Inf2C - Computer Systems
Lectures 3-4
Assembly Language

Boris Grot

School of Informatics
University of Edinburgh
Announcements: Labs

- **Purpose:**
  - prep for coursework by getting familiar with the tools
  - reinforce class concepts

- **Regular schedule (from this week):**
  - Mon: 2-3pm, 5-6pm in FH 3.D02
  - Wed: 3-4pm in FH 3.D01
  - Thr: 11am-12pm in FH 1.B32

- **Extras (whenever coursework is out):**
  - Tue: 11am-12pm in FH 3.D01
  - Wed: 2-3pm in FH 3.D01
  - Fri: 1-2pm in FH 3.D01

- **Drop-in format**
  - You may attend any & all lab sessions
  - Demonstrator (Amna, Margus, Oscar) on-hand to answer questions
Announcements: Other

- Tutorial 1 running this week
  - Attempt the problems ahead of time
  - Come in with specific questions
  - You’ll get out what you put in

- Online discussion forum: Ask Piazza
Previous lecture

- Representing numbers
  - Binary encoding
  - Negative numbers
  - Floating point numbers
  - Characters & strings

- Other things
  - Binary addition
  - Shifting
  - Hex
Lectures 3-4: MIPS instructions

- Motivation: Learn how a processor’s ‘native’ language looks like
- We will examine the MIPS ISA
  - ISA: Instruction Set Architecture – the language of the computer
  - MIPS: a real-world ISA used by many different processors since the 80s.
    - Regular and representative → great for learning!
- ISA reference online:
Outline

- Instruction set
- Basic arithmetic & logic instructions
- Processor registers
- Getting data to/from the memory
- Control-flow instructions
- Method calls
Processor instructions

- **Instruction set architecture (ISA):** the interface between the software and the hardware
  - Collection of all machine instructions recognized by a particular processor
  - Also, privilege levels, memory management, etc.

- The ISA abstracts away the hardware details from the programmer
  - Similar to how an object hides its implementation details from its users
  - Enables multiple implementations (called microarchitectures) of the same ISA.
Assembly language

- Instructions are just strings of binary numbers
  - Example: 00000011110100100000001000001011
  - For a human, very hard to make out which instruction is which (a sequence of 32-64 bits)

- **Assembly language**: symbolic representation of machine instructions
  - Assembly language makes it easy for humans to read and write machine code
MIPS assembly: a simple example

High-level language (HLL): \( a[0] = b[0] + 10 \)

MIPS assembly language:

\[
\begin{align*}
\text{lw} & \quad r4, 0(r2) \quad \# \text{ get the value of } b[0] \text{ from memory} \\
& \quad \# \text{ and store it in register } r4 \\
\text{add} & \quad r5, r4, 10 \quad \# \text{ compute } b[0] + 10 \text{ and store into } r5 \\
\text{sw} & \quad r5, 0(r1) \quad \# \text{ store } r5 \text{ into } a[0]
\end{align*}
\]

Things to notice:

– Separate instructions to access data (in memory) & operate on it
  ▪ MIPS does not allow to operate on memory directly
– All instructions have similar format
MIPS arithmetic & logical operations

- Data processing instructions look like:
  operation   destination, 1st operand, 2nd operand
  
  add a, b, c   \( a = b + c \)

  sub a, b, c   \( a = b - c \)

- Bit-wise logical instructions: and, or, xor

- Shift instructions:
  
  sll a, b, shamt   \( a = b \ll \text{shamt} \)

  srl a, b, shamt   \( a = b \gg \text{shamt} \) (logical)

  sra a, b, shamt   \( a = b \gg \text{shamt} \) (arithmetic)
Registers

- ISA places restrictions on instruction operands
  - How many operands and where they come from

- MIPS processors operate on registers only
  - Registers are storage locations inside the processor that hold program variables and control state

- Registers are generally sized to contain machine’s word
  - 32 or 64 bits common today

- Processors have relatively few programmer-exposed registers
  - MIPS has 32
  - x86 has 8-16
MIPS instruction example

