Lecture Topics

- Today: Assembly Language Data Org. (H&H 6.3)
- Next: continued

Announcements

- Self-study Module #9
- Project #9 (due no later than 11/9)
- Project #10 (due no later than 11/21)
Exam #2

- Tuesday, 11/14 during lecture
- 18% of course grade
- Bring #2 pencils (multiple choice)
- Bring MSU ID
- One 8.5x11 note sheet allowed
- Old exam on course website

Data Movement Instructions

Two forms of data movement:

Load: RAM → Register

Store: Register → RAM

The data movement instructions allow us to copy values between a memory location and a register (and vice versa)
Store Instructions

Four different instructions:

- STRB  store byte (1 byte)
- STRH  store halfword (2 bytes)
- STR   store word (4 bytes)
- STRD  store doubleword (8 bytes)

All four operate in the same way, just transfer different numbers of bytes.

Assembly language examples

Assuming registers contain values:

- `strb r2, [r1,r5]`
- `strh r10, [r4,#8]`
- `str r7, [r5,#-4]`
- `strd r6, [r2]`

Offset missing? Implied zero.
Load Instructions

Six different instructions:

- LDRB  load unsigned byte (1 byte)
- LDRSB load signed byte (1 byte)
- LDRH  load unsigned halfword (2 bytes)
- LDRSH load signed halfword (2 bytes)
- LDR   load word (4 bytes)
- LDRD  load doubleword (8 bytes)

Assembly language examples

Assuming registers contain values:

```
  ldrb  r2, [r8,r3]
  ldrsh r5, [r4,#6]
  ldr   r1, [r5,#-8]
  ldrd  r4, [r3]
```

Offset missing? Implied zero.
Load and Store Instructions

Why support different sizes, as well as signed and unsigned loads?

High-level languages support different sizes of variables, as well as signed and unsigned integers.

```c
unsigned char A;
short int B;
unsigned long long C;
```

Composite Data Objects

Besides scalar data objects, we also have composite data objects (more than one piece of data grouped together).

Array – group of values, all with same type

Record – group of values, may be different types
Records

In C/C++, the keyword “struct” is used to indicate that you’re declaring a record.

```c
struct point
{
    char label;
    int x_coord;
    int y_coord;
};
```

```c
struct point A;
A.label = 'Q';
A.x_coord = 15;
A.y_coord = 3;
```
The compiler will create a **record mapping table** when it processes the declaration of **struct point**:

<table>
<thead>
<tr>
<th>field</th>
<th>size of field</th>
<th>offset in record</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>1 byte</td>
<td>+0</td>
</tr>
<tr>
<td>x_coord</td>
<td>4 bytes</td>
<td>+4</td>
</tr>
<tr>
<td>y_coord</td>
<td>4 bytes</td>
<td>+8</td>
</tr>
</tbody>
</table>

The offset for **x_coord** is +4 because LDR and STR will be used to access that field, so the address of that field must be a multiple of 4 (alignment restrictions).

We need a total of 12 bytes for the record (3 bytes wasted).

The compiler will allocate 12 bytes when it processes the declaration of variable **A**:

- **label** (1 byte) at +0
- **padding** (3 bytes) at +4
- **x_coord** (4 bytes) at +8
- **yCoord** (4 bytes) at +12
Example #23

Example which works with two objects of type “struct point” (from previous slides).

Course website:

~cse320/Examples/example23.pdf

Arrays

All elements in an array must be the same type (and thus the same size).

C/C++ declarations:

```c
int A[100];
char B[20];
unsigned long long C[50];
```
The compiler uses an **array mapping function** to handle array subscripting.

Assume:

```c
int I, A[100];

... A[I] ...
```

The compiler calculates the address of `A[I]`:

```
address of A + (I * size of one item)
```

where "size of one item" is the number of bytes allocated for one element of the array.

---

Assume:

```c
int I, A[100];

// Address of A + (0*4)
A[0] = -99;

// Address of A + (I*4)
```
Example #24

Example which works with an array of type “short int” (2-byte signed integers).

Course website:

~cse320/Examples/example24.pdf

Note use of “.balign 2” to align on a 2-byte address:

```assembly
.data
.balign 2
vector:
 .short 21, -45, 96, 72, -33, 67
```

Generates six 2-byte values:

```
0015 ffd3 0060 0048 ffdf 0043
```
NUM = 6

.data
.balign 2
vector:
.short 21, -45, 96, 72, -33, 67

.global main
.text
.balign 4
main:
    push {lr}

mov    r4, #0    @ r4: sum
mov    r5, #0    @ r5: index
ldr    r8, =vector    @ r8: &vector
loop:
    cmp    r5, #NUM    @ Compare index, NUM
    bge    endloop    @ exit when GE
    lsl    r6, r5, #1    @ r6: index*2
    ldrsh  r7, [r8,r6]    @ r7: array element
    add    r4, r4, r7    @ r4: update sum
    add    r5, r5, #1    @ r5: update index
    b        loop
endloop:
    ldr    r0, =fmt          @ 1st arg:  &fmt
    mov    r1, r4          @ 2nd arg:  sum
    bl     printf          @ Display the sum

    ldr    r0, =vector     @ 1st arg:  &vector
    add    r1, #NUM        @ 2nd arg:  elements
    bl     display2        @ Display memory

    pop    {lr}
    bx     lr

fmt:    .asciz "\nSum of the array elements:  %d\n"
        .balign 4

<2 lemon:~/Examples > gcc example24.s \
    /user/cse320/lib/memlib.o

<3 lemon:~/Examples > a.out

The sum of the array elements:  178

Memory contents from 00020770 to 0002077c
00020770:  0015
00020772:  ffd3
00020774:  0060
00020776:  0048
00020778:  ffdf
0002077a:  0043
Example #24 recap

Array of 6 elements, each element 2 bytes.

