.type power, @function

power:
pushl %ebp                     #save old base pointer
movl %esp, %ebp              #make stack pointer the base pointer
subl $4, %esp               #get room for our local storage

movl 8(%ebp), %ebx          #put first argument in %ebx
movl 12(%ebp), %ecx        #put second argument in %ecx

movl %ebx, -4(%ebp)        #store current result

power_loop_start:
cmpl $1, %ecx              #if the power is 1, we are done
je end_power
movl -4(%ebp), %eax        #move the current result into %eax
imull %ebx, %eax           #multiply the current result by
                          #the base number
movl %eax, -4(%ebp)       #store the current result

decl %ecx                  #decrease the power
jmp power_loop_start      #run for the next power

end_power:
movl -4(%ebp), %eax        #return value goes in %eax
movl %ebp, %esp           #restore the stack pointer
popl %ebp                  #restore the base pointer
ret
int sum(int A[], int len)

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

.section .text
.globl sum
.type sum, @function

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

begin:
  cmpl 12(%ebp), %ecx
  jge done
  addl (%edx, %ecx, 4), %eax
  incl %ecx
  jmp begin

done:
movl %ebp, %esp
popl %ebp
ret
reversing an array

```c
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--;
    }
}
```

```assembly
.section .text
.globl rev_array
.type rev_array, @function

rev_array:
pushl %ebp
movl %esp, %ebp
pushl %ebx
movl 12(%ebp), %eax
decl %eax
movl 8(%ebp), %ecx
leal (%ecx,%eax,4), %edx

begin:
cmpl %ecx, %edx
jle 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
```

** eax - &A[0], then t1
** ebx - t2
** ecx - p
** edx - q
```c
int to_upper(int c)
{
    if (c > 122 || c < 97)
        return c;
    return c-32;
}
```

```assembly
.section .text
.globl to_upper
.type to_upper, @function
to_upper:
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
```
void str_to_upper(char *str) {
    char *s = str;
    while (*s != '\0') {
        *s = to_upper(*s);
        s++;
    }
}

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

str_to_upper:
    pushl %ebp
    movl %esp, %ebp
    subl $4, %esp
    movl 8(%ebp), %ecx
    xorl %eax, %eax
begin:
    movb (%ecx), %al
    cmpb $0, %al
    je done
    movl %eax, -4(%ebp)
call to_upper
    movb %al, (%ecx)
incl %ecx
    jmp begin
done:
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
loop instruction

• uses %ecx for loop control

• basic form:

1. mov counter val into %ecx
2. lb:
3. do stuff
4. loop lb

• instruction also decrements %ecx
loop instruction example

/* 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 %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
loop

instruction

example

output same as:

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

/* 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
fake quiz

• implement the strlen() function in assembly
.type strlen, @function
.globl strlen

strlen:
    pushl %ebp
    movl %esp, %ebp

    /* 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 pointer */
    incl %edi
    jmp loop_beginning

exit:
    movl %ebp, %esp
    popl %ebp
    ret
another possibility

```assembly
.globl strlen2

hashalign 10

.bit

strlen2:
    pushl %ebp
    movl %esp, %ebp

    /* veax now contains the address of the begining of the string */
    movl 8(%ebp), veax

    loop_beginning:
    /* have we found the null character? */
    cmpb u0, %eax
    je exit

    /* increment counter */
    incl %eax
    jmp loop_beginning

exit:
    /* length = address of end of string stored in %eax - address of begining of the string stored in 8(%ebp) */
    subl 8(%ebp), %eax
    movl %eax, %esp
    popl %ebp
    ret
```
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
conditional jump

• example: `cmp Y, X`

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>jump if they were equal</td>
</tr>
<tr>
<td>jg</td>
<td>jump if the 2nd was greater than the first</td>
</tr>
<tr>
<td>jge</td>
<td>jump if the 2nd was greater or equal than the first</td>
</tr>
<tr>
<td>jl</td>
<td>jump if the 2nd was less than the first</td>
</tr>
<tr>
<td>jle</td>
<td>jump if the 2nd was less than or equal to the first</td>
</tr>
</tbody>
</table>

