DEE 1050 Computer Organization
Lecture 3 Instructions: Language of the Computer

Dr. Tian-Sheuan Chang
tschang@twins.ee.nctu.edu.tw
Dept. Electronics Engineering
National Chiao-Tung University
Outline

- Part I. Introduction
- Part II. Case study with MIPS instructions
- Part III. Tool chain
Outline

- Operations
- Operands
- Control flow
- MIPS addressing mode
Introduction

• Computer language
  – Words: instructions
  – Vocabulary: instruction set
  – Similar for all, like regional dialect?

• Design goal of computer language
  – To find a language that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost

• Reading list
  – COD3E: ch. 2
Instructions: Difference with HLL

- Language of the Machine
  - More **primitive** than higher level languages
    - e.g., no sophisticated control flow
  - Very **restrictive**
    - e.g., MIPS Arithmetic Instructions

- We’ll be working with the MIPS instruction set architecture
  - Similar to other architectures developed since the 1980's
  - Almost 100 million MIPS processors manufactured in 2002
  - Used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, …
How to Design the Instructions?

- Operations
  - Arithmetic
  - Logical
  - Datapath

- Operands
  - Datapath

- Control flow
  - Decision control
  - Procedures calls
  - Control

```c
int add5 (int a)
{
    int tmp = a + 5;
    return tmp;
}

void main ()
{
    int a = 7;
    int c;
    if (a == 7)
    {
        c = add5(a);
    }
}
```
Operations: MIPS arithmetic

- Each arithmetic instruction performs only one operation and have 3 operands
- Operand order is fixed (destination first)

```
add a, b, c  # a = b + c
```

“The natural number of operands for an operation like addition is three…requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple”
MIPS arithmetic

- **Design Principle 1:** simplicity favors regularity.
- Of course this complicates some things...

  C code: \[ a = b + c + d; \]
  
  MIPS code:
  
  ```
  add a, b, c
  add a, a, d
  ```

  Q. Will variable no. of operands be faster? how about four or more operands in one instructions
MIPS Logical Operations (Section 2.5)

- Why logical operations
  - Useful to operate on fields of bit or individual bits
- Q. Can some multiply by $2^i$? Divide by $2^i$? Invert?
- Q. Why “NOT” maps to “nor”?
  - $A \text{ NOR } 0 = \text{ NOT } (A \text{ OR } 0) = \text{ NOT } (A)$
- Q. Why not “nori” (nor immediate)?
  - Constant are rare for for NOR

<table>
<thead>
<tr>
<th>operations</th>
<th>c operators</th>
<th>mips</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift left</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>shift right</td>
<td>&gt;&gt;</td>
<td>srl</td>
</tr>
<tr>
<td>bit-by-bit and</td>
<td>&amp;</td>
<td>and, andi</td>
</tr>
<tr>
<td>bit-by-bit or</td>
<td></td>
<td>or, ori</td>
</tr>
<tr>
<td>bit-by-bit not</td>
<td>~</td>
<td>nor</td>
</tr>
</tbody>
</table>
• Intended to be blank
Operands of the Computer Hardware (Section 2.3)

• Difference with HLL like C
  – Limited number, why?
  – Operands are restricted to hardware-built registers
  – Registers are **primitive and visible** to programmer

• MIPS operands
  – Operands must be **registers**, only 32 registers provided
  – Each register contains 32 bits
  – Why 32?

• **Design Principle 2: smaller is faster.**
Operand Type

- 3 Types
  - Register operands
    - All arithmetic operations are in the register operands
  - Memory operands
    - Array or structure
    - Only load/store can access memory
  - Constant or immediate operands
    - Small value will be in the instruction
    - Large value will be stored separately
Register Operand Example

- Register representation
  - `*`, in MIPS
    - `$s0, $s1`.. Registers corresponding to the variables of C programs
    - `$t0, $t1`.. temporary registers need to compile the program
  - (this might be different in other assembly language)

Translate the following C program into MIPS

```
f = (g + h) - (i + j);
Assume f, g, h, i, j uses $s0, .. $s4
```

Add `$t0, $s1, $s2`
Add `$t1, $s3, $s4`
Sub `$s0, $t0 - $t1`

HW/SW IF: How Compiler Use Registers

- Problem: more variables than available registers
- Solution
  - Keep the most frequently used variables in registers
  - Place the rest in memory (called spilling registers), use load and store to move variables between registers and memory
  - Why?
    - Register is faster but its size is small
    - Compiler must use register efficiently
Memory Operands: Array and Structures