- **Assembly:**
  
  \[\text{add}\ $s1,\ $s2,\ $s3\quad \#\quad s1 = s2 + s3\]

- **Binary (R-format – used for arithmetic instructions):**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>18</td>
<td>19</td>
<td>17</td>
<td>0</td>
<td>32_{10}</td>
</tr>
<tr>
<td>0</td>
<td>18</td>
<td>9</td>
<td>8</td>
<td>0</td>
<td>34_{10}</td>
</tr>
</tbody>
</table>

Each and every assembly instructions translates to exactly 1 machine instruction (no choice or ambiguity)
More on MIPS registers

- Most registers are available for any use
  - with a few important exceptions
- Program (C/Java) variables held in regs $s0–$s7
- Temporary variables: $t0–$t9
- Registers with special roles
  - Register 0 ($zero) is hardwired to 0
  - Program Counter (PC) holds address of next instruction to be executed (it is not a general purpose registers)
  - Other special-purpose registers exist
Immediate operands

- What if we need to operate on a constant?
  - Common in arithmetic operations, loop index updates (e.g., \( i++ \)), or even character manipulations (e.g., changing the case of an ASCII character)

- MIPS has dedicated instructions with one constant (\textbf{immediate}) operand
  - e.g. \texttt{addi \$r1, \$r1, 1 \# r1=r1+1}

- General form: \texttt{opi \$r1, \$r2, n}
Loading a constant operand

- Load a (small) constant into a register:
  - Constant is 16 bits and is signed
  
  \[
  \text{addi } s0, zero, n \quad \# \ s0 = n \quad (s0_{31-16} = \text{sign ext}; \quad s0_{15-0} = n)
  \]

- What if need a larger/smaller constant?

- Assembler pseudo-instruction \textit{li} \texttt{reg,constant}
  
  – Translated into 1 instruction for immediates < 16bits and into more instructions for more complicated cases (e.g. 2 instructions for a 32-bit immediate)

  \[
  \text{lui } s1, n1 \quad \# \ s1_{31-16} = n1; \quad s1_{15-0} = 0 \\
  \text{ori } s1, s1, n2 \quad \# \ s1_{31-16} = n1; \quad s1_{15-0} = n2
  \]
Getting at the data

- Programming statement:
  
  \[ g = h + A[1] \]

  where

  - \( h \) is in register \$s1\)
  - \( A[0] \) is the first element of array \( A \) and is pointed to by \$s2\)

- MIPS:

  \[
  \begin{align*}
  \text{lw} & \quad \text{offset} \\
  \text{lw} & \quad \text{memory[4+}$s2$]} \\
  \text{add} & \quad h + \text{$t1$}
  \end{align*}
  \]

  \( 2^{32} - 4 \)
Data-transfer instructions

- **Load Word:**
  \[ lw \ r1,n(r2) \quad \# \quad r1=memory[n+r2] \]

- **Store Word:**
  \[ sw \ r1,n(r2) \quad \# \quad memory[n+r2]=r1 \]

- **Load Byte:**
  \[ lb \ r1,n(r2) \quad \# \quad r1_{7-0}=\ memory[n+r2]\]
  \[ r1_{31-8}=\ sign \ extension \]

- **Store Byte:**
  \[ sb \ r1,n(r2) \quad \# \quad memory[n+r2]=r1_{7-0}\]
  \[ no \ sign \ extension \]
Memory addressing

- Memory is **byte addressable**, but it is organised so that a whole word can be accessed directly.

- Where can a word be stored?
  - Option 1: anywhere (**unaligned**)
  - Option 2: at an address that is a multiple of the word size (**aligned**)
  - Both options in use today: MIPS requires alignment, x86 doesn’t
  - What are the trade-offs?
Memory addressing: Endianness

Given a memory address, **Endianness** tells us where to find the starting byte of a word.
The history of the term “Endianness”

D. Cohen introduced the terms Little- and Big-Endian for byte ordering in a 1981 IEEE Computer article. In his analysis of byte ordering issues, the "endian" names were drawn from Jonathan Swift's satire “Gulliver’s Travels”, in which civil war erupts over whether the big or the small end of a soft-boiled egg is the proper end to crack open.  