• be careful about the operand order
FLAGS

- many others. we care most about:

<table>
<thead>
<tr>
<th>CF</th>
<th>carry flag</th>
<th>ZF</th>
<th>zero flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td>sign flag</td>
<td>OF</td>
<td>overflow flag</td>
</tr>
</tbody>
</table>

- 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
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| ID | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF | VIF |

X  ID Flag (ID)
X  Virtual Interrupt Pending (VIP)
X  Virtual Interrupt Flag (VIF)
X  Alignment Check (AC)
X  Virtual-8086 Mode (VM)
X  Resume Flag (RF)
X  Nested Task (NT)
X  I/O Privilege Level (IOPL)
X  Overflow Flag (OF)
C  Direction Flag (DF)
X  Interrupt Enable Flag (IF)
X  Trap Flag (TF)
S  Sign Flag (SF)
S  Zero Flag (ZF)
S  Auxiliary Carry Flag (AF)
S  Parity Flag (PF)
S  Carry Flag (CF)

S  Indicates a Status Flag
C  Indicates a Control Flag
X  Indicates a System Flag
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.
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>sets D</td>
<td></td>
<td>D ← SF</td>
<td>negative</td>
</tr>
<tr>
<td>setns D</td>
<td></td>
<td>D ← ~ SF</td>
<td>non-negative</td>
</tr>
<tr>
<td>setg D</td>
<td>setnle</td>
<td>D ← ~(SF ^ OF) &amp; ~ZF</td>
<td>greater (signed &gt;)</td>
</tr>
<tr>
<td>setge D</td>
<td>setnl</td>
<td>D ← ~(SF ^ OF)</td>
<td>greater or equal (signed &gt;=)</td>
</tr>
<tr>
<td>setl D</td>
<td>setnge</td>
<td>D ← SF ^ OF</td>
<td>less (signed &lt;)</td>
</tr>
<tr>
<td>setle D</td>
<td>setng</td>
<td>D ← (SF ^ OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>seta D</td>
<td>setnbe</td>
<td>D ← ~CF &amp; ~ZF</td>
<td>above (unsigned &gt;)</td>
</tr>
<tr>
<td>setae D</td>
<td>setnb</td>
<td>D ← ~CF</td>
<td>above or equal (unsigned &gt;=)</td>
</tr>
<tr>
<td>setb D</td>
<td>setnae</td>
<td>D ← CF</td>
<td>below (unsigned &lt;)</td>
</tr>
<tr>
<td>setbe D</td>
<td>setna</td>
<td>D ← CF</td>
<td>ZF</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></td>
<td>1</td>
<td>direct jump</td>
</tr>
<tr>
<td>jmp *operand</td>
<td></td>
<td>1</td>
<td>indirect jump</td>
</tr>
<tr>
<td>je label</td>
<td>jz</td>
<td>ZF</td>
<td>equal/zero</td>
</tr>
<tr>
<td>jne label</td>
<td>jnz</td>
<td>~ZF</td>
<td>not equal/zero</td>
</tr>
<tr>
<td>js label</td>
<td></td>
<td>SF</td>
<td>negative</td>
</tr>
<tr>
<td>jns label</td>
<td></td>
<td>~SF</td>
<td>non-negative</td>
</tr>
<tr>
<td>jg label</td>
<td>jnle</td>
<td>~(SF ^ OF) &amp; ~ZF</td>
<td>greater (signed &gt;)</td>
</tr>
<tr>
<td>jge label</td>
<td>jnl</td>
<td>~(SF ^ OF)</td>
<td>greater or equal (signed &gt;=)</td>
</tr>
<tr>
<td>jl label</td>
<td>jnge</td>
<td>SF ^ OF</td>
<td>less (signed &lt;)</td>
</tr>
<tr>
<td>jle label</td>
<td>jng</td>
<td>(SF ^ OF)</td>
<td>less or equal (signed &lt;=)</td>
</tr>
<tr>
<td>ja label</td>
<td>jnbe</td>
<td>~CF &amp; ~ZF</td>
<td>above (unsigned &gt;)</td>
</tr>
<tr>
<td>jae label</td>
<td>jnb</td>
<td>~CF</td>
<td>above or equal (unsigned &gt;=)</td>
</tr>
<tr>
<td>jb label</td>
<td>jnae</td>
<td>CF</td>
<td>below (unsigned &lt;)</td>
</tr>
<tr>
<td>jbe label</td>
<td>jna</td>
<td>CF</td>
<td>less or equal (unsigned &lt;=)</td>
</tr>
</tbody>
</table>

