MIPS Assembly

Design of Digital Circuits 2017
Srdjan Capkun
Onur Mutlu
(Guest starring: Frank K. Gürkaynak and Aanjhan Ranganathan)

http://www.syssec.ethz.ch/education/Digitaltechnik_17

Adapted from Digital Design and Computer Architecture, David Money Harris & Sarah L. Harris ©2007 Elsevier
In This Lecture

- Assembly Language
- Architecture Design Principles
  - Simplicity favors regularity
  - Make the common case fast
  - Smaller is faster
  - Good design demands good compromises
- Where to store data (memory/register)
- Main types of MIPS instructions
  - (R)egister type
  - (I)mmediate type
  - (J)ump type
Introduction

- Jumping up a few levels of abstraction
- **Architecture:** the programmer’s view of the computer
  - Defined by instructions (operations) and operand locations
- **Microarchitecture:** Implementation of an architecture (Chapter 7)

### Abstraction Levels

<table>
<thead>
<tr>
<th>Abstraction Levels</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Software</td>
<td>Programs</td>
</tr>
<tr>
<td>Operating Systems</td>
<td>Device drivers</td>
</tr>
<tr>
<td>Architecture</td>
<td>Instructions, Registers</td>
</tr>
<tr>
<td>Micro architecture</td>
<td>Datapath, Controllers</td>
</tr>
<tr>
<td>Logic</td>
<td>Adders, Memories</td>
</tr>
<tr>
<td>Digital Circuits</td>
<td>AND gates, NOT gates</td>
</tr>
<tr>
<td>Analog Circuits</td>
<td>Amplifiers</td>
</tr>
<tr>
<td>Devices</td>
<td>Transistors, Diodes</td>
</tr>
<tr>
<td>Physics</td>
<td>Electrons</td>
</tr>
</tbody>
</table>
Assembly Language

- To command a computer, you must understand its language
  - *Instructions*: words in a computer’s language
  - *Instruction set*: the vocabulary of a computer’s language

- Instructions indicate the operation to perform and the operands to use
  - *Assembly language*: human-readable format of instructions
  - *Machine language*: computer-readable format (1’s and 0’s)

- MIPS architecture:
  - Developed by John Hennessy and colleagues at Stanford in the 1980’s
  - Used in many commercial systems (Silicon Graphics, Nintendo, Cisco)

- Once you’ve learned one architecture, it’s easy to learn others
Architecture Design Principles

- Underlying design principles, as articulated by Hennessy and Patterson:
  - Simplicity favors regularity
  - Make the common case fast
  - Smaller is faster
  - Good design demands good compromises
MIPS Instructions: Addition

High-level code

\[
a = b + c;
\]

MIPS assembly

\[
\text{add } a, b, c
\]

- **add**: mnemonic indicates what operation to perform
- **b, c**: source operands on which the operation is performed
- **a**: destination operand to which the result is written
MIPS Instructions: Subtraction

**High-level code**

\[ a = b - c; \]

**MIPS assembly**

`sub a, b, c`

- Subtraction is similar to addition, **only mnemonic changes**
- **sub**: mnemonic indicates what operation to perform
- **b, c**: source operands on which the operation is performed
- **a**: destination operand to which the result is written
Design Principle 1

Simplicity favors regularity

- Consistent instruction format

- Same number of operands (two sources and one destination)
  - easier to encode and handle in hardware
Instructions: More Complex Code

<table>
<thead>
<tr>
<th>High-level code</th>
<th>MIPS assembly code</th>
</tr>
</thead>
</table>
| \( a = b + c - d; \) | \( \text{add } t, b, c \quad \# t = b + c \)  
|                  | \( \text{sub } a, t, d \quad \# a = t - d \) |

More complex code is handled by multiple MIPS instructions.
Design Principle 2

Make the common case fast

- MIPS includes only simple, commonly used instructions
- Hardware to decode and execute the instruction can be simple, small, and fast
- More complex instructions (that are less common) can be performed using multiple simple instructions
RISC and CISC

