Chapter 2
Instructions:
Language of the Computer
Outline

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

• Instruction set architecture
  (taking MIPS ISA as an example)
• Operands (2.3)
  – Register operands and their organization
  – Memory operands, data transfer
  – Immediate operands
• Instruction format
• Operations
  – Arithmetic and logical
  – Decision making and branches
  – Jumps for procedures
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*
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 (運算元)**
  - Arithmetic
  - Logical
  - => Datapath

- **Operands (運算子)**
  - => 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);
}
```
Assembly Language vs. Machine Language

- Assembly provides convenient *symbolic representation*
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is *the underlying reality*
  - e.g., destination is no longer first
- Assembly can provide *pseudoinstructions*
  - e.g., “move $t0, $t1” exists only in Assembly
  - would be implemented using “add $t0,$t1,$zero”
- When considering performance you should count real instructions
Recall in C Language

● Operators: +, −, *, /, % (mod), ...

  − 7/4==1, 7%4==3

● Operands:
  − Variables: lower, upper, fahr, celsius
  − Constants: 0, 1000, −17, 15.4

● Assignment statement:

  variable = expression

  − Expressions consist of operators operating on operands, e.g.,

    celsius = 5*(fahr-32)/9;
    a = b+c+d−e;
When Translating to Assembly ...

a = b + 5;

[Diagram:]
- **Statement**
  - **Constant**
    - Operands
      - Register
      - Memory
      - Operator (op code)

- **load** $r1, M[b]
- **load** $r2, 5
- **add** $r3, $r1, $r2
- **store** $r3, M[a]
Components of an ISA

- Organization of *programmable storage*
  - registers
  - memory: flat, segmented
  - Modes of addressing and accessing data items and instructions

- Data types and data structures
  - encoding and representation (next chapter)

- Instruction formats

- Instruction set (or operation code)
  - ALU, control transfer, exceptional handling
MIPS ISA as an Example

Instruction categories:
- Load/Store
- Computational
- Jump and Branch
- Floating Point
- Memory Management
- Special

3 Instruction Formats: all 32 bits wide

<table>
<thead>
<tr>
<th>OP</th>
<th>$rs</th>
<th>$rt</th>
<th>$rd</th>
<th>sa</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>$rs</td>
<td>$rt</td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
<tr>
<td>OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>jump target</td>
</tr>
</tbody>
</table>
Operations: MIPS arithmetic

- Each arithmetic instructions 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.*
- All arithmetic instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: \( a = b + c \)

MIPS ‘code’: \texttt{add a, b, c}

(we’ll talk about registers in a bit)
MIPS arithmetic

- Of course this complicates some things...

  C code: \[ a = b + c + d; \]

  MIPS code:  
  ```
  add a, b, c
  add a, a, d
  ```

- Operands must be registers, only 32 registers provided
- Each register contains 32 bits
Design Principle

- Simplicity favors regularity
- Smaller is faster
Registers vs. Memory

- Arithmetic instructions operands must be registers, — only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables
Outline

- Instruction set architecture
  (using MIPS ISA as an example)
- Operands of Hardware (2.3)
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Instruction format
- Operations
  - Arithmetic and logical
  - Decision making and branches
  - Jumps for procedures
Operands of the Computer Hardware

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

- MIPS Register operands
  - 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
Operands and Registers

● Unlike high-level language, MIPS assembly don’t use variables
  => assembly operands are registers
    – Limited number of special locations built directly into the hardware
    – Operations are performed on these

● Benefits:
  – Registers in hardware => faster than memory
  – Registers are easier for a compiler to use
    ● e.g., as a place for temporary storage
  – Registers can hold variables to reduce memory traffic and improve code density (since register named with fewer bits than memory location)
MIPS Registers

- 32 registers, each is 32 bits wide
  - Why 32? smaller is faster
  - Groups of 32 bits called a word in MIPS
  - Registers are numbered from 0 to 31
  - Each can be referred to by number or name
  - Number references:
    - $0, $1, $2, ... $30, $31
  - By convention, each register also has a name to make it easier to code, e.g.,
    - $16 - $23 $s0 - $s7 (C variables)
    - $8 - $15 $t0 - $t7 (temporary)

- 32 x 32-bit FP registers (paired DP)
- Others: HI, LO, PC
## Registers Conventions for MIPS

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>callee saves</td>
</tr>
<tr>
<td></td>
<td>(caller can clobber)</td>
</tr>
<tr>
<td>s7</td>
<td>temporary (cont’d)</td>
</tr>
<tr>
<td>t8</td>
<td>temporary (cont’d)</td>
</tr>
<tr>
<td>t9</td>
<td></td>
</tr>
<tr>
<td>k0</td>
<td>reserved for OS kernel</td>
</tr>
<tr>
<td>k1</td>
<td></td>
</tr>
<tr>
<td>gp</td>
<td>pointer to global area</td>
</tr>
<tr>
<td>sp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>fp</td>
<td>frame pointer</td>
</tr>
<tr>
<td>ra</td>
<td>return address (HW)</td>
</tr>
</tbody>
</table>

*Fig. 2.18*
MIPS R2000 Organization

Fig. A.10.1
Register Operand

- Syntax of basic MIPS arithmetic/logic instructions:

  1 2 3 4
  add $s0,$s1,$s2    # f = g + h

  1) operation by name
  2) operand getting result ("destination")
  3) 1st operand for operation ("source1")
  4) 2nd operand for operation ("source2")

- Each instruction is 32 bits
- Syntax is rigid: 1 operator, 3 operands
  - Why? *Keep hardware simple via regularity*
Register Operand Example

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

- How to do the following C statement?

  \[
  f = (g + h) - (i + j);
  \]

Assume $f, g, h, i, j$ uses $\texttt{s0, .. s4}$

- $\texttt{add \ s0, s1, s2}$  \# $f = g + h$
- $\texttt{add \ t0, s3, s4}$  \# $t0 = i + j$
- $\texttt{sub \ s0, s0, t0}$  \# $f = (g+h)-(i+j)$
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
Outline

• Instruction set architecture
  (using MIPS ISA as an example)

• Operands (2.3)
  – Register operands and their organization
  – Memory operands, data transfer
  – Immediate operands

• Instruction format

• Operations
  – Arithmetic and logical
  – Decision making and branches
  – Jumps for procedures
Memory Operands: Array and Structures

- Data are stored in memory
- “data transfer instructions”
  - Transfer data *between memory and registers*
  - Load lw: move data from memory to a register
  - Store st: move data from a register to memory
Memory Operands

- C variables map onto registers; what about large data structures like arrays?
  - Memory contains such data structures
- But MIPS arithmetic instructions operate on registers, not directly on memory
  - *Data transfer instructions* (lw, sw, ...) to transfer between memory and register
  - A way to address memory operands
Array Example

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

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

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

• 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.

<table>
<thead>
<tr>
<th></th>
<th>8 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Memory Organization

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

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 \textit{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
A Note about Memory: Alignment

- MIPS requires that all words start at addresses that are multiples of 4 bytes.

- Called Alignment: objects must fall on address that is multiple of their size.
Array Example for Real MIPS Memory Address

- Code for byte addressable memory

<table>
<thead>
<tr>
<th>Original</th>
<th>Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>assume g, h =&gt; $s1, $s2</td>
<td>assume g, h =&gt; $s1, $s2</td>
</tr>
<tr>
<td>base address =&gt; $s3</td>
<td>base address =&gt; $s3, word data</td>
</tr>
<tr>
<td>lw  $t0, 8($s3)</td>
<td>lw  $t0, 32($s3)</td>
</tr>
<tr>
<td>add  $s1, $2, $t0</td>
<td>add  $s1, $2, $t0</td>
</tr>
<tr>
<td>sw  12($s3),  $s1</td>
<td>sw  48($s3),  $s1</td>
</tr>
</tbody>
</table>

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

    |   | 00 | 0F | 42 | 40 |
    |---|---|---|---|---|
    | ... | 79 | 80 | 81 | 82 | 83 | 84 | ... |

  - On a “little-endian” machine, it’s the other way around

    |   | 40 | 42 | 0F | 00 |
    |---|---|---|---|---|
    | ... | 79 | 80 | 81 | 82 | 83 | 84 | ... |
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
  - \textit{Data transfer instructions}
    - Load: \textit{lw} copy data from memory to registers
    - Store: \textit{sw} copy data from registers to memory
    - How: instruction supplies the memory address
Data Transfer: Memory to Register (1/2)

● To transfer a word of data, need to specify two things:
  – Register: specify this by number (0 - 31)
  – Memory address: more difficult
    ● Think of memory as a 1D array
    ● Address it by supplying a pointer to a memory address
    ● Offset (in bytes) from this pointer
    ● The desired memory address is the sum of these two values, e.g., 8 ($t0)
    ● Specifies the memory address pointed to by the value in $t0, plus 8 bytes (why “bytes”, not “words”?)
    ● Each address is 32 bits
Data Transfer: Memory to Register (2/2)

- Load Instruction Syntax:
  
  1  2  3  4
  
  lw $t0,12($s0)

  1) operation name
  2) register that will receive value
  3) numerical offset in bytes
  4) register containing pointer to memory

- Example: lw $t0,12($s0)
  
  - lw (Load Word, so a word (32 bits) is loaded at a time)
  - Take the pointer in $s0, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register $t0

- Notes:
  
  - $s0 is called the base register, 12 is called the offset
  - Offset is generally used in accessing elements of array: base register points to the beginning of the array
Data Transfer: Register to Memory

- Also want to store value from a register into memory
- Store instruction syntax is identical to Load instruction syntax

Example: \texttt{sw \$t0,12($s0)}
  - \texttt{sw} (meaning Store Word, so 32 bits or one word are loaded at a time)
  - This instruction will take the pointer in \$s0, add 12 bytes to it, and then store the value from register \$t0 into the memory address pointed to by the calculated sum
Compilation with Memory

 Compile by hand using registers:

\[
\begin{align*}
&s1: g, \ s2: h, \ s3: \text{base address of } A \\
g &= h + A[8];
\end{align*}
\]

- What offset in `lw` to select an array element `A[8]` in a C program?
  - 4x8=32 bytes to select `A[8]`
  - 1st transfer from memory to register:
    \[
    \begin{align*}
    \text{lw} & \quad \text{\$t0, 32($s3)} \quad \# \ \text{\$t0 gets } A[8] \\