- Data are stored in memory
- “data transfer instructions”
  - Transfer data between memory and registers
  - Load \texttt{lw}: move data from memory to a register
  - Store \texttt{st}: move data from a register to memory
Array Example

• Load format
  - lw register names, const offset(base register)

\[ g = h + a[8] \]
assume \( g, h \Rightarrow \$s1, \$s2 \)
base address \( \Rightarrow \$s3 \)

\[ \text{lw } \$t0, 8(\$s3) \quad \# \text{lw: load word} \]
add \( \$s1, \$2, \$t0 \)

If \( a[12] = h + a[8] \)
Add one more instructions
\[ \text{sw 12[\$s3], \$s1} \]
Memory and Data Sizes

So far, we’ve only talked about uniform data sizes. Actual data come in many different sizes:

- **Single bits**: ("boolean" values, true or false)
- **Bytes (8 bits)**: Characters (ASCII), very small integers
- **Halfwords (16 bits)**: Characters (Unicode), short integers
- **Words (32 bits)**: Long integers, floating-point (FP) numbers
- **Double-words (64 bits)**: Very long integers, double-precision FP
- **Quad-words (128 bits)**: Quad-precision floating-point numbers
Different Data Sizes

How do we handle different data sizes?
• Pick one size to be the unit stored in a single address
• Store larger datum in a set of contiguous memory locations
• Store smaller datum in one location; use shift & mask ops

Today, almost all machines (including MIPS) are “byte-addressable” – each addressable location in memory holds 8 bits.
Memory Organization – Byte Addressing