- **Reduced instruction set computer (RISC)**
  - means: small number of simple instructions
  - example: MIPS

- **Complex instruction set computers (CISC)**
  - means: large number of instructions
  - example: Intel’s x86
Operands

- A computer needs a physical location from which to retrieve binary operands

- A computer retrieves operands from:
  - Registers
  - Memory
  - Constants (also called immediates)
Operands: Registers

- Main Memory is slow

- Most architectures have a small set of (fast) registers
  - MIPS has thirty-two 32-bit registers

- MIPS is called a 32-bit architecture because it operates on 32-bit data
  - A 64-bit version of MIPS also exists, but we will consider only the 32-bit version
Design Principle 3

Smaller is Faster

- MIPS includes only a small number of registers

- Just as retrieving data from a few books on your table is faster than sorting through 1000 books, retrieving data from 32 registers is faster than retrieving it from 1000 registers or a large memory.
# The MIPS Register Set

<table>
<thead>
<tr>
<th><strong>Name</strong></th>
<th><strong>Register Number</strong></th>
<th><strong>Usage</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>assembler temporary</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>procedure return values</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>procedure 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 variables</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$k0-$k1</td>
<td>26-27</td>
<td>OS 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>procedure return address</td>
</tr>
</tbody>
</table>
Operands: Registers

- **Written with a dollar sign ($) before their name**
  - For example, register 0 is written “$0”, pronounced “register zero” or “dollar zero”

- **Certain registers used for specific purposes:**
  - $0 always holds the constant value 0
  - The **saved registers**, $s0-$s7, are used to hold variables
  - The **temporary registers**, $t0 - $t9, are used to hold intermediate values during a larger computation

- For now, we only use the temporary registers ($t0 - $t9) and the saved registers ($s0 - $s7)

- We will use the other registers in later slides
Instructions with registers

High-level code

\[ a = b + c; \]

MIPS assembly

\[
# \text{"$s0 = a"} \\
# \text{"$s1 = b"} \\
# \text{"$s2 = c"} \\
\text{add } $s0, $s1, $s2
\]

- Revisit add instruction
  - The source and destination operands are now in registers
Operands: Memory

■ Too much data to fit in only 32 registers

■ Store more data in memory
  ▪ Memory is large, so it can hold a lot of data
  ▪ But it’s also slow

■ Commonly used variables kept in registers

■ Using a combination of registers and memory, a program can access a large amount of data fairly quickly
Word-Addressable Memory

- Each 32-bit data word has a unique address

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000003</td>
<td>4 0 F 3 0 7 8 8</td>
<td>3</td>
</tr>
<tr>
<td>0000000002</td>
<td>0 1 E E 2 8 4 2</td>
<td>2</td>
</tr>
<tr>
<td>0000000001</td>
<td>F 2 F 1 A C 0 7</td>
<td>1</td>
</tr>
<tr>
<td>0000000000</td>
<td>A B C D E F 7 8</td>
<td>0</td>
</tr>
</tbody>
</table>
Reading Word-Addressable Memory

- Memory reads are called loads
- Mnemonic: load word (lw)
- Example: read a word of data at memory address 1 into $s3

```
lw $s3, 1($0)  # read memory word 1 into $s3
```
Reading Word-Addressable Memory

- **Example:** read a word of data at memory address 1 into $s3

- **Memory address calculation:**
  - add the base address ($0) to the offset (1)
  - address = ($0 + 1) = 1
  - $s3 holds the value **0xF2F1AC07** after the instruction completes