Note similarities to the set instructions
/ from Programming from the Ground Up */
.data

.data_items:                   #These are the data items
   .long 3,67,34,222,45,75,54,34,44,33,22,11,66,0

.text

.globl _start

_start:
   movl $0, %edi       # move 0 into the index register
   movl data_items(%edi,4), %eax # load the first byte of data
   movl %eax, %ebx    # since this is the first item, %eax is # the biggest

.start_loop:                # start loop
   cmpl $0, %eax       # check to see if we’ve hit the end
   je loop_exit       # load next value
   incl %edi           # compare values
   movl data_items(%edi,4), %eax
   cmpl %ebx, %eax     # jump to loop beginning if the new
   jle start_loop     # one isn’t bigger

   movl %eax, %ebx     # move the value as the largest
   jmp start_loop     # jump to loop beginning

loop_exit:
   # %ebx is the status code for the exit system call
   # and it already has the maximum number
   movl $1, %eax       #1 is the exit() syscall
   int $0x80
• if a is in %eax, and b is in %ebx

```assembly
/* file aLTb.s */
.text
.equ LINUX_SYSCALL, 0x80
.equ EXIT_SYSCALL, 1
.globl _start
_start:
nop
movl $10, %eax
movl $20, %ebx
cmpl %ebx, %eax
setl %al
movzbl %al, %eax
movl %eax, %ebx
movl $EXIT_SYSCALL, %eax
int $LINUX_SYSCALL
```
be careful about operand order

• gas operand order reverse of
  – Intel manuals
  – many other assemblers

• `cmp %ebx, %eax and then jl` 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
```c
#include <stdio.h>

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

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

   printf("intA[]\n");
   for (i=0; i<LEN; i++)
      printf("%p ", (void*)&intA[i]);

   printf("\ncharA[]\n");
   for (i=0; i<LEN; i++)
      printf("%p ", (void*)&charA[i]);

   printf("\njunkA[]\n");
   for (i=0; i<LEN; i++)
      printf("%p ", (void*)&junkA[i]);

   printf("\n");
   return 0;
}
## output

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbffff4c8</td>
<td>0xbffff4cc</td>
<td>0xbffff4d0</td>
<td>0xbffff4d4</td>
<td>0xbffff4d8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbffff4c3</td>
<td>0xbffff4c4</td>
<td>0xbffff4c5</td>
<td>0xbffff4c6</td>
<td>0xbffff4c7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbffff45c</td>
<td>0xbffff470</td>
<td>0xbffff484</td>
<td>0xbffff498</td>
<td>0xbffff4ac</td>
</tr>
</tbody>
</table>
an assembly example

/* file SimpleArr.s */
data
msg1:
  .asciz "A[%d]=%d\n"
msg2:
  .asciz "Done.\n"
.text
.globl _start
.equ SIZE, 10
_start:
pushl %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

print_loop_beginning:
pushl %edi
pushl (%ebx, %edi, 4)
pushl $msg1
call printf
addl $12, %esp
incl %edi
cmpl $SIZE, %edi
jl print_loop_beginning

pushl $msg2
call printf
addl $4, %esp

/* we’re exiting anyway, but it can’t hurt */
movl %ebp, %esp
popl %ebp

movl $1, %eax
movl $0, %ebx
int $0x80
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");
    for (i=0; i<NUMROWS; i++) {
        for (j=0; j<NUMCOLS; j++)
            printf("%p", (void*)&A[i][j]);
        printf("\n");
    }

    return 0;
}
```

what do we expect?
output

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbffffff438</td>
<td>0xbffffff43c</td>
<td>0xbffffff440</td>
</tr>
<tr>
<td>0xbffffff444</td>
<td>0xbffffff448</td>
<td>0xbffffff44c</td>
</tr>
<tr>
<td>0xbffffff450</td>
<td>0xbffffff454</td>
<td>0xbffffff458</td>
</tr>
<tr>
<td>0xbffffff45c</td>
<td>0xbffffff460</td>
<td>0xbffffff464</td>
</tr>
</tbody>
</table>
structs