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array.
- "Byte addressing" means that the index points to a byte of memory.

```
0  8 bits of data
1  8 bits of data
2  8 bits of data
3  8 bits of data
4  8 bits of data
5  8 bits of data
6  8 bits of data
...```

8 bits of data
Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.

<table>
<thead>
<tr>
<th>0</th>
<th>32 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>8</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>12</td>
<td>32 bits of data</td>
</tr>
</tbody>
</table>

- $2^{32}$ bytes with byte addresses from 0 to $2^{32}-1$
- $2^{30}$ words with byte addresses 0, 4, 8, ... $2^{32}-4$
- Words are aligned
  - i.e., what are the least 2 significant bits of a word address? To select the byte
- **Alignment restriction in MIPS**
  - Words must start at addresses that are multiples of 4
Array Example for Real MIPS Memory Address

- Code for byte addressable memory

Original
a[12] = h + a[8]
assume g, h => $s1, $s2
base address => $s3

lw $t0, 8($s3)
add $s1, $2, $t0
sw 12($s3), $s1

Updated
a[12] = h + a[8]
assume g, h => $s1, $s2
base address => $s3, word data

lw $t0, 32($s3)
add $s1, $2, $t0
sw 48($s3), $s1

Remember arithmetic operands are registers, not memory!
Can’t write: add 48($s3), $s2, 32($s3)
Byte-Order ("Endianness")

For a multi-byte datum, which part goes in which byte?

- If $1$ contains $1,000,000$ (F4240H) and we store it into address 80:
  - On a "big-endian" machine, the "big" end goes into address 80
    
    \[ \begin{array}{cccc}
    & 00 & 0F & 42 & 40 \\
    \ldots & 79 & 80 & 81 & 82 & 83 & 84 & \ldots \\
    \end{array} \]
  - On a "little-endian" machine, it’s the other way around
    
    \[ \begin{array}{cccc}
    & 40 & 42 & 0F & 00 \\
    \ldots & 79 & 80 & 81 & 82 & 83 & 84 & \ldots \\
    \end{array} \]
Big-Endian vs. Little-Endian

• Big-endian machines: MIPS, Sparc, 68000
• Little-endian machines: most Intel processors, Alpha, VAX

– No real reason one is better than the other…
– Compatibility problems transferring multi-byte data between big-endian and little-endian machines – CAREFUL!

• Bi-endian machines: ARM, User’s choice
Registers Operands vs. Memory Operands

- Arithmetic instructions operands must be registers,
  - only 32 registers provided
  - Compiler associates variables with registers
- What about programs with lots of variables? Like array and structures
  - Data structures are kept in memory
  - **Data transfer instructions**
    - Load: `lw` copy data from memory to registers
    - Store: `sw` copy data from registers to memory
    - How: instruction supplies the memory address
Constant or Immediate Operands

- Small constants are used quite frequently (>50% of operands in SPEC2000 benchmark)
  
e.g., \( A = A + 5; \)  
  \( B = B + 1; \)  
  \( C = C - 18; \)

- Solutions? Why not?
  - put 'constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.

- MIPS Instructions:
  
  ```
  addi $s1, $s1, 5
  slti $s2, $s1, 10
  andi $s1, $s1, 6
  ori $s1, $s1, 4
  ```

- Design Principle 3: Make the common case fast.

- Q: why only “addi” and no “subi”
  - Negative constants
How about larger constants?

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction
  \[
  \text{lui } t0, \text{ 1010101010101010} \]
  \[
  \text{ori } t0, t0, \text{ 1010101010101010} \]
  \[
  \begin{array}{c|c}
  1010101010101010 & 0000000000000000 \\
  \end{array}
  \]

- Then must get the lower order bits right, i.e.,
  \[
  \text{ori } t0, t0, \text{ 1010101010101010} \\
  \begin{array}{c|c}
  1010101010101010 & 0000000000000000 \\
  0000000000000000 & 1010101010101010 \\
  \end{array}
  \]
### MIPS assembly language

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td>Data transfer</td>
<td>load word</td>
<td>lw $s1,100 ($s2)</td>
<td>&amp;$s1 = Memory [$s2 + 100]</td>
<td>Data from memory to register</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $s1,100 ($s2)</td>
<td>Memory [$s2 + 100] = $s1</td>
<td>Data from register to memory</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>branch on equal</td>
<td>beq $s1, $s2, L</td>
<td>if ($s1 == $s2) go to L</td>
<td>Equal test and branch</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>bne $s1, $s2, L</td>
<td>if ($s1 != $s2) go to L</td>
<td>Not equal test and branch</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td>slt $s1, $s2, $s3</td>
<td>if ($s2 &lt; $s3) $s1 = 1; else $s1 = 0</td>
<td>Compare less than: for beq, bne</td>
</tr>
<tr>
<td>Unconditional jump</td>
<td>jump</td>
<td>j 2500</td>
<td>go to 10000</td>
<td>jump to target address</td>
</tr>
<tr>
<td></td>
<td>jump register</td>
<td>jr $sra</td>
<td>go to $sra</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>jal 2500</td>
<td>$sra = PC + 4; go to 1000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>
INFO: MIPS Registers

• 32 regs with R0 = 0
• Reserved registers : R1, R26, R27.
• Special usage:
  – R28: pointer to global area
  – R29: stack pointer
  – R30: frame pointer
  – R31: return address
INFO: Standard Register Conventions

• The 32 integer registers in the MIPS are “general-purpose” – any can be used as an operand or result of an arithmetic operation.

• But making different pieces of software work together is easier if certain conventions are followed concerning which registers are to be used for what purposes.

• These conventions are usually suggested by the vendor and supported by the compilers.
INFO: MIPS Registers and Usage Convention

<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-$a3</td>
<td>4-7</td>
<td>arguments</td>
</tr>
<tr>
<td>$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>

Register 1 ($at) reserved for assembler, 26-27 for operating system
## INFO: MIPS Registers and Usage Convention

<table>
<thead>
<tr>
<th>Register name</th>
<th>Number</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</td>
<td>2</td>
<td>Expression evaluation and results of a function</td>
</tr>
<tr>
<td>v1</td>
<td>3</td>
<td>Expression evaluation and results of a function</td>
</tr>
<tr>
<td>a0</td>
<td>4</td>
<td>Argument 1</td>
</tr>
<tr>
<td>a1</td>
<td>5</td>
<td>Argument 2</td>
</tr>
<tr>
<td>a2</td>
<td>6</td>
<td>Argument 3</td>
</tr>
<tr>
<td>a3</td>
<td>7</td>
<td>Argument 4</td>
</tr>
<tr>
<td>t0</td>
<td>8</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>t1</td>
<td>9</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>t2</td>
<td>10</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>t3</td>
<td>11</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>t4</td>
<td>12</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>t5</td>
<td>13</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>t6</td>
<td>14</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>t7</td>
<td>15</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>s0</td>
<td>16</td>
<td>Saved temporary (preserved across call)</td>
</tr>
<tr>
<td>s1</td>
<td>17</td>
<td>Saved temporary (preserved across call)</td>
</tr>
<tr>
<td>s2</td>
<td>18</td>
<td>Saved temporary (preserved across call)</td>
</tr>
<tr>
<td>s3</td>
<td>19</td>
<td>Saved temporary (preserved across call)</td>
</tr>
<tr>
<td>s4</td>
<td>20</td>
<td>Saved temporary (preserved across call)</td>
</tr>
<tr>
<td>s5</td>
<td>21</td>
<td>Saved temporary (preserved across call)</td>
</tr>
<tr>
<td>s6</td>
<td>22</td>
<td>Saved temporary (preserved across call)</td>
</tr>
<tr>
<td>s7</td>
<td>23</td>
<td>Saved temporary (preserved across call)</td>
</tr>
<tr>
<td>t8</td>
<td>24</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>t9</td>
<td>25</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>k0</td>
<td>26</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>k1</td>
<td>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 call)</td>
</tr>
</tbody>
</table>
Our First Example

- Can we figure out the code?