- **Any register may be used to store the base address**

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000003</td>
<td>4 0 F 3 0 7 8 8</td>
<td>Word 3</td>
</tr>
<tr>
<td>00000002</td>
<td>0 1 E E 2 8 4 2</td>
<td>Word 2</td>
</tr>
<tr>
<td>00000001</td>
<td>F 2 F 1 A C 0 7</td>
<td>Word 1</td>
</tr>
<tr>
<td>00000000</td>
<td>A B C D E F 7 8</td>
<td>Word 0</td>
</tr>
</tbody>
</table>
Writing Word-Addressable Memory

- Memory writes are called stores
- Mnemonic: store word (sw)
- Example: Write (store) the value held in $t4 into memory address 7

```
sw $t4, 0x7($0)  # write the value in $t4
               # to memory word 7
```
Writing Word-Addressable Memory

- **Example:** Write (store) the value held in $t4 into memory address 7

- **Memory address calculation:**
  - add the base address ($0) to the offset (7)
  - address = ($0 + 7) = 7
  - Offset can be written in decimal (default) or hexadecimal

- **Any register may be used to store the base address**

```
sw $t4, 0x7($0)  # write the value in $t4
               # to memory word 7
```
Byte-Addressable Memory

- Each data byte has a unique address

- Load/store words or single bytes: load byte (lb) and store byte (sb)

- Each 32-bit words has 4 bytes, so the word address increments by 4. **MIPS uses byte addressable memory**

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Data</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000C</td>
<td>4 0 F 3 0 7 8 8</td>
<td>Word 3</td>
</tr>
<tr>
<td>000000008</td>
<td>0 1 E E 2 8 4 2</td>
<td>Word 2</td>
</tr>
<tr>
<td>000000004</td>
<td>F 2 F 1 A C 0 7</td>
<td>Word 1</td>
</tr>
<tr>
<td>000000000</td>
<td>A B C D E F 7 8</td>
<td>Word 0</td>
</tr>
</tbody>
</table>

width = 4 bytes
Reading Byte-Addressable Memory

- Load a word of data at memory address 4 into $s3.

- Memory address calculation:
  - add the base address ($0) to the offset (4)
  - address = ($0 + 4) = 4

- $s3$ holds the value 0xF2F1AC07 after the instruction completes

```
lw $s3, 4($0)  # read word at address 4 into $s3
```
Writing Byte-Addressable Memory

- Example: store the value held in $t7 into the eleventh 32-bit memory location.

- Memory address calculation:
  - Byte addressable address for word eleven
    \[11 \times 4 = 44_{10} = 0x2C_{16}\]
  - add the base address ($0) to the offset (0x2c)
  - address = ($0 + 44) = 44

```
sw $t7, 44($0)  # write $t7 into address 44
```
Big-Endian and Little-Endian Memory

- How to number bytes within a word?

- Word address is the same for big- or little-endian
  - *Little-endian:* byte numbers start at the little (least significant) end
  - *Big-endian:* byte numbers start at the big (most significant) end

<table>
<thead>
<tr>
<th>Byte Address</th>
<th>Word Address</th>
<th>Byte Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
</tr>
<tr>
<td>C D E F</td>
<td>C</td>
<td>F E D C</td>
</tr>
<tr>
<td>8 9 A B</td>
<td>8</td>
<td>B A 9 8</td>
</tr>
<tr>
<td>4 5 6 7</td>
<td>4</td>
<td>7 6 5 4</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>0</td>
<td>3 2 1 0</td>
</tr>
</tbody>
</table>
Big-Endian and Little-Endian Memory

- From Jonathan Swift’s Gulliver’s Travels where the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end.
  - As indicated by the farcical name, it doesn’t really matter which addressing type is used – except when the two systems need to share data!

