Lecture Topics

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

Announcements

- Self-study Module #8
- Project #8 (due no later than 11/2)
- Project #9 (due no later than 11/9)
Memory is viewed as a linear sequence of bytes (flat memory model).

Addresses are 32 bits wide, so there are $2^{32}$ bytes (4 Gigabytes).

Addresses range from 00000000 to FFFFFFFF.

Manage as “segments” of 64 Kilobytes:
(00000000 to 0000FFFF, 00010000 to 0001FFFF, 00020000 to 0000FFFF, etc.)
Relationship to C/C++ data objects:

- **program lifetime – data section**
  - global variables
  - static variables

- **block lifetime – stack**
  - local variables
  - parameters

- **programmer-defined lifetime – heap**
  - dynamically allocated memory

Operating system controls access to segments (equivalent of file permissions):

- OS segment – no user access
- text segment – read-only access
- data segment – read-write access
- heap – read-write access
- stack – read-write access

Invalid access: segmentation fault
Bytes Within A Word

- **Little-endian**: byte numbers start at the little (least significant) end
- **Big-endian**: byte numbers start at the big (most significant) end
- **Swift’s Gulliver’s Travels**: the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end
- **It doesn’t really matter** which addressing type used – except when two systems share data

```
.word 0x00010203
.word 0x04050607
.word 0x08090A0B
.word 0x0C0D0E0F
```

// little-endian     // big-endian

```
0000 03020100     0000 00010203
0004 07060504     0004 04050607
0008 0B0A0908     0008 08090A0B
000c 0F0E0D0C     000c 0C0D0E0F
```
Textbook’s convention

Memory locations shown from largest address to smallest address:

<table>
<thead>
<tr>
<th>Big-Endian</th>
<th>Little-Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Byte Address</strong></td>
<td><strong>Word Address</strong></td>
</tr>
<tr>
<td>C D E F</td>
<td>C</td>
</tr>
<tr>
<td>8 9 A B</td>
<td>8</td>
</tr>
<tr>
<td>4 5 6 7</td>
<td>4</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>0</td>
</tr>
</tbody>
</table>

MSB | LSB | MSB | LSB |

Assembler Directives

Some assembler directives are used to control the assembly process (take some action during assembly, as opposed to generating machine language which will take some action during execution).

Some assembler directives are used to reserve memory (and generate initial values).
Switch between sections (starts in .text by default). Force next address to be a multiple of 4 (machine language instructions are 4 bytes, PC used to access RAM).

```
.text
.balign 4
```

Switch between sections. Force next address to be a multiple of 4 (LDR or STR will be used to access total).

```
data
.balign 4
total: .word 0
```

---

**Reserve Space**

Directives for integer-like objects:

- `blank: .byte 0x20`
- `max: .short 500`
- `sum: .word 0`
- `list: .word 5, 10, -6, 25`
Reserve Space

Directives for string objects:

```
str: .ascii "First"
    .asciz "Second"
```

Directive for uninitialized space:

```
room: .skip 100
```

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)
Data Movement Instructions

The target address for a Load or a Store instruction is formed from two parts:

- a base register
- an offset

The offset is one of two things:

- an immediate value (12 bits)
- a register

---

Data Movement Instructions

Format:

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd</td>
<td>0</td>
<td>I</td>
<td>P</td>
<td>U</td>
<td>B</td>
<td>W</td>
<td>L</td>
<td>Rn</td>
<td>Rd</td>
<td>addressing_mode_specific</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **I, P, U, W**: Are bits that distinguish between different types of addressing modes. See Addressing Mode 2 - Load and Store Word or Unsigned Byte on page A5-18
- **L bit**: Distinguishes between a Load (L=1) and a Store instruction (L=0).
- **B bit**: Distinguishes between an unsigned byte (B=1) and a word (B=0) access.
- **Rn**: Specifies the base register used by <addressing_mode>.
- **Rd**: Specifies the register whose contents are to be loaded or stored.
Comparison

Data manipulation:

<table>
<thead>
<tr>
<th>cond</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>shifter_operand</th>
</tr>
</thead>
</table>

Data movement:

<table>
<thead>
<tr>
<th>cond</th>
<th>0</th>
<th>1</th>
<th>I</th>
<th>P</th>
<th>U</th>
<th>B</th>
<th>W</th>
<th>L</th>
<th>Rn</th>
<th>Rd</th>
<th>addressing_mode_specific</th>
</tr>
</thead>
</table>

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
Assuming registers contain values:

\[
\begin{align*}
\text{strb} & \quad r2, [r1,r5] \\
\text{strh} & \quad r10, [r4,#8] \\
\text{str} & \quad r7, [r5,#-4] \\
\text{strd} & \quad r6, [r2]
\end{align*}
\]

Offset missing? Implied zero.