From the article:

This is an attempt to stop a war. I hope it is not too late and that somehow, magically perhaps, peace will prevail...

… the issue is: "What is the proper byte order in [network] messages?"

For reference, note that Lilliput and Little-Endians both start with "L", and that both Blefuscu and Big-Endians start with a "B".
Instruction formats

- Instruction representation composed of **bit-fields**
- Similar instructions have the same format
- **MIPS instruction formats:**
  - **R-format** (**add/sub, and/or, sll/slr, ...**)
    - 6 5 5 5 5 6 (number of bits)
    - op | rs | rt | rd | shamt | func
      - Main opcode | 1st operand | 2nd operand | result | shift | sub-function opcode
  - **I-format** (**addi, lw, sw, ...**)
    - 6 5 5 16 (number of bits)
    - op | rs | rt | immediate
      - 1st operand | result operand
A simple program to swap array elements

```c
int main(void) {
    int size = 6;
    int v[] = {1, 10, 2, 20, 3, 30}; // array w/ 6 elements

    for (int i=0; i<size; i+=2)
        swap(v, i); // pass array (by reference) and index i
}

void swap(int v[], int idx) {
    int temp;
    temp = v[idx];
    v[idx] = v[idx+1];
    v[idx+1] = temp;
}
```
Swap() in MIPS

swap:

# inputs: $a0 - array base, $a1 - index
# Compute the address into the array
sll $t0, $a1, 2    # reg $t1 = idx * 4
add $t0, $a0, $t0 # reg $t1 = v + (idx*4)
    # $t1 holds the addr of v[idx]

# Load the two values to be swapped
lw  $t1, 0($t0)    # reg $t0 = v[idx]
lw  $t2, 4($t0)    # reg $t2 = v[idx+1]

# Store the swapped values back to memory
sw  $t2, 0($t0)    # v[idx] = $t2
sw  $t1, 4($t0)    # v[idx+1] = $t0
MIPS instructions – part 2

- Last time:
  - Data processing instructions: add, sub, and, …
    - Registers only and immediate types
  - Data transfer instructions: lw, sw, lb, sb
  - Instruction encoding

- Today:
  - Control transfer instructions
Control transfers: If structures

Java/C:

```java
if (i!=j) {
    stmt1
} else {
    stmt2
    stmt3
}
```

MIPS: “branch if equal”

```mips
beq $s1,$s2,label
```

- compare value in $s1 with value in $s2
- if equal, branch to instruction marked label

```mips
beq $s1,$s2,label2
```

```mips
stmt1
```

```mips
j label3  # skip stmt2
```

```mips
label2: stmt2
```

```mips
label3: stmt3
```

“if case”

“else case”

“follow through”
Control transfer instructions

- Conditional branches, I-format: \texttt{beq r1,r2,label}

\begin{center}
\begin{tabular}{c|c|c|c}
6 & 5 & 5 & 16 \\
4 & r1 & r2 & offset \\
\end{tabular}
\end{center}

- In assembly code, label is usually a string
- In machine code, label is obtained from the immediate operand as: \texttt{branch target = PC + 4 \times \text{offset}}

- Similarly: \texttt{bne r1,r2,label} \ # if r1\!\!=\!\!r2 go to label

- Unconditional jump, J-format: \texttt{j label}

\begin{center}
\begin{tabular}{c|c}
6 & 26 \\
2 & target \\
\end{tabular}
\end{center}
Loops in assembly language

- **Java/C:** \( \text{while (count!}=0) \text{ stmt} \)
- **MIPS:**
  
  ```assembly
  loop:
      beq $s1,$zero,end # $s1 holds count
  stmt
  j loop # branch back to loop
  end:  ...
  ```
Loops in assembly language

