2008 S2 COMP1921
2. Memory & Execution

Oliver Diessel
odiessel@cse.unsw.edu.au

Last updated: 12:30 7 Aug 2008
Memory & Execution

- Topics to be covered:
  - Memory representation of program data
  - Execution of C idiom on a processor

- Approach:
  - MIPS processor, memory model & assembly language used as representative example
  - Expect you to understand representation of data and how common constructs are executed
  - Do not expect you to be able to translate C program fragments into assembly language
  - MIPS reference provided for students who wish to understand assembly language in more detail
Assembly Code

MIPS assembly code consists of:

- instructions (e.g. `add $t0,$t1,$t2`)
- directives (e.g. `.asciiz "a string"`)
- labels (e.g. `main:`, `endloop:`)
- comments (anything after a `#`)
Assembly Code Example

Example – “Hello, World”:

```
.text
main:
    la $a0, hello      # get addr of string
    li $v0, 4          # set up call to print_string
    syscall           # execute call
    jr $ra             # return 0;

.data
hello:
    .asciiz "Hello, World!\n"
    .word 1,2,3,4,5    # example of “const int array”
vec:
    .space 20          # “char” array example
```
MIPS Memory Map
- where are programs & data stored?

Locally declared variables and function parameters

Data dynamically allocated by your program (see later)

Your program’s global variables are stored here
MIPS Registers

- Simple MIPS processors have 32 x 32-bit registers
  - Conventions for using them have evolved

<table>
<thead>
<tr>
<th>Names</th>
<th>#’s</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>0</td>
<td>Constant 0</td>
</tr>
<tr>
<td>at</td>
<td>1</td>
<td>Reserved for assembler</td>
</tr>
<tr>
<td>v0, v1</td>
<td>2,3</td>
<td>Expression evaluation and results of a function</td>
</tr>
<tr>
<td>a0..a3</td>
<td>4..7</td>
<td>Arguments 1-4</td>
</tr>
<tr>
<td>t0..t7</td>
<td>8..16</td>
<td>Temporary (not preserved across function calls)</td>
</tr>
<tr>
<td>s0..s7</td>
<td>16..23</td>
<td>Saved temporary (preserved across function calls)</td>
</tr>
<tr>
<td>t8, t9</td>
<td>24,25</td>
<td>Temporary (not preserved across function calls)</td>
</tr>
<tr>
<td>k0, k1</td>
<td>26,27</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>gp</td>
<td>28</td>
<td>Pointer to global area</td>
</tr>
<tr>
<td>sp</td>
<td>29</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>fp</td>
<td>30</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>ra</td>
<td>31</td>
<td>Return address (used by function calls)</td>
</tr>
</tbody>
</table>
Constants

- Simple constants can be immediate operands. For example:
  - `int i; ... i = 255; ⇒ li $t0, 255`
  - `char ch; ... ch = 'a'; ⇒ li $t1, 'a'`

- Strings need to be stored in memory and accessed by address or name. For example:
  - `char *c; ... c = "xyzzy"; ⇒`

```
.data
string:
  .asciiz "xyzzy"
.text
...
lw $s0, string
```

"Load Immediate"

"Load Address"
Variables

- Simple variables: e.g. int, char, float and pointers are placed in:
  - a register (if local and heavily used)
  - the stack (if local and less-heavily used)
  - the data segment (if global or static)

- Structured variables: e.g. arrays and structs (see soon) are placed in
  - the stack (if local)
  - the data segment (if global or static)

- All dynamically allocated objects (a little later, but next couple of weeks) are placed in the heap
Variable Allocation

- For the following program:
  ```
  char buffer[100]; int nchars;

  Bool find(char ch, char str[]) {
    int i; Bool found;

    found = FALSE;
    for (i = 0; str[i] != '\0'; ++i) {
      if (str[i] == ch) found = TRUE;
    }
    return found;
  }
  ```

- Typical allocations:
  - `buffer` and `nchars` in the data segment
  - `ch`, `str`, `i` and `found` on the stack and/or in registers
**Memory allocation**

- Depends on the type of data and the processor:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Number of bytes typically allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>int, float</td>
<td>4</td>
</tr>
<tr>
<td>double, long</td>
<td>8</td>
</tr>
<tr>
<td>Pointer e.g. int *, char *</td>
<td>4</td>
</tr>
<tr>
<td>Array e.g. int a[10], char s[20]</td>
<td>Num elements * base data type</td>
</tr>
<tr>
<td>String constant</td>
<td>Num chars + 1</td>
</tr>
<tr>
<td>Other constants</td>
<td>Size of data type</td>
</tr>
</tbody>
</table>
Quick Quiz (Assumed Knowledge)

- How are variables of type `char` represented?
  - What is the name of the code?
  - What is the value of ‘A’?