<table>
<thead>
<tr>
<th>Byte Address</th>
<th>Word Address</th>
<th>Byte Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>C D E F</td>
<td>C</td>
<td>F E D C</td>
</tr>
<tr>
<td>8 9 A B</td>
<td>8</td>
<td>B A 9 8</td>
</tr>
<tr>
<td>4 5 6 7</td>
<td>4</td>
<td>7 6 5 4</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>0</td>
<td>3 2 1 0</td>
</tr>
</tbody>
</table>

MSB       LSB       MSB       LSB
### Big- and Little-Endian Example

Suppose $t0$ initially contains 0x23456789. After the following program is run on a big-endian system, what value does $s0$ contain? In a little-endian system?

```
sw $t0, 0($0)
lb $s0, 1($0)
```

<table>
<thead>
<tr>
<th>Big-Endian</th>
<th>Little-Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Word Address</strong></td>
<td><strong>Byte Address</strong></td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>3 2 1 0</td>
</tr>
<tr>
<td>Data Value</td>
<td>Data Value</td>
</tr>
<tr>
<td>23 45 67 89</td>
<td>23 45 67 89</td>
</tr>
<tr>
<td>MSB</td>
<td>MSB</td>
</tr>
<tr>
<td>LSB</td>
<td>LSB</td>
</tr>
</tbody>
</table>
Big- and Little-Endian Example

- Suppose $t0$ initially contains 0x23456789. After the following program is run on a big-endian system, what value does $s0$ contain? In a little-endian system?

```
sw $t0, 0($0)
lb $s0, 1($0)
```

- Big-endian: 0x00000045
  Little-endian: 0x00000067
Design Principle 4

Good design demands good compromises

- **Multiple instruction formats allow flexibility**
  - `add, sub:` use 3 register operands
  - `lw, sw:` use 2 register operands and a constant

- **Number of instruction formats kept small**
  - to adhere to design principles 1 and 3 (simplicity favors regularity and smaller is faster)
Operands: Constants/Immediates

**High-level code**

```plaintext
a = a + 4;
b = a - 12;
```

**MIPS assembly code**

```plaintext
# $s0 = a, $s1 = b
addi $s0, $s0, 4
addi $s1, $s0, -12
```

- **lw and sw illustrate the use of constants or immediates**
  - Called immediates because they are directly available
  - Immediates don’t require a register or memory access

- **The add immediate (addi) instruction adds an immediate to a variable (held in a register)**
  - An immediate is a 16-bit two’s complement number

- **Is subtract immediate (subi) necessary?**

Machine Language

- Computers only understand 1’s and 0’s
- Machine language: binary representation of instructions
- 32-bit instructions
  - Again, simplicity favors regularity: 32-bit data, 32-bit instructions, and possibly also 32-bit addresses
- Three instruction formats:
  - **R-Type**: register operands
  - **I-Type**: immediate operand
  - **J-Type**: for jumping (we’ll discuss later)
R-Type

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>

- **Register-type, 3 register operands:**
  - \(rs, rt\): source registers
  - \(rd\): destination register

- **Other fields:**
  - \(op\): the operation code or opcode (0 for R-type instructions)
  - \(funct\): the function together, the opcode and function tell the computer what operation to perform
  - \(shamt\): the shift amount for shift instructions, otherwise it’s 0
R-Type Examples

**Assembly Code**

```
add $s0, $s1, $s2
sub $t0, $t3, $t5
```

**Field Values**

<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>0</td>
<td>17</td>
<td>18</td>
<td>16</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>13</td>
<td>8</td>
<td>0</td>
<td>34</td>
</tr>
</tbody>
</table>

<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>10000</td>
<td>00000</td>
<td>100000</td>
</tr>
<tr>
<td>000000</td>
<td>01011</td>
<td>01101</td>
<td>01000</td>
<td>00000</td>
<td>100010</td>
</tr>
</tbody>
</table>

**Machine Code**

```
(0x02328020)
(0x016D4022)
```

*Note* the order of registers in the assembly code: `add rd, rs, rt`
I-Type

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

- Immediate-type, has 3 operands:
  - rs, rt: register operands
  - imm: 16-bit two’s complement immediate