- **Java/C:**
  
  ```java
  while (flag1 && flag2) stmt
  ```

- **MIPS:**
  ```
  loop:
  beq $s1,$zero,end  # $s1 holds flag1
  beq $s2,$zero,end  # $s2 holds flag2
  stmt
  j loop  # branch back to loop
  end:   ...
  ```
Comparisons

- “Set if less than” (R-format): `slt r1, r2, r3`
  - set r1 to 1 if r2<r3, otherwise set r1 to 0

- E.g., Java/C: `while (i > j) stmt`

- MIPS:
  - assume that `$s1` contains `i` and `$s2` contains `j`

```
loop:
  slt $t0,$s2,$s1    # $t0 = (j < i)
  beq $t0,$zero,end  # branch if i <= j
  stmt
  j loop  # jump back to loop
end:  ...
```

Inf2C Computer Systems - 2016-2017. © Boris Grot
MIPS Instruction Format Summary

- **R-type (register to register)**
  - three register operands
  - most arithmetic, logical and shift instructions

- **I-type (register with immediate)**
  - instructions which use two registers and a constant
  - arithmetic/logical with immediate operand
  - load and store
  - branch instructions with relative branch distance

- **J-type (jump)**
  - jump instructions with a 26 bit address
Practice problem:

What C code does the following piece of MIPS assembly code correspond to?

```
slt $t0, $s1, $s2
beq $t0, $zero, l1
and $s3, $s1, $s2
j 12
l1: or $s3, $s2, $s1
l2:
```

(a) if (s1 < s2) s3 = s2 | s1; else s3 = s1 & s2;
(b) if (s1 <= s2) s3 = s2 | s1; else s3 = s1 & s2;
(c) if (s1 < s2) s1 = s3 | s2; else s2 = s3 & s1;
(d) if (s1 <= s2) s2 = s3 & s1; else s1 = s3 | s2;
(e) if (s1 < s2) s3 = s1 & s2; else s3 = s2 | s1;
Common MIPS Arithmetic & Logical Operators

- **Integer Arithmetic**
  - +      add
  - -      sub
  - *      multiply
  - /      divide
  - %      remainder

- **Shifts**
  - >> (signed)  shift-right-arithmetic
  - >> (unsigned) shift-right-logical
  - <<                shift-left-logical

- **Bit-wise logic**
  - |      or
  - &      and
  - ^      xor
  - ~      not

- **Boolean**
  - ||     (src1 != 0 or src2 != 0)
  - &&     (src1 != 0 and src2 != 0)
Common MIPS Relational Operators

- **Relational**
  - `<` slt, sltu, slti, sltiu
  - `<=` sle, sleu
  - `>` sgt, sgtu
  - `>=` sge, sgeu
  - `==` seq
  - `!=` sne

**Pseudo-instructions**

<table>
<thead>
<tr>
<th>C operator</th>
<th>Comparison</th>
<th>Reverse</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>seq</td>
<td>0</td>
<td>bne</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>seq</td>
<td>0</td>
<td>beq</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>slt, sltu, ..</td>
<td>0</td>
<td>bne</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>slt, sltu, ..</td>
<td>1</td>
<td>bne</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>slt, sltu, ..</td>
<td>1</td>
<td>beq</td>
</tr>
</tbody>
</table>
Method calls

- Method calls are essential even for a small program
- Most ISAs provide support for method calls
- Java/C:

```java
void foo() {
    ...
    return;
}
```

where do we return to?
MIPS support for method calls

- Jumping into the method: `jal label`
  - “jump and link”:
    - set $ra to PC+4
    - set PC to label
  - J-format instruction

- Returning: `jr ra`
  - “jump register”: set PC to value in register $ra
  - Note that any register can be used as a jump target
MIPS register convention on method calls

- Method parameters: $a0 - $a4
- Return values: $v0, $v1
- Regs preserved across call boundaries: $s0 - $s7
- Regs not preserved across call boundaries: $t0 - $t9

What about nested method calls?
Using a stack for method calls

- Nested calls ⇒ must save return address to prevent overwriting. Solution: use a stack in memory.
Using a stack for method calls

- Nested calls ⇒ must save return address to prevent overwriting. Solution: use a stack in memory