Assume: \( r6: 11223344 \quad r4: 00020400 \)

\[
\text{str} \quad r6, [r4]
\]

Contents of RAM:

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>00020400</td>
<td>88 44</td>
</tr>
<tr>
<td>00020401</td>
<td>99 33</td>
</tr>
<tr>
<td>00020402</td>
<td>AA 22</td>
</tr>
<tr>
<td>00020403</td>
<td>BB 11</td>
</tr>
<tr>
<td>00020404</td>
<td>CC</td>
</tr>
<tr>
<td>00020405</td>
<td>DD</td>
</tr>
<tr>
<td>00020406</td>
<td>EE</td>
</tr>
<tr>
<td>00020407</td>
<td>FF</td>
</tr>
</tbody>
</table>

Contents of \( r6 \) copied into memory at effective address \( r4+0 \)

What about STRB and STRH? Only part of register contents will be copied into memory (1 byte or 2 bytes).
Assume: r6: 11223344  r4: 00020400

**strb r6, [r4]**  **strh r6, [r4]**

**Contents of RAM:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00020400</td>
<td>88 44</td>
</tr>
<tr>
<td>00020401</td>
<td>99 33</td>
</tr>
<tr>
<td>00020402</td>
<td>AA</td>
</tr>
<tr>
<td>00020403</td>
<td>BB</td>
</tr>
<tr>
<td>00020404</td>
<td>CC</td>
</tr>
<tr>
<td>00020405</td>
<td>DD</td>
</tr>
<tr>
<td>00020406</td>
<td>EE</td>
</tr>
<tr>
<td>00020407</td>
<td>FF</td>
</tr>
</tbody>
</table>

Assume: r6: 11223344  r7: 55667788

**strd r6, [r4]**

**Contents of RAM:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00020400</td>
<td>88 44</td>
</tr>
<tr>
<td>00020401</td>
<td>99 33</td>
</tr>
<tr>
<td>00020402</td>
<td>AA 22</td>
</tr>
<tr>
<td>00020403</td>
<td>BB 11</td>
</tr>
<tr>
<td>00020404</td>
<td>CC 88</td>
</tr>
<tr>
<td>00020405</td>
<td>DD 77</td>
</tr>
<tr>
<td>00020406</td>
<td>EE 66</td>
</tr>
<tr>
<td>00020407</td>
<td>FF 55</td>
</tr>
</tbody>
</table>

**STRD**

Contents of r6 copied into memory at effective address r4+0

Contents of r7 copied into memory at r4+4

Register “rd” must be an even numbered register.
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:

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

Offset missing? Implied zero.
Assume: \( r6: 11223344 \) \( r4: 00020400 \)

**Contents of RAM:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00020400</td>
<td>88</td>
</tr>
<tr>
<td>00020401</td>
<td>99</td>
</tr>
<tr>
<td>00020402</td>
<td>AA</td>
</tr>
<tr>
<td>00020403</td>
<td>BB</td>
</tr>
<tr>
<td>00020404</td>
<td>CC</td>
</tr>
<tr>
<td>00020405</td>
<td>DD</td>
</tr>
<tr>
<td>00020406</td>
<td>EE</td>
</tr>
<tr>
<td>00020407</td>
<td>FF</td>
</tr>
</tbody>
</table>

\[ \text{ldr} \ r6, \ [r4] \]
\[ \text{r6: } 00000088 \]

\[ \text{ldrsb} \ r6, \ [r4] \]
\[ \text{r6: } \text{FFFFFF88} \]

\[ \text{ldrh} \ r6, \ [r4] \]
\[ \text{r6: } 00009988 \]

\[ \text{ldrsh} \ r6, \ [r4] \]
\[ \text{r6: } \text{FFFF9988} \]

Assume:
\( r6: 11223344 \) \( r4: 00020400 \)

**Contents of RAM:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00020400</td>
<td>88</td>
</tr>
<tr>
<td>00020401</td>
<td>99</td>
</tr>
<tr>
<td>00020402</td>
<td>AA</td>
</tr>
<tr>
<td>00020403</td>
<td>BB</td>
</tr>
<tr>
<td>00020404</td>
<td>CC</td>
</tr>
<tr>
<td>00020405</td>
<td>DD</td>
</tr>
<tr>
<td>00020406</td>
<td>EE</td>
</tr>
<tr>
<td>00020407</td>
<td>FF</td>
</tr>
</tbody>
</table>

\[ \text{ldr} \ r6, \ [r4] \]
\[ \text{r6: } \text{BBAA9988} \]

\[ \text{ldrd} \ r6, \ [r4] \]
\[ \text{r6: } \text{BBAA9988} \]
\[ \text{r7: } \text{FFEEEDCC} \]