    \text{add} & \quad \text{32 to \$s3 to select } A[8], \text{put into \$t0}
    \end{align*}
    \]

- Next add it to `h` and place in `g`
  \[
  \begin{align*}
  \text{add} & \quad \text{\$s1, \$s2, \$t0} \quad \# \ \text{\$s1 = h+A[8]}
  \end{align*}
  \]
## MIPS Data Transfer Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw</td>
<td>Store word</td>
</tr>
<tr>
<td>sh</td>
<td>Store half</td>
</tr>
<tr>
<td>sb</td>
<td>Store byte</td>
</tr>
<tr>
<td>lw</td>
<td>Load word</td>
</tr>
<tr>
<td>lh</td>
<td>Load halfword</td>
</tr>
<tr>
<td>lhu</td>
<td>Load halfword unsigned</td>
</tr>
<tr>
<td>lb</td>
<td>Load byte</td>
</tr>
<tr>
<td>lbu</td>
<td>Load byte unsigned</td>
</tr>
<tr>
<td>lui</td>
<td>Load Upper Immediate (16 bits shifted left by 16)</td>
</tr>
</tbody>
</table>

What does it mean?
Load Byte Signed/Unsigned

$\text{lb } \$t1, 0(\$t0) \quad \begin{array}{c} FFFFFFFF \ F7 \end{array} \quad \text{Sign-extended}

\text{lbu } \$t2, 0(\$t0) \quad \begin{array}{c} 000000 \ F7 \end{array} \quad \text{Zero-extended}
Role of Registers vs. Memory

- What if more variables than registers?
  - Compiler tries to keep most frequently used variables in registers
  - Writes less common variables to memory

- Why not keep all variables in memory?
  - Smaller is faster: registers are faster than memory
  - Registers more versatile:
    - MIPS arithmetic instructions can read 2 registers, operate on them, and write 1 per instruction
    - MIPS data transfers only read or write 1 operand per instruction, and no operation
Outline

● Instruction set architecture
  (using MIPS ISA as an example)
● Operands (Sec 2.3)
  – Register operands and their organization
  – Memory operands, data transfer, and addressing
  – Immediate operands
● Instruction format
● Operations
  – Arithmetic and logical
  – Decision making and branches
  – Jumps for procedures
Constant or Immediate Operands

- **Small constants used frequently (>50% of operands in SPEC2000 benchmark)**
  
  e.g.,  
  
  \[
  \begin{align*}
  A &= A + 5; \\
  B &= B + 1; \\
  C &= C - 18;
  \end{align*}
  \]

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

- **MIPS Instructions:**
  - `addi $29, $29, 4`
  - `slti $8, $18, 10`
  - `andi $29, $29, 6`
  - `ori $29, $29, 4`

- **Design Principle:** *Make the common case fast*

- **Q:** why only “addi” and no “subi”
  - Negative constants
Constant or Immediate Operands

- **Immediate: numerical *constants***
  - Often appear in code, so there are special instructions for them
  - Add Immediate:
    \[
    f = g + 10 \quad \text{(in C)}
    \]
    \[
    \text{addi } \$s0,\$s1,10 \quad \text{(in MIPS)}
    \]
    where \$s0,\$s1 are associated with \(f, g\)
  - Syntax similar to `add` instruction, except that last argument is a number instead of a register
  - One particular immediate, the number zero (0), appears very often in code; so we define register zero (\$0 or \$zero) to always 0
  - This is defined in hardware, so an instruction like `addi \$0,\$0,5` will not do anything
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

\[
\text{lui } \$t0, \ 101010101010101010
\]

\[
\begin{array}{c|c}
1010101010101010 & 0000000000000000 \end{array}
\]

- Then must get the lower order bits right, i.e.,

\[
\text{ori } \$t0, \ \$t0, \ 101010101010101010
\]

\[
\begin{array}{c|c}
1010101010101010 & 0000000000000000 \end{array}
\]

\[
\begin{array}{c|c}
0000000000000000 & 1010101010101010 \end{array}
\]
So far

MIPS operands

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 registers</td>
<td>$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra</td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. $gp (28) is the global pointer, $sp(29) is the stack pointer, $fp (30) is the frame pointer, and $ra (31) is the return address.</td>
</tr>
<tr>
<td>$2^{30}$ memory words</td>
<td>Memory [0], Memory [4],..., Memory[42949672920]</td>
<td>Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled register, such as those saved on procedure calls.</td>
</tr>
</tbody>
</table>

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 $ra</td>
<td>go to $ra</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>jal 2500</td>
<td>$ra = 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
### Registers Conventions for MIPS

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_0$</td>
<td>expression evaluation &amp; function results</td>
</tr>
<tr>
<td>$v_1$</td>
<td>function results</td>
</tr>
<tr>
<td>$a_0$</td>
<td>arguments</td>
</tr>
<tr>
<td>$a_1$</td>
<td></td>
</tr>
<tr>
<td>$a_2$</td>
<td></td>
</tr>
<tr>
<td>$a_3$</td>
<td></td>
</tr>
<tr>
<td>$t_0$</td>
<td>temporary: caller saves (callee can clobber)</td>
</tr>
<tr>
<td>$t_7$</td>
<td></td>
</tr>
<tr>
<td>$s_0$</td>
<td>callee saves</td>
</tr>
<tr>
<td></td>
<td>(caller can clobber)</td>
</tr>
<tr>
<td>$s_7$</td>
<td></td>
</tr>
<tr>
<td>$t_8$</td>
<td>temporary (cont’d)</td>
</tr>
<tr>
<td>$t_9$</td>
<td></td>
</tr>
<tr>
<td>$k_0$</td>
<td>reserved for OS kernel</td>
</tr>
<tr>
<td>$k_1$</td>
<td></td>
</tr>
<tr>
<td>$gp$</td>
<td>pointer to global area</td>
</tr>
<tr>
<td>$sp$</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp$</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra$</td>
<td>return address (HW)</td>
</tr>
</tbody>
</table>

**Fig. 2.18**
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 op

• 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;
}
```