Access to individual elements:

```assembly
lsr r6, r5, #1 @ r6: index*2
ldrh r7, [r8,r6] @ r7: array element
```

Alternative:

```assembly
mov r6, #2 @ r6: 2
mul r6, r5, r6 @ r6: index*2
ldrh r7, [r8,r6] @ r7: array element
```

Example #25

Example which works with an array of records; each record contains a character string (array of type “char”).

Function “main” written in C and function “print_all” written in assembly.

~cse320/Examples/example25.pdf
Record for Example #25

Data for one student: name (character string) and two exam scores (integers):

```c
struct student
{
    char name[12];
    short exam1;
    short exam2;
};
```

Record mapping table for `struct student`:

<table>
<thead>
<tr>
<th>Field</th>
<th>Size of Field</th>
<th>Offset in Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>12 * 1 byte</td>
<td>+0</td>
</tr>
<tr>
<td>exam1</td>
<td>2 bytes</td>
<td>+12</td>
</tr>
<tr>
<td>exam2</td>
<td>2 bytes</td>
<td>+14</td>
</tr>
</tbody>
</table>

We need a total of 16 bytes for one record (no padding required).
The compiler will allocate 16 bytes when it processes the declaration of an object of type `struct student`:

```
0

+12

+14

name (12 bytes)

exam1 (2 bytes)

exam2 (2 bytes)
```

```
<1 lemon:~/Examples > cat example25.driver.c

#include <stdio.h>
#include "/user/cse320/lib/memlib.h"

#define MAX 8

struct student
{
    char name[12];
    short exam1;
    short exam2;
};

void print_all( struct student *, int );
```
int main()
{
    struct student list[MAX];
    int i;
    for (i=0; i<MAX; i++)
    {
        scanf( "%s %hd %hd", &list[i].name[0],
            &list[i].exam1, &list[i].exam2);
    }
    print_all( &list[0], MAX );
    display( &list[0], MAX*16, 16 );
}

<2 lemon:/Examples > gcc example25.driver.c \ 
    example25.support.s /user/cse320/lib/memlib.o

<3 lemon:/Examples > a.out < example25.data

<table>
<thead>
<tr>
<th>Student name</th>
<th>Exam 1</th>
<th>Exam 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>70</td>
<td>75</td>
</tr>
<tr>
<td>Doe</td>
<td>74</td>
<td>84</td>
</tr>
<tr>
<td>Evans</td>
<td>49</td>
<td>57</td>
</tr>
<tr>
<td>Fernandez</td>
<td>89</td>
<td>72</td>
</tr>
<tr>
<td>Gray</td>
<td>66</td>
<td>77</td>
</tr>
<tr>
<td>Green</td>
<td>85</td>
<td>92</td>
</tr>
<tr>
<td>Jones</td>
<td>81</td>
<td>83</td>
</tr>
<tr>
<td>Smith</td>
<td>50</td>
<td>62</td>
</tr>
</tbody>
</table>
Memory contents from 7e955974 to 7e9559f4
7e955974: 42 72 6f 77 6e 00 00 00 00 00 00 00 00 00 46 00 4b 00
7e955984: 44 6f 65 00 00 00 00 00 cc af f2 76 4a 00 54 00
7e955994: 45 76 61 6e 73 00 00 00 00 00 00 00 00 00 31 00 39 00
7e9559a4: 46 65 72 6e 61 6e 64 65 7a 00 00 00 59 00 48 00
7e9559b4: 47 72 61 79 00 00 00 e8 4b d9 76 42 00 4d 00
7e9559c4: 47 72 65 6e 00 95 7e 8c 51 f0 76 55 00 5c 00
7e9559d4: 4a 6f 6e 65 73 00 01 00 c8 07 01 00 51 00 53 00
7e9559e4: 53 6d 69 74 68 00 00 00 7c 03 00 01 00 32 00 3e 00

<4 lemon:~/Examples > cat example25.support.s

.global print_all
.text
.balign 4
print_all:
    push {r4,r5,r6,r7,r8,lr}
    mov r4, r0          @ 1st arg: base address
    mov r5, r1          @ 2nd arg: num records
    ldr r0, =fmt1       @ print column headers
    bl printf
mov r6, #0          @ init index to zero
loop:
    cmp r6, r5          @ comp index, elements
    bge endloop
    lsl r7, r6, #4      @ offset <= index * 16
    add r8, r4, r7      @ addr <= base + offset
    ldr r0, =fmt2       @ print one record
    add r1, r8, #0      @ address of name field
    ldrh r2, [r8, #12]   @ contents exam1 field
    ldrh r3, [r8, #14]   @ contents exam2 field
    bl printf
    add r6, r6, #1      @ increment index by one
b  loop

endloop:
    ldr r0, =fmt3       @ print a blank line
    bl printf
    pop {r4,r5,r6,r7,r8,lr}
    bx lr

fmt1: .ascii "\n"
fmt2: .asciz "%-12s %6d %6d\n"
fmt3: .asciz "\n"
Example #25 recap

Array of 8 elements, each element 16 bytes.

Access to individual elements (one record):

```
lsl    r7, r6, #4   @ offset <= index * 16
add    r8, r4, r7  @ addr <= base + offset

@ r8 contains address of current record
```

Access to fields within one record:

```
ldr    r0, =fmt2   @ print one record
add    r1, r8, #0  @ address of name field
ldrh   r2, [r8, #12] @ contents exam1 field
ldrh   r3, [r8, #14] @ contents exam2 field
bl     printf

@ r8 contains address of current record
```