CSE 30: Computer Organization and Systems Programming

Lecture 11: Memory map of C programs
ARM Programmer’s Model

Diba Mirza
University of California, San Diego
Typical ARM Memory Map

- **Text**: Machine code (read-only)
- **Data**: Initialized static and global data
- **BSS**: Uninitialized or zero-initialized global and static data
- **Heap**: Data is created at run time in this section of memory
- **Stack**: Interface to inputs/outputs
- **OS and Memory-Mapped IO**: Interface to inputs/outputs

Memory addresses:
- 0x00000000
- 0xFFFFFFFFFC
# Program Memory Map

- **"Text"** (instructions in machine language)
- **"Data"** contains any global or static variables which have a pre-defined value and can be modified. That is any variables that are not defined within a function (and thus can be accessed from anywhere) or are defined in a function but are defined as *static* so they retain their value across subsequent calls.
- **"BSS"** also known as *uninitialized data*, is usually adjacent to the data segment. The BSS segment contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.
- **"Heap"** (for dynamically allocated data)
- **"Stack"** (for function local variables)

Heap and stack change in size as the program executes.
Viewing the memory map with ‘size’

- You can use the size command to check the memory map of your executable

```c
mem.c:
void foo() {
}
```

```
gcc -c mem.c

size mem.o
```

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>14 hello.o</td>
</tr>
</tbody>
</table>
Checking the memory map

```c
void foo() {
    int i=10; //line 1
}
```

Output of size before adding line 1

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>20 32 14 hello.o</td>
</tr>
</tbody>
</table>

The size of which section of the memory map of the program will increase on adding line 1?

A. Text
B. Data
C. BSS
D. None of the above
E. All of the above
void foo() {
    static int i=10;  //line 1
}

Output of size before adding line 1

The size of which section of the memory map of the program will increase on adding line 1?

A. Text
B. Data
C. BSS
D. None of the above
E. All of the above
Checking the memory map

int i=0;  //line 1
void foo() {
}

Output of size before adding line 1

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>14</td>
<td>hello.o</td>
</tr>
</tbody>
</table>

The size of which section of the memory map of the program will increase on adding line 1?

A. Text
B. Data
C. BSS
D. None of the above
E. All of the above
Translations

- High-level language program (in C)
  ```c
  swap (int v[], int k)
  {
      int temp;
      temp = v[k];
      v[k] = v[k+1];
      v[k+1] = temp;
  }
  ```

- Assembly language program (for MIPS)
  ```assembly
  swap: sll $2, $5, 2
  add $2, $4, $2
  lw $15, 0($2)
  lw $16, 4($2)
  sw $16, 0($2)
  sw $15, 4($2)
  jr $31
  ```

- Machine (object, binary) code (for MIPS)
  ```bin
  000000 00000 00101 0001000010000000
  000000 00100 00010 0001000000100000
  ... 4 bytes
  ```
## Steps in program execution

<table>
<thead>
<tr>
<th>Address</th>
<th>Operation</th>
<th>Registers</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4000</td>
<td>mov r2, r5</td>
<td>r2, r5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x4004</td>
<td>add r4, r2, #1</td>
<td>r4, r2, #1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x4008</td>
<td>ldr r10, [r6, r2]</td>
<td>r10, [r6, r2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x400c</td>
<td>ldr r11, [r6, r4]</td>
<td>r11, [r6, r4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x4010</td>
<td>mov r1, r10</td>
<td>r1, r10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x4014</td>
<td>str r11, [r6, r2]</td>
<td>r11, [r6, r2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x4018</td>
<td>str r1, [r6, r4]</td>
<td>r1, [r6, r4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x401c</td>
<td>bx lr</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Program in memory is executed following a cycle of Fetch - Decode - Execute.
The ARM Register Set

- Registers: (Very) Small amount of memory inside the CPU
- Each ARM register is 32 bits wide
- 30 general purpose registers
  - 6 status registers
  - 1 program counter

General Purpose Registers
- No data type
- Few ➞ fast
- Refer as r0 - r12

Special Purpose Registers
- Stack pointer (points to the top of the stack)
- Program Counter (points to the next instruction that should be executed)
ARM Assembly Variables: Registers

- Unlike HLL like C or Java, assembly cannot use variables
  - Why not? Keep Hardware Simple
- Data is put into a register before it is used for arithmetic, tested, etc.
- Result is stored in a register (later stored to memory)
- Benefit: Since registers are directly in hardware, they are very fast
- In C (and most High Level Languages) variables declared first and given a type
  - Example:
    ```
    int fahr, celsius;
    char a, b, c, d, e;
    ```
- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match `int` and `char` variables)
- In Assembly Language, the registers have no type; operation determines how register contents are treated
.syntax unified
.text
.align 8
.global get_min_ARM
.func get_min_ARM, get_min_ARM
.type get_min_ARM, %function

get_min_ARM:    (label)
    @ Save caller's registers on the stack
    push {r4-r11, ip, lr}

    @ YOUR CODE GOES HERE (list *ls is in r0)
    @-----------------------------
    @ (your code)
    @ put your return value in r0 here:
    @-----------------------------
    @ restore caller's registers
    pop {r4-r11, ip, lr}
    @ ARM equivalent of return
    BX lr
.endfunc
.end
Basic Types of Instructions

1. Arithmetic: Only involves processor and registers
   - compute the sum (or difference) of two registers, store the result in a register
   - move the contents of one register to another

2. Memory Instructions: Transfer of data between registers and memory
   - load a word from memory into a register
   - store the contents of a register into a memory word

3. Control Transfer Instructions: Change flow of execution
   - jump to another instruction
   - conditional jump (e.g., branch if register == 0)
   - jump to a subroutine
Arithmetic Instructions

In C:
\[
a = b + c;
\]

In ARM:
\[
ADD r0, r1, r2
\]
\[
\text{operands 1 & 2 must be registers}
\]
\[
\text{we can specify operand 3 in different ways}
\]
\[
\text{operand 1 is always the destination register}
\]
Specifying constants in Arithmetic Instructions

In C:

\[ a = b + 10; \]

In ARM:

\[ \text{ADD } r0, r1, \#10 \]

- Immediates are numerical constants.
- They appear often in code, so there are ways to indicate their existence.
How big can immediates be?

Q: What is a plausible range for the immediate in the instruction ADD r0, r1, <immediate>?

A. 0 to \((2^{32}-1)\)

B. 0 to 255

Since the entire instruction needs to be encoded using 32 bits or 4 bytes, we cannot use all of the bits to specify the value of the immediate. We need to specify bits to encode the opcodes, and the first two operands. So 0-255 seems like a more plausible range (1 byte for the value of the immediate).