- to push a word:
  
  ```
  addi $sp,$sp,-4  # move sp down
  sw $ra,0($sp)  # save ra on top of stack
  ```

- to pop a word:
  
  ```
  lw $ra,0($sp)  # fetch value from stack
  addi $sp,$sp,4  # move sp up
  ```

In MIPS, $sp always points to the last valid word on the stack.
Other uses of the stack

- Stack used to save caller’s registers, so that they can be used by the callee
  - In MIPS, these are the $t registers
  - General issue: “caller save” vs “callee save”

- Stack can also be used to pass & return parameters
  - Gets around the limited number of parameter and return value registers

- Finally, stack is used for local variables within the function
A simple program to swap array elements

```c
int main(void) {
    int size = 6;
    int v[] = {1, 10, 2, 20, 3, 30};  // array w/ 6 elements

    for (int i=0; i<size; i+=2)
        swap(v, i);  // pass array (by reference) and index i
}

void swap(int v[], int idx) {
    int temp;
    temp = v[idx];
    v[idx] = v[idx+1];
    v[idx+1] = temp;
}
```
Main() in MIPS

main:

# Initialize variable
la $s0, array    # $s0: base addr of v[]
addi $s1, $zero, 0 # $s1: index into the array (i)
addi $s2, $zero, 6 # $s1: array size

loop:

beq $s1, $s2, out
move $a0, $s0      # $a0 = $s0 (array base pointer)
move $a1, $s1      # $a1 = $s1 (index)
jal swap            # call swap

addi $s1, $s1, 2   # increment the index
j loop

out:
Swap(): what’s missing?

swap:

# Compute the address into the array
sll $t0, $a1, 2    # reg $t1 = idx * 4
add $t0, $a0, $t0  # reg $t1 = v + (idx*4)
    # $t1 holds the addr of v[idx]

# Load the two values to be swapped
lw  $t1, 0($t0)    # reg $t0 = v[idx]
lw  $t2, 4($t0)    # reg $t2 = v[idx+1]

# Store the swapped values back to memory
sw  $t2, 0($t0)    # v[idx] = $t2
sw  $t1, 4($t0)    # v[idx+1] = $t0
Swap (): complete version

swap:

# Compute the address into the array
sll $t0, $a1, 2    # reg $t1 = idx * 4
add $t0, $a0, $t0  # reg $t1 = v + (idx*4)
    # $t1 holds the addr of v[idx]

# Load the two values to be swapped
lw  $t1, 0($t0)    # reg $t0 = v[idx]
lw  $t2, 4($t0)    # reg $t2 = v[idx+1]

# Store the swapped values back to memory
sw  $t2, 0($t0)    # v[idx] = $t2
sw  $t1, 4($t0)    # v[idx+1] = $t0

jr  $ra            # return to main
Should an ISA be simple or complex?

- ISAs range in complexity
- MIPS is a relatively simple ISA.
  – But is that the right design choice?
- Consider the earlier example:

  High-level language (HLL): \[ a[0] = b[0] + 10 \]
  
  Assembly language:
  
  – Simple: \[ \text{lwr4,0(r2)} \# r4=\text{memory[r2+0]} \]  
  \[ \text{add} r5,r4,10 \# r5=r4+10 \]  
  \[ \text{sw} r5,0(r1) \# \text{memory[r1+0]=r5} \]  
  
  – Complex: \[ \text{ADDW3 (R5),(R2),10} \]

What are the trade-offs??
CISC vs RISC ISAs

- Complex Instruction Set (CISC)
  - Appeared in early computers, including x86
  - Computers programmed in assembly → high-level language features as instructions
  - Very few registers → operands can be in memory
  - Very little memory → variable length instructions to minimize code size

- Reduced Instruction Set (RISC)
  - Appeared in the 80s. Used today in ARM, MIPS, and SPARC ISAs.
  - Compilers → Simple instructions
  - More registers → load-store architecture
  - More memory & faster clock frequency → fixed length, fixed format instructions for easy, fast decoding logic