Lecture Topics

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

Announcements

- Self-study Module #8 (subprograms)
- Self-study Module #9 (data organization)
- Project #7 (due no later than 6/13)
Exam #2

- Friday, 6/16 during lecture
- 20% of course grade
- One 8.5x11 note sheet (both sides) allowed
- Study suggestions on course website

ARM Memory Model

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 0002FFFF, 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

Data Movement Instructions

Two forms of data movement:

Load:  RAM $\rightarrow$ Register

Store:  Register $\rightarrow$ RAM

The data movement instructions allow us to copy values between a memory location and a register (and vice versa)
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>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>0</td>
<td>1</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>
</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 45-18
- **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:

Data movement:

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:

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

Offset missing? Implied zero.

Assume: r6: 11223344  r4: 00020400

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

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</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: \( r6: 11223344 \) \( r4: 00020400 \)

\begin{align*}
\text{strb } r6, [r4] & \quad \text{strh } r6, [r4] \\
\text{Contents of RAM:} & \quad \text{Contents of RAM:} \\
00020400: & \ 88 \ 44 \quad 00020400: \ 88 \ 44 \\
00020401: & \ 99 \ 33 \\
00020402: & \ AA \quad 00020402: \ AA \\
00020403: & \ BB \\
00020404: & \ CC \\
00020405: & \ DD \\
00020406: & \ EE \\
00020407: & \ FF \\
\end{align*}

Assume: \( r6: 11223344 \) \( r7: 55667788 \)

\begin{align*}
\text{strd } r6, [r4] & \quad \text{STRD} \\
\text{Contents of RAM:} & \quad \text{Contents of RAM:} \\
00020400: & \ 88 \ 44 \quad 00020400: \ 88 \ 44 \\
00020401: & \ 99 \ 33 \quad 00020401: \ 99 \ 33 \\
00020402: & \ AA \quad 00020402: \ AA \\
00020403: & \ BB \quad 00020403: \ BB \\
00020404: & \ CC \quad 00020404: \ CC \\
00020405: & \ DD \quad 00020405: \ DD \\
00020406: & \ EE \quad 00020406: \ EE \\
00020407: & \ FF \quad 00020407: \ FF \\
\end{align*}

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
ldr 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>
<th>Instruction</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00020400</td>
<td>88</td>
<td>ldrb r6, [r4]</td>
<td>r6:</td>
<td>00000088</td>
</tr>
<tr>
<td>00020401</td>
<td>99</td>
<td>ldrsb r6, [r4]</td>
<td>r6:</td>
<td>FFFFFFF88</td>
</tr>
<tr>
<td>00020402</td>
<td>AA</td>
<td>ldrh r6, [r4]</td>
<td>r6:</td>
<td>FFFFF9988</td>
</tr>
<tr>
<td>00020403</td>
<td>BB</td>
<td>ldrsh r6, [r4]</td>
<td>r6:</td>
<td>FFF9988</td>
</tr>
<tr>
<td>00020404</td>
<td>CC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00020405</td>
<td>DD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00020406</td>
<td>EE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00020407</td>
<td>FF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume: r6: 11223344  r4: 00020400

Contents of RAM:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Instruction</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00020400</td>
<td>88</td>
<td>ldr r6, [r4]</td>
<td>r6:</td>
<td>BBAA9988</td>
</tr>
<tr>
<td>00020401</td>
<td>99</td>
<td>ldrd r6, [r4]</td>
<td>r6:</td>
<td>BBAA9988</td>
</tr>
<tr>
<td>00020402</td>
<td>AA</td>
<td></td>
<td>r7:</td>
<td>FFEEDDCC</td>
</tr>
<tr>
<td>00020403</td>
<td>BB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00020404</td>
<td>CC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00020405</td>
<td>DD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00020406</td>
<td>EE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00020407</td>
<td>FF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

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

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

```
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")

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

```
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)
     ← padding (3 bytes)
+4  ← 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

---

```
31
<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
```

---

32
```
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
```

```
ldr   r0, =fmt      @ 1st arg: &fmt
mov   r1, #30       @ 2nd arg: count
bl    display1     @ 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 ← 0x000000C 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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010420</td>
<td>e52de004</td>
</tr>
<tr>
<td>00010424</td>
<td>e59f406c</td>
</tr>
<tr>
<td>00010428</td>
<td>e59f006c</td>
</tr>
<tr>
<td>0001042c</td>
<td>e5d41000</td>
</tr>
<tr>
<td>00010430</td>
<td>e5942004</td>
</tr>
<tr>
<td>00010434</td>
<td>e5943008</td>
</tr>
<tr>
<td>00010438</td>
<td>ebfffffa2</td>
</tr>
<tr>
<td>0001043c</td>
<td>e59f405c</td>
</tr>
<tr>
<td>00010440</td>
<td>e59f0054</td>
</tr>
<tr>
<td>00010444</td>
<td>e5d41000</td>
</tr>
<tr>
<td>00010448</td>
<td>e5942004</td>
</tr>
</tbody>
</table>

Machine language for `push (lr)`

### Memory contents from 00010480 to 0001049e

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010480</td>
<td>0a</td>
</tr>
<tr>
<td>00010481</td>
<td>50</td>
</tr>
<tr>
<td>00010482</td>
<td>6f</td>
</tr>
<tr>
<td>00010483</td>
<td>69</td>
</tr>
<tr>
<td>00010484</td>
<td>6e</td>
</tr>
<tr>
<td>00010485</td>
<td>74</td>
</tr>
<tr>
<td>00010486</td>
<td>20</td>
</tr>
<tr>
<td>00010487</td>
<td>25</td>
</tr>
<tr>
<td>00010488</td>
<td>63</td>
</tr>
<tr>
<td>00010489</td>
<td>3a</td>
</tr>
</tbody>
</table>

ASCII character codes for "\nPoint %c:"
Arrays

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

C/C++ declarations:

\[
\begin{align*}
\text{int } & \ A[100]; \\
\text{char } & \ B[20]; \\
\text{unsigned long long } & \ C[50]; \\
\end{align*}
\]

The compiler uses an **array mapping function** to handle array subscripting.

Assume:

\[
\begin{align*}
\text{int } & \ I, \ A[100]; \\
\end{align*}
\]

\[\ldots \ A[I] \ \ldots\]

The compiler calculates the address of \(A[I]\):

\[
\text{address of } A \ + \ (I \ \times \ \text{size of one item})
\]

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:

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

Generates six 2-byte values:

```
0015 ffd3 0060 0048 ffd0 0043
```

<1 lemon:~/Examples > cat example24.s

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"
<2 lemon:~/Examples > gcc example24.s \n/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:

```
ls1    r6, r5, #1   @ r6:  index*2
ldrsh  r7, [r8,r6]  @ r7:  array element
```

Alternative:

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