- Other fields:
  - op: the opcode

- Simplicity favors regularity: all instructions have opcode

- Operation is completely determined by the opcode
I-Type Examples

<table>
<thead>
<tr>
<th>Assembly Code</th>
<th>Field Values</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addi $s0, $s1, 5</code></td>
<td>8 17 16 5</td>
<td>(0x22300005)</td>
</tr>
<tr>
<td><code>addi $t0, $s3, -12</code></td>
<td>8 19 8 -12</td>
<td>(0x2268FFF4)</td>
</tr>
<tr>
<td><code>lw  $t2, 32($0)</code></td>
<td>35 0 10 32</td>
<td>(0x8C0A0020)</td>
</tr>
<tr>
<td><code>sw  $s1, 4($t1)</code></td>
<td>43 9 17 4</td>
<td>(0xAD310004)</td>
</tr>
</tbody>
</table>

**Note** the differing order of registers in the assembly and machine codes:

- `addi rt, rs, imm`
- `lw rt, imm(rs)`
- `sw rt, imm(rs)`
Machine Language: J-Type

- **Jump-type**
- **26-bit address operand (addr)**
- **Used for jump instructions (j)**

### J-Type

<table>
<thead>
<tr>
<th>op</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>
Review: Instruction Formats

**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>

**I-Type**

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

**J-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>
The Power of the Stored Program

- 32-bit instructions and data stored in memory

- Sequence of instructions: only difference between two applications (for example, a text editor and a video game)

- To run a new program:
  - No rewiring required
  - Simply store new program in memory

- The processor hardware executes the program:
  - fetches (reads) the instructions from memory in sequence
  - performs the specified operation
Program counter

- The processor hardware executes the program:
  - fetches (reads) the instructions from memory in sequence
  - performs the specified operation
  - continues with the next instruction

- The program counter (PC) keeps track of the current instruction
  - In MIPS, programs typically start at memory address 0x00400000
The Stored Program

<table>
<thead>
<tr>
<th>Address</th>
<th>Instructions</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>00400000C</td>
<td>0 1 6 D 4 0 2 2</td>
<td>0x8C0A0020</td>
</tr>
<tr>
<td>00400008</td>
<td>2 2 6 8 F F F 4</td>
<td>0x02328020</td>
</tr>
<tr>
<td>00400004</td>
<td>0 2 3 2 8 0 2 0</td>
<td>0x2268004</td>
</tr>
<tr>
<td>00400000</td>
<td>8 C 0 A 0 0 2 0</td>
<td>0x016D022</td>
</tr>
</tbody>
</table>

Assembly Code

lw  $t2, 32($0)
add $s0, $s1, $s2
addi $t0, $s3, -12
sub $t0, $t3, $t5

Machine Code

0x8C0A0020
0x02328020
0x2268004
0x016D022
Interpreting Machine Language Code

- **Start with opcode**
  - Opcode tells how to parse the remaining bits

- **If opcode is all 0’s**
  - R-type instruction
  - Function bits tell what instruction it is

- **Otherwise**
  - Opcode tells what instruction it is
Branching

- Allows a program to execute instructions out of sequence

- **Conditional branches**
  - branch if equal: `beq` (I-type)
  - branch if not equal: `bne` (I-type)

- **Unconditional branches**
  - jump: `j` (J-type)
  - jump register: `jr` (R-type)
  - jump and link: `jal` (J-type)

these are the only two J-type instructions
Conditional Branching (beq)

```mips
# MIPS assembly
addi $s0, $0, 4
addi $s1, $0, 1
sll $s1, $s1, 2
beq $s0, $s1, target
addi $s1, $s1, 1
sub $s1, $s1, $s0

target:
add $s1, $s1, $s0
```

**Labels** indicate instruction locations in a program. They cannot use reserved words and must be followed by a colon (`:`).
Conditional Branching (beq)