swap:
- muli $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

$4: v的start address
$5: index k
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>
Outline

- Instruction set architecture (using MIPS ISA as an example)
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Instruction format (Sec. 2.4.~2.9)
- Operations
  - Arithmetic and logical
  - Decision making and branches
  - Jumps for procedures
MIPS Instruction Format

● One instruction is 32 bits
  => divide instruction word into “fields”
    – Each field tells computer something about instruction

● We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:
  – \textit{R-format}: for register
  – \textit{I-format}: for immediate, and \texttt{lw} and \texttt{sw} (since the offset counts as an immediate)
  – \textit{J-format}: for jump
Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

### R-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>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-Format

<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>

### J-Format

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>
R-Format Instructions (1/2)

- Define the following “fields”:

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

- **opcode**: *operation* of instruction (Note: 0 for all R-Format instructions)
- **rs** (Source Register): *generally* used to specify register containing first operand
- **rt** (Target Register): *generally* used to specify register containing second operand
- **rd** (Destination Register): *generally* used to specify register which will receive result of computation
- **shamt**: shift amount
- **funct**: function; this field selects the *variant* of the operation in the op field called *function code*

Question: Why aren’t opcode and funct a single 12-bit field?
R-Format Instructions (2/2)

● Notes about register fields:
  – Each register field is exactly 5 bits, which means that it can specify any unsigned integer in the range 0-31. Each of these fields specifies one of the 32 registers by number.