• should remember from earlier in semester

• How would a car be laid out in memory?

```c
/* car.h */

#ifndef __CAR_H__
#define __CAR_H__

typedef struct {
    char make[15];
    char model[15];
    int year;
    int cmpg;    /* city MPG */
    int hmpg;    /* highway MPG */
} car;

void printCar(car*);
void printCarLong(car*);

#endif
```
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;
}
```
the output

1. make[0xbfffff4b4]: Toyota
2. model[0xbfffff4c3]: Prius
3. year[0xbfffff4d4]: 2004
4. city MPG[0xbfffff4d8]: 35
5. highway MPG[0xbfffff4dc]: 50
6. make[0xbfffff488]: Ford
7. model[0xbfffff497]: Expedition
8. year[0xbfffff4a8]: 2004
9. city MPG[0xbfffff4ac]: 8
10. highway MPG[0xbfffff4b0]: 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
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
• 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)
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><strong>Windows</strong> - lowest 3 bits=000₂</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Linux</strong> - 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 $K$
example. what do we expect?

```c
/* file sizeofst.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\n",
            sizeof(F[0]));
    printf("sizeof F[] = %lu\n",
            sizeof(F));
    printf("sizeof goo = %lu\n",
            sizeof(G[0]));
    printf("sizeof G[] = %lu\n",
            sizeof(G));
    return 0;
}
```
/* file sizeofst.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
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
```
examples

• `asm("movl $25, %eax");`
• `asm("movl $25, %eax\n\ncmpl %eax, ebx");`
• `asm ("pushl %eax\n\nmovl $0, %eax\n\npopl %eax");`
from the Linux kernel

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

```c
unsigned char c = ch;

if (c == '\n')
    putchar('\r'); /* \n -> \r\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" (0x0e00|ch));
```
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
extended asm: some syntax

- \texttt{asm("assembly template" : <output> : <input> : <ClobberedRegs>)}
  - \textbf{output}: comma separated list of output operands
  - \textbf{input}: comma separated list of input operands
  - \textbf{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.
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>‘r’</td>
<td>use any general purpose register</td>
</tr>
<tr>
<td>‘a’</td>
<td>%eax, %ax, or %al</td>
</tr>
<tr>
<td>‘b’</td>
<td>%ebx, %bx, or %bl</td>
</tr>
<tr>
<td>‘c’</td>
<td>%ecx, %cx, or %cl</td>
</tr>
<tr>
<td>‘d’</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, <em>e.g.</em> <code>asm(“addl %0, %0”: “a” (var): “0” (var)).</code> In this case ‘0’ refers to the first variable, ‘var’, in the output list.</td>
</tr>
</tbody>
</table>
constraint prefixes

• constraints may be prefixed by:
  – '=': the variable is write-only (it's never read)
  – '&': variable is an "earlyclobber"
    • it will be modified before all input operands are used
    • specify so that the location used isn't used by any input operand (because the earlyclobber variable will clobber it)

• for example, if we wrote “=r”, we want any general purpose register, and in our assembly, we plan on writing, but not reading.
/ * file cpuid2.s */

.output:

.output: The processor Vendor ID is '%s'

.section .bss
.lcomm buffer, 12

.section .text
.globl _start

_start:

movl $0, %eax
cpuid
movl $buffer, %edi
movl %ebx, (%edi)
movl %edx, 4(%edi)
movl %ecx, 8(%edi)
pushl $buffer
pushl $output
call printf
addl $8, %esp
pushl $0
call exit
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;
}

CPUID from C using extended asm
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