# MIPS assembly

- `addi $s0, $0, 4`  # $s0 = 0 + 4 = 4
- `addi $s1, $0, 1`  # $s1 = 0 + 1 = 1
- `sll $s1, $s1, 2`  # $s1 = 1 << 2 = 4
- `beq $s0, $s1, target`  # branch is taken
- `addi $s1, $s1, 1`  # not executed
- `sub $s1, $s1, $s0`  # not executed

**target:**  # label

- `add $s1, $s1, $s0`  # $s1 = 4 + 4 = 8

**Labels** indicate instruction locations in a program. They cannot use reserved words and must be followed by a colon (:).
The Branch Not Taken (bne)

# MIPS assembly

```
addi $s0, $0, 4  # $s0 = 0 + 4 = 4
addi $s1, $0, 1  # $s1 = 0 + 1 = 1
sll $s1, $s1, 2  # $s1 = 1 << 2 = 4
bne $s0, $s1, target  # branch not taken
addi $s1, $s1, 1  # $s1 = 4 + 1 = 5
sub $s1, $s1, $s0  # $s1 = 5 - 4 = 1
```

**target:**

```
add $s1, $s1, $s0  # $s1 = 1 + 4 = 5
```
Unconditional Branching / Jumping (j)

# MIPS assembly

```mips
addi $s0, $0, 4  # $s0 = 4
addi $s1, $0, 1  # $s1 = 1
j target         # jump to target
sra $s1, $s1, 2  # not executed
addi $s1, $s1, 1 # not executed
sub $s1, $s1, $s0 # not executed

target:
add $s1, $s1, $s0 # $s1 = 1 + 4 = 5
```
Unconditional Branching (jr)

# MIPS assembly

0x00002000  addi $s0, $0, 0x2010  # load 0x2010 to $s0
0x00002004  jr  $s0           # jump to $s0
0x00002008  addi $s1, $0, 1   # not executed
0x0000200C  sra  $s1, $s1, 2  # not executed
0x00002010  lw  $s3, 44($s1)  # program continues
High-Level Code Constructs

- if statements
- if/else statements
- while loops
- for loops
**If Statement**

**High-level code**

```java
if (i == j)
    f = g + h;

f = f - i;
```

**MIPS assembly code**

```mips
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
```
If Statement

High-level code

```plaintext
if (i == j)
    f = g + h;

f = f - i;
```

MIPS assembly code

```plaintext
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
    bne $s3, $s4, L1
    add $s0, $s1, $s2
L1: sub $s0, $s0, $s3
```

- Notice that the assembly tests for the opposite case (i != j) than the test in the high-level code (i == j)
If / Else Statement

High-level code

```java
if (i == j)
    f = g + h;
else
    f = f - i;
```

MIPS assembly code

```c
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
```
If / Else Statement

**High-level code**

```plaintext
if (i == j)
    f = g + h;
else
    f = f - i;
```

**MIPS assembly code**

```plaintext
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
bne $s3, $s4, L1
add $s0, $s1, $s2
j done
L1:    sub $s0, $s0, $s3
done:
```

---

High-level code

```plaintext
if (i == j)
    f = g + h;
else
    f = f - i;
```

MIPS assembly code

```plaintext
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
bne $s3, $s4, L1
add $s0, $s1, $s2
j done
L1:    sub $s0, $s0, $s3
done:
```
While Loops

**High-level code**

```c
// determines the power
// of x such that 2x = 128
int pow = 1;
int x = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

**MIPS assembly code**

```assembly
# $s0 = pow, $s1 = x
```

---

55
While Loops

**High-level code**

// determines the power of x such that 2x = 128
int pow = 1;
int x = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}

**MIPS assembly code**

# $s0 = pow, $s1 = x
addi $s0, $0, 1
add $s1, $0, $0
addi $t0, $0, 128

while: beq $s0, $t0, done
sll $s0, $s0, 1
addi $s1, $s1, 1
j while

done:

- Notice that the assembly tests for the opposite case (pow == 128) than the test in the high-level code (pow != 128)