```c
swap(int v[], int k);
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

```
multi $2, $5, 4
add $2, $4, $2
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
jr $31
```
So far we’ve learned:

- MIPS
  - loading words but addressing bytes
  - arithmetic on registers only

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory[$s2+100]</td>
</tr>
<tr>
<td>sw $s1, 100($s2)</td>
<td>Memory[$s2+100] = $s1</td>
</tr>
</tbody>
</table>
• Intended to be blank
MIPS: 32-bit instruction and data
MIPS fields are given names to make them easier to discuss: (R-type)

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

Here is the meaning of each name of the fields in MIPS instructions:

- **op**: operation of the instruction, called the **opcode**
- **rs**: the first register source operand
- **rt**: the second register source operand
- **rd**: the register destination operand; it gets the result of the operation
- **shamt**: shift amount
- **funct**: function; this field selects the **variant** of the operation in the op field called **function code**
**Instruction Format: Example**

- **Instructions**, like registers and words of data, are also **32 bits long**
  - Example: `add $t1, $s1, $s2`
  - registers have numbers, `$t1=9, s1=17, s2=18`
  - *why?*

- **Instruction Format:**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>
What if Longer Field is Required?

• Consider the load-word and store-word instructions
  – Load word: two registers and a constant
  – **Constant < 32** if any above 5-bit fields is used
  – What would the regularity principle have us do?
  – **Principle 4: Good design demands a compromise**

• Introduce a **new type of instruction format**
  – **I-type** for immediate and data transfer instructions
  – other format was **R-type** for register

• **Example:** `lw $t0, 32($s2)`

  
<table>
<thead>
<tr>
<th>35</th>
<th>18</th>
<th>9</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>16 bit number</td>
</tr>
</tbody>
</table>

• Where's the compromise?
  – Keep instruction the same length with different formats
  – Keep the formats similar
Data Transfer Instructions

- I-type (base + 16 bit offsets)

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

`lw`  
t0, 8 ($s3) --- # Temporary reg t0 gets A[8]

Note: s3 stores the start address of A

Also, rs is the base register, rt stores the destination register.
## Complete MIPS Instruction Formats

<table>
<thead>
<tr>
<th>R-Format</th>
<th>I-Format</th>
<th>J-Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>op</td>
<td>op</td>
</tr>
<tr>
<td>rs</td>
<td>rs</td>
<td></td>
</tr>
<tr>
<td>rt</td>
<td>rt</td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shamt</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>funct</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
<tr>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
</tr>
<tr>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
</tr>
<tr>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>6 bits</td>
<td></td>
</tr>
</tbody>
</table>

### Simple and regular format
Fields in MIPS Instructions

- **op**: Specifies the operation; tells which format to use
- **rs**: First source register
- **rt**: Second source register (or dest. For load)
- **rd**: Destination register
- **shamt**: Shift amount
- **funct**: Further elaboration on opcode
- **address**: Immediate constant, displacement, or branch target
BIG PICTURE: Stored Program Concept

- Instructions are represented as **numbers**
- **Programs** are stored in memory — to be read or written just like data

![Processor and Memory Diagram]

- **Fetch & Execute Cycle**
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the “next” instruction and continue

- **Consequence**
  - **Binary compatibility** due to number representation