- How are variables of type `int` represented?
  - How do you represent any positive integer value as a binary number?
  - Can you represent it as a hexadecimal number?
  - Do you know how to add two binary numbers together?
Extending your knowledge 1

- C has three integer data types: `short`, `int`, `long`
  - These vary in the number of bits (typically 16, 32 & 64 bits respectively) used to represent them and thus the range of numbers they can store
  - These data types are `signed` i.e. can represent positive & negative integers and thus divide the range of numbers they can represent between each (0 treated as 1st +ve)
- C also provides the `unsigned` type which is 32 bits large and only stores positive numbers
  - The `int` value `0xFFFFFFFF` represents -1
  - The `unsigned` value `0xFFFFFFFF` represents $2^{32}-1$
### IEEE Floating Point Type Representations

- **Extending your knowledge**

#### IEEE 754 Single precision floating point format (float)

<table>
<thead>
<tr>
<th>S</th>
<th>exponent</th>
<th>mantissa/significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8 bits</td>
<td>23 bits</td>
</tr>
</tbody>
</table>

#### IEEE 754 Double precision floating point format (double)

<table>
<thead>
<tr>
<th>S</th>
<th>exponent</th>
<th>mantissa/significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11 bits</td>
<td>20 bits</td>
</tr>
</tbody>
</table>

**mantissa/significand**

Word 2: 32 bits
Floating point representation
- Extending your knowledge 3

\[(\text{sign}) ± 2^{\text{mantissa}} \times 2^{\text{exponent}}]\]

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Error</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>(1,23,8)</td>
<td>(\approx 1/10^7)</td>
<td>(\approx 10^{\pm 38})</td>
</tr>
<tr>
<td>double</td>
<td>(1,52,11)</td>
<td>(\approx 1/10^{16})</td>
<td>(\approx 10^{\pm 308})</td>
</tr>
</tbody>
</table>
Addressing Objects: Endianess

- **Big Endian:**
  - word address corresponds to most significant byte
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

- **Little Endian:**
  - word address corresponds to least significant byte
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

![Endian byte addresses diagram]

*Big endian byte:* 0 1 2 3

*Little endian byte addresses:* 3 2 1 0
Endianness Example

- Assume the integer with hexadecimal value 0x01020408 is stored at address 0x4000 in memory.
- Depending upon endianness, the following representations in memory can be derived:

<table>
<thead>
<tr>
<th>Big Endian byte addresses</th>
<th>Little Endian byte addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4000</td>
<td>0x4000</td>
</tr>
<tr>
<td>0x4001</td>
<td>0x4004</td>
</tr>
<tr>
<td>0x4002</td>
<td>0x08</td>
</tr>
<tr>
<td>0x4003</td>
<td>0x04</td>
</tr>
<tr>
<td>0x4004</td>
<td>0x02</td>
</tr>
<tr>
<td>0x4005</td>
<td>0x01</td>
</tr>
</tbody>
</table>

- 0x01020408 = 1 \times 2^{24} = 16,777,216 + 
  2 \times 2^{16} = 131,072 + 
  4 \times 2^8 = 1,024 + 
  8 \times 2^0 = 8 

  \[ 16,909,320 \]
Another Endianess Example

- Assume the string “Mary had” is stored at address 0x4000 in memory.
- Depending upon endianess, the following representations in memory can be derived:

<table>
<thead>
<tr>
<th>Big Endian byte addresses</th>
<th>Little Endian byte addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4000 ‘M’ ‘a’ ‘r’ ‘y’</td>
<td>0x4000 ‘y’ ‘r’ ‘a’ ‘M’</td>
</tr>
<tr>
<td>0x4004 ‘ ’ ‘h’ ‘a’ ‘d’</td>
<td>0x4004 ‘d’ ‘a’ ‘h’ ‘ ’</td>
</tr>
</tbody>
</table>
Addressing Objects: Alignment

- The address of an object is required to be aligned with an address that is a multiple of its size
  - Thus, a `char` can be placed at any address since it is only one byte large
  - But an `int` must be located at an address that is a multiple of 4. Thus 0x0, 0x4 and 0x4000 are valid `int` addresses, but 0x2, 0x3 and 0x42 aren’t

```
Addressing Objects: Alignment

<table>
<thead>
<tr>
<th>Byte offsets</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Aligned</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Not Aligned</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Integer address alignment with byte address
Assignment

- Use data movement operation appropriate for location of object. For example, $i = 5$

  - If $i$ is in a register (e.g. $s0$):
    ```
    li $s0, 5
    ```

  - If $i$ is in data segment (e.g. address 0x1020):
    ```
    li $t0, 5
    sw $t0, 0x1020
    ```

  - If $i$ is on the stack (e.g. offset 20 from $sp$):
    ```
    li $t0, 5
    sw $t0, 20($sp)
    ```

  Relative addressing or base+index addressing

“Store Word”
Boolean Expressions

- Could evaluate using “set/test” operators:

  e.g. \((x > 5 \ \&\& \ y+2 < x \ |\| \ x*x == 9)\)
  \[\Rightarrow \ (\text{assuming } s_1 = x, \ s_2 = y)\]

  - `sgt $t1, $s1, 5`  \# \(t_1 = x > 5\); “set greater than”
  - `add $t0, $s2, 2`  \# \(t_0 = y+2\).
  - `slt $t0, $t0, $s1`  \# \(t_0 = y+2 < x\); “set less than”
  - `and $t1, $t0, $t1`  \# \(t_1 = t_0 \ \&\& \ t_1\),
  - `mul $t2, $s1, $s1`  \# \(t_2 = x*x\); “multiply”
  - `seq $t0, $t2, 9`  \# \(t_0 = t_2 == 9\); “set equal”
  - `or $v0, $t0, $t1`  \# \(v_0 = t_0 \ |\| \ t_1\)

- Note: C actually uses “short-circuit” evaluation. (Evaluates just enough of expression to determine result)
if Statements

- Translation scheme to make C more MIPS-like (assembly languages don’t have structured code constructs):

```plaintext
if (cond1)
    stats1
else if (cond2)
    stats2 =>
else if (cond3)
    stats3
::: :::
else
    statsn
endif:

L1: if (!cond1) goto L2;
    stats1 ; goto endif;
L2: if (!cond2) goto L3;
    stats2 ; goto endif;
L3: if (!cond3) goto L4;
    stats3 ; goto endif;
::: :::
Ln:
    statsn
endif:
```
if Statements

Then in MIPS assembly code, this becomes:

```
L1:

code for cond₁ → $v0
beq $v0, $0, L2

code for stats₁
b endif

L2: ...
```

"Branch Equal"

"Branch"
while Loops

Translation scheme to make C more MIPS-like:

while (cond)

stats ⇒ stats1 ; goto L1;
:::
::: L2: :::

Then the MIPS version becomes:

L1:

*code for cond* → $v0
beq $v0, $0, L2
*code for stats*
b L1

L2: ...
while Loops

- Convert for loops to while loops:

```
for (init; cond; next)  while (cond) {
    stats  ⇒  stats;
    ::::  next;
    }
```
Pointers

- Use `la` (load address) to get addresses of memory objects
  
e.g. \( \text{ip} = \&i \Rightarrow \text{la } \$s0, i \)

  Note: can’t get addresses of register objects.

- Use `(reg)` – register indirect - addressing mode to dereference
  
e.g. \( *\text{ip} = 5 \Rightarrow \)
  
  \[
  \begin{align*}
  \text{li } \$t0, 5 & \quad \# \ t0 = 5; \\
  \text{sw } \$t0, (\$s0) & \quad \# \ *\text{ip} = t0;
  \end{align*}
  \]

- Invalid addresses cause a system “trap” (run-time error) handler to be invoked. For example:
  
  \[
  \begin{align*}
  \text{li } \$t0, 5 & \quad \# \ t0 = 5; \\
  \text{sw } \$t0, (\$t0) & \quad \# \ *t0 = t0;
  \end{align*}
  \]
**Arrays**

- An array declared as `type array[count];` is allocated `count \times \text{sizeof}(type)` bytes. For example:

<table>
<thead>
<tr>
<th>C declarations:</th>
<th>int a[6];</th>
<th>char s[10];</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS directives:</td>
<td>a: .space 24</td>
<td>s: .space 10</td>
</tr>
<tr>
<td>Memory maps:</td>
<td>a</td>
<td>s</td>
</tr>
<tr>
<td></td>
<td>a[0] 0x1000</td>
<td>s[0] 0x2000</td>
</tr>
<tr>
<td></td>
<td>a[1] 0x1004</td>
<td>s[1] 0x2004</td>
</tr>
<tr>
<td></td>
<td>a[2] 0x1008</td>
<td>s[2] 0x2004</td>
</tr>
<tr>
<td></td>
<td>a[3] 0x100C</td>
<td>s[3] 0x2004</td>
</tr>
<tr>
<td></td>
<td>a[4] 0x1010</td>
<td>s[4] 0x2004</td>
</tr>
<tr>
<td></td>
<td>a[5] 0x1014</td>
<td>s[5] 0x2004</td>
</tr>
<tr>
<td></td>
<td>a[6] 0x1018</td>
<td>s[6] 0x2004</td>
</tr>
<tr>
<td></td>
<td>a[7] 0x101C</td>
<td>s[7] 0x2004</td>
</tr>
<tr>
<td></td>
<td>a[8] 0x1020</td>
<td>s[8] 0x2004</td>
</tr>
<tr>
<td></td>
<td>a[9] 0x1024</td>
<td>s[9] 0x2004</td>
</tr>
<tr>
<td></td>
<td>a[10] 0x1028</td>
<td></td>
</tr>
</tbody>
</table>
Arrays

To access array element a[i], compute address as:

\[ &a[0] + i \times \text{sizeof}(a[0]) \]

\[
\text{for } (\text{sum} = i = 0; i < 6; ++i) \\
\quad \text{sum} += a[i]; \quad \Rightarrow (\text{assuming } s0=\text{sum}, s1=i)
\]

\[
\begin{align*}
\text{move } s0, 0 & \quad \# \text{ sum } = 0; \\
\text{move } s1, 0 & \quad \# i = 0; \\
\text{la } t0, a & \quad \# t0 = \&a[0]; \\
\text{loop:} & \quad \# \text{ while } (i < 6) \\
\text{bge } s1, 6, \text{ done} & \quad \# \{ \text{Branch Greater than or Equal to}\} \\
\text{mul } t1, s1, 4 & \quad \# \text{ offset } = i*4 \\
\text{add } t1, t0, t1 & \quad \# \text{ addr } = \&a[0]+\text{offset} \\
\text{lw } t2, (t1) & \quad \# t2 = a[i]; \quad \text{“Load Word”} \\
\text{add } s0, s0, t2 & \quad \# \text{ sum } += t2; \\
\text{add } s1, s1, 1 & \quad \# i = i + 1; \\
\text{j loop} & \quad \# \} \quad \text{“Jump”} \\
\text{done:}
\end{align*}
\]
Arrays and Pointers

- Using pointers to access arrays is more efficient than using an index (less address computation)

```c
int a[6], *p;
for (sum = 0, p = a; p < &a[6]; ++p)
    sum += *p;
⇒ (assuming $s0=sum, $s1=p)

move $s0, $0          # sum = 0;
la $s1, a             # p = a;
la $t0, a+24          # t0 = &a[6];
loop:                 # while (a < t0)
bge $s1, $t0, done   # {
    lw $t1, ($s1)     #   t1 = *p;
    add $s0, $s0, $t1 #   sum += t1;
    add $s1, $s1, 4   #   ++p;
    j loop            # }
done:
```
Where to from here?

- Back to learning C….
  - `structs` (Ch. 8)
  - Dynamic memory allocation (Ch. 10)