For LDRD, "rd" must be an even numbered register.
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;
```

```c
char A = 'X';
char B = 'Z';
A = 'Q';
B = A;
```

```assembly
A: .byte 'X'
B: .byte 'Z'

A = 'Q';
    mov  r0, #'Q'
    ldr  r1, =A
    strb r0, [r1]

B = A;
    ldr  r0, =A
    ldrb r1, [r0]
    ldr  r2, =B
    strb r1, [r2]
```
int A = 0, B = 0;  
A: .word 0  
B: .word 0

A = 125;  
    mov  r0, #125  
    ldr  r1, =A  
    str  r0, [r1]

B = A;  
    ldr  r0, =A  
    ldr  r1, [r0]  
    ldr  r2, =B  
    str  r1, [r2]

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
Data Structures

More complex data structures can be built out of records and arrays:

- Array of records
- Chain of records ("linked list")
- Tree of records ("binary search tree")

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

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:

```
+0    | label (1 byte)
+4    | padding (3 bytes)
+8    | x_coord (4 bytes)
+8    | y_coord (4 bytes)
```

Example #23

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

Course website:

`~cse320/Examples/example23.pdf`
Note use of “.balign 4” to generate 3 bytes of padding:

```
.data
.balign 4
pointA:
.byte 'A' @ Label
.balign 4
.word 12, 16 @ X and Y coordinates
pointB:
.byte 'B' @ Label
.balign 4
.word 19, 14 @ X and Y coordinates
```

Alternatives to handle padding: .skip 3 or .byte 0,0,0

```
<1 lemon:~/Examples > cat example23.s

.global main
.text
.balign 4
main:
    push {lr}
ldr r4, =pointA @ address of "pointA"
ldr r0, =fmt @ 1st arg: &fmt
ldrb r1, [r4,#0] @ 2nd arg: label
ldr r2, [r4,#4] @ 3rd arg: X coord
ldr r3, [r4,#8] @ 4th arg: Y coord
bl printf @ Display one point
```
ldr  r4, =pointB  @ address of "pointB"
ldr  r0, =fmt    @ 1st arg: &fmt
ldrb r1, [r4,#0] @ 2nd arg: label
ldr  r2, [r4,#4] @ 3rd arg: X coord
ldr  r3, [r4,#8] @ 4th arg: Y coord
bl  printf     @ Display one point

ldr  r0, =pointA @ 1st arg: &pointA
mov  r1, #24    @ 2nd arg: count
bl  display1   @ Display memory

ldr  r0, =main  @ 1st arg: &main
mov  r1, #24    @ 2nd arg: count
bl  display4   @ Display memory

pop   {lr}
bx    lr

fmt:  .asciz "\nPoint %c: ( %d, %d )\n"
      .balign 4

Note use of ".balign 4" after ".asciz" to generate any 
padding needed to get back onto a 4-byte boundary 
(formatting string is a read-only constant in the ".text" 
section).
<2 lemon:~/Examples > gcc example23.s ~/lib/memlib.o

<3 lemon:~/Examples > a.out

Point A: (12, 16)
Point B: (19, 14)

Memory contents from 0002077c to 00020794
0002077c: 41 ← 'A'
0002077d: 00
0002077e: 00
0002077f: 00
00020780: 0c ← 0x0000000C or 12
00020781: 00
00020782: 00
00020783: 00
00020784: 10 ← 0x00000010 or 16
00020785: 00
00020786: 00
00020787: 00
...
Memory contents from 00010420 to 00010480
00010420: e52de004
00010424: e59f406c
00010428: e59f006c
0001042c: e5d41000
00010430: e5942004
00010434: e5943008
00010438: ebffe0f0
0001043c: e59f405c
00010440: e59f0054
00010444: e5d41000
00010448: e5942004
.
.

machine language for push (lr)

Memory contents from 00010480 to 0001049e
00010480: 0a
00010481: 50
00010482: 6f
00010483: 69
00010484: 6e
00010485: 74
00010486: 20
00010487: 25
00010488: 63
00010489: 3a
.
.

ASCII character codes for "\nPoint %c:"