CS 240, Episode 2: Memory + MIPS

- Poll by email later today to set office hours.
- Reminder: PS1 due in one week.
- Today:
  - Review bytes, binary, hex
  - Words
  - Memory organization
  - Intro to MIPS architecture and assembly language

Modern Digital Computer

How are data and instructions represented?
How does a program find its data in memory?

Review

byte = 8 bits

- Conventional smallest unit of data
- Binary 00000000₂ -- 11111111₂
- Decimal 00₀₁₀ -- 25₅₁₀
- Hexadecimal 0₀₁₆ -- FF₁₆
  - Base 16
  - Practice
- Notation:
  - Binary: 0b1011
  - Decimal: 11
  - Hex: 0xB

Modern Digital Computer

Processor

Instructions

Memory

data

How are data and instructions represented?
How does a program find its data in memory?

Review

byte = 8 bits

- Conventional smallest unit of data
- Binary 00000000₂ -- 11111111₂
- Decimal 00₀₁₀ -- 25₅₁₀
- Hexadecimal 0₀₁₆ -- FF₁₆
  - Base 16
  - Practice
- Notation:
  - Binary: 0b1011
  - Decimal: 11
  - Hex: 0xB

word

- Normal/largest unit of data in machine
- ISA-dependent size -- MIPS: 4 bytes (32 bits)
  - ISA = Instruction Set Architecture
- word size = register size = address size

Java/C int = 4 bytes: 11,501,584
machine code

The binary representation of a computer’s instruction set is known as its machine code.

On MIPS, 1 machine code instruction = 1 word

What to do (operation code)

What to do it with*

*Well, more or less.

In hex

The MIPS processor

- Register = word-sized storage
- 32 registers
  - Named, like local variables.

- [Invisible:] Hardware to implement operations using data in registers.
### MIPS registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>The constant value 0</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>values for results and expression evaluation</td>
</tr>
<tr>
<td>$a0-$a7</td>
<td>4-7</td>
<td>arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</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</td>
</tr>
</tbody>
</table>

### Modern Digital Computer

- **Processor**
- **Memory**
- **Instructions**
- **Data**

How does a program find its data in memory?

### Byte-Addressable Memory

- Memory = array of bytes (*locations*), unique *address* = index.
- Read/Write
- Programs refer to bytes in memory by their *addresses*.
- Address = word
- Address space size?

### Words in Memory

- Address of word = address of 1st byte in word
- **Alignment**
  - Data of size $n$ bytes stored at $a$ only if $a \mod n = 0$
  - $n$ is a power of 2
  - Required (MIPS) or recommended (x86), depending on platform.
- Why?
MIPS memory organization

- Up to $2^{32}$ bytes = $2^{30}$ words
- Word-aligned

How are the bytes of a word ordered in memory?

<table>
<thead>
<tr>
<th>MIPS Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{31}$-$4$</td>
</tr>
<tr>
<td>$A+24$</td>
</tr>
<tr>
<td>$A+16$</td>
</tr>
<tr>
<td>$A+8$</td>
</tr>
<tr>
<td>$A+4$</td>
</tr>
<tr>
<td>$A$</td>
</tr>
<tr>
<td>$12$</td>
</tr>
<tr>
<td>$8$</td>
</tr>
<tr>
<td>$4$</td>
</tr>
<tr>
<td>$0$</td>
</tr>
</tbody>
</table>

Back to decimal notation

Endianness:

byte order within memory words

Little End

Big End

x86

Little-endian memory layout.

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>2A</td>
</tr>
<tr>
<td>02</td>
<td>8B</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>08</td>
</tr>
</tbody>
</table>

Little end first:

Least significant byte at lowest address.
Most significant byte at highest address.
Position increases as address increases.

Big End first:

Least significant byte at highest address.
Most significant byte at lowest address.
Position decreases as address increases.

MIPS

Big-endian memory layout.

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>0B</td>
</tr>
<tr>
<td>02</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>86</td>
</tr>
<tr>
<td>00</td>
<td>2A</td>
</tr>
</tbody>
</table>

Big-endian memory layout.
The MIPS machine

MIPS Memory

MIPS private regs
PC
IR

MIPS user regs
$zero
$v0
$s0
$s1
$s2
$s3
$s4
$s5

I/O devices

Introductions

Hello, world!
MIPS assembly language

Let's write a program in assembly.

Assembler directives

# test
.globl main
main:

# comment

.data

# program data

# program instructions

# prints (you guessed it) “Hello world!” to screen
# Our first assembly language program
# prints (you guessed it) “Hello world!” to screen
# Our first assembly language program
# prints (you guessed it) “Hello world!” to screen
# prints (you guessed it) “Hello world!” to screen
# prints (you guessed it) “Hello world!” to screen
# prints (you guessed it) “Hello world!” to screen
# prints (you guessed it) “Hello world!” to screen
# prints (you guessed it) “Hello world!” to screen
# prints (you guessed it) “Hello world!” to screen
# prints (you guessed it) “Hello world!” to screen
# prints (you guessed it) “Hello world!” to screen
# prints (you guessed it) “Hello world!” to screen
# prints (you guessed it) “Hello world!” to screen
# prints (you guessed it) “Hello world!” to screen
# prints (you guessed it) “Hello world!” to screen
First reserve room for our message

```
# Our first assembly language program
# prints (you guessed it) "Hello world!" to screen

.globl main
main:
```

Another assembler directive

data . asciiz "Hello world!"

syscall does all the work (why?)

```
li $v0, 4
la $a0, str
```

Okay, OS do your thing
Finally we halt

```
helloWorld.asm
#
# Our first assembly language program
# prints (you guessed it) "Hello world!" to screen
# ############################################################################
.text
# program instructions
.globl main
main:
    li $v0, 4    # sys call code print_str
    la $a0, str # addr of string to print
    syscall    # print the prompt
    li $v0, 10   # sys code halt
    syscall     # halt

.data
# program data
str: .asciiz "Hello world!"
```

Our second assembly program

```
addTwoNumbers.asm
#
# This program computes the sum of two numbers
# X and Y that are input during program execution.
# ############################################################################
# program instructions
# read 1st num into location X
# read 2nd num into location Y
# add Y to X and store in AC
# put the sum into location SUM
# write SUM to screen
# and stop
# program variables
```

Assembling and running our program*

```
addTwoNumbers.asm
#
# This program computes the sum of two numbers
# X and Y that are input during program execution.
# ############################################################################
# program instructions
# read 1st num into location X
# read 2nd num into location Y
# add Y to X and store in AC
# put the sum into location SUM
# write SUM to screen
# and stop
# program variables
```

*More in lab.
Where should X, Y and SUM go?

```assembly
.globl main
main:
    # input X
    # input Y
    # add X to Y and
    # store SUM
    # output SUM
    # halt

.globl data
.data
```

In registers ...

```assembly
.globl main
main:
    # input X
    # input Y
    # add X to Y and
    # store SUM
    # output SUM
    # halt

.globl data
.data
```

How do we input values to X and Y?

```assembly
.globl main
main:
    # input X
    # input Y
    # add X to Y and
    # store SUM
    # output SUM
    # halt

.globl data
.data
```

First reserve room for prompt

```assembly
.globl main
main:
    # input X
    # input Y
    # add X to Y and
    # store SUM
    # output SUM
    # halt

.globl data
.data
```

Another assembler directive
### Reading in an integer

```assembly
# program data

# program instructions

.globl main
main:
  li $v0, 4  # sys call code {print_str}
  la $a0, str  # addr of string to print
  syscall  # print the prompt
    # input x; input Y
  beqz Sys your thing
    # add X to Y and
    # store SUM
    # output SUM
    # halt
.data  # program data
str: .asciiz "Enter number: 
```

### addTwoNumbers.asm

```assembly
li $v0, 4  # sys call code {print_str}
la $a0, str  # addr of string to print
syscall  # print the prompt
    # input x; input Y
```
Reading in X

```
.globl main
main:
    li $v0, 4  # sys call code print_str
    syscall  # print the prompt
    li $v0, 5  # sys call code read integer
    syscall  # read integer
    add $t0, $v0, $s0  # sum X and Y
   
.data
str: .asciiz "Enter number: "
```

The user enters 3

```
move $t0, $v0  # store SUM
```

```bash
The user enters 3 and it miraculously appears in register $v0
```

```
move $t0, $v0  # copy contents of $v0 to $t0
```
Second verse, same as the first*

```
.text
.globl main
main:
  li $v0, 4          # syscall code print_str
  la $a0, str       # addr of string to print
  syscall           # print the prompt
  li $v0, 5          # syscall code read integer
  syscall           # add X to Y & store SUM
  move $t1, $v0     # $t1 is the variable Y
  move $t1, $v0     # $t1 is the variable Y
  move $t0, $v0     # $t0 is the variable X
  move $t0, $v0     # $t0 is the variable X
  move $t0, $v0     # $t0 is the variable X
  li $v0, 5          # syscall code read integer
  syscall           # output SUM
  # halt
.data             # program data
str: .asciiz "Enter number:  

*We should print another prompt here, but PowerPoint space is tight.
```

Adding is easy

```
.text
.globl main
main:
  li $v0, 4          # syscall code print_str
  la $a0, str       # addr of string to print
  syscall           # print the prompt
  li $v0, 5          # syscall code read integer
  syscall           # output SUM
  # halt
.data             # program data
str: .asciiz "Enter number:  
```

MIPS Memory

```
MIPS Memory
(A-4) 0000
(A-3) 0000
(A-2) 0000
(A-1) 0000
(A) 0000
(B) 0000
(C) 0000
(D) 0000
```

Now all we need to do is print ...

```
main:
  li $v0, 4          # syscall code print_str
  la $a0, str       # addr of string to print
  syscall           # print the prompt
  li $v0, 5          # syscall code read integer
  syscall           # print integer
  syscall           # output SUM
  # halt
```

li  $v0, 1
move $a0, $t2
syscall

MIPS Memory

... and halt

main:
li  $v0, 4 # sys code print_str
la $a0, str
syscall # print the prompt
li  $v0, 5 # sys code read integer
syscall
move  $t0, $v0 # $t0 is the variable X
li  $v0, 5 # sys code read integer
syscall
move  $t1, $v0 # $t1 is the variable Y
add$t2, $t0, $t1 # SUM <- X + Y
li  $v0, 1 # sys code print integer
syscall # now print it
li  $v0, 10 # sys code halt
syscall # that's all, folks.