● Final field:
  – `shamt`: contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is useless, so this field is only 5 bits
  – This field is set to 0 in all but the shift instructions
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

● Instruction Format:

```
000000 10001 10010 01000 00000 100000
```

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>
R-Format Example

- **MIPS Instruction:**
  
  ```
  add $8, $9, $10  // $8 = $9 + $10
  ```
  
  - opcode = 0 (look up in table)
  - funct = 32 (look up in table)
  - rs = 9 (first operand)
  - rt = 10 (second operand)
  - rd = 8 (destination)
  - shamt = 0 (not a shift)

  **binary representation:**

<table>
<thead>
<tr>
<th>000000</th>
<th>01001</th>
<th>01010</th>
<th>01000</th>
<th>00000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
</tbody>
</table>

  called a **Machine Language Instruction**
What if Longer Field is Required?

- Consider the load-word and store-word instructions
  - Load word: two registers and a constant
  - $\text{Constant} < 32$ if any above 5-bit fields is used
  - What would the regularity principle have us do?
  - Principle 4: \textit{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: \texttt{lw $t0, 32($s2)}

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

- Where's the compromise?
  - Keep instruction the same length with different formats
  - Keep the formats similar
I-Format Instructions

- Define the following “fields”:

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
</tr>
</tbody>
</table>

- opcode: uniquely specifies an I-format instruction
- rs: specifies the only register operand and is the base register
- rt: specifies register which will receive result of computation (target register)
- addi, slti, immediate is sign-extended to 32 bits, and treated as a signed integer
- 16 bits \(\Rightarrow\) can be used to represent immediate up to \(2^{16}\) different values

- Key concept: Only one field is inconsistent with R-format. Most importantly, opcode is still in same location
I-Format Example 1

**MIPS Instruction:**

addi $21, $22, -50  // $21 = $22 - 50

- opcode = 8 (look up in table)
- rs = 22 (register containing operand)
- rt = 21 (target register)
- immediate = -50 (by default, this is decimal)

**decimal representation:**

| 8  | 22 | 21 | -50 |

**binary representation:**

001000 10110 10101 11111111111001110
I-Format Example 2

- **MIPS Instruction:**
  
  \[
  \text{lw} \quad \text{t}0,1200(\text{t}1)
  \]

  - opcode = 35 (look up in table)
  - rs = 9 (base register)
  - rt = 8 (destination register)
  - immediate = 1200 (offset)

  **decimal representation:**

  | 35 | 9 | 8 | 1200 |

  **binary representation:**

  \[
  \text{100011 01001 01000 0000010010110000}
  \]
I-Format Problem

What if immediate is too big to fit in immediate field?

- **Load Upper Immediate:**
  
  \[
  \text{lui \ register, \ immediate}
  \]
  
  - puts 16-bit immediate in upper half (high order half) of the specified register, and sets lower half to 0s
    
    \[
    \text{addi \ $t0,\$t0, 0xABABCDCD}
    \]
    
    becomes:
    
    \[
    \text{lui \ $at, 0xABAB}
    \]
    
    \[
    \text{ori \ $at, \$at, 0xCDCD}
    \]
    
    \[
    \text{add \ $t0,\$t0,\$at}
    \]

```
LUI R1
R1 0000 ... 0000
```
## Complete MIPS Instruction Formats

**R-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>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-Format**

<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>

**J-Format**

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</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 Idea: Stored-Program Concept

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

![Diagram](image)

- 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
Big Idea: Stored-Program Concept

- One consequence: everything addressed
  - Everything has a memory address: instructions, data
    - both branches and jumps use these
  - One register keeps address of the instruction being executed: “Program Counter” (PC)
    - Basically a pointer to memory: Intel calls it Instruction Address Pointer, which is better
  - A register can hold any 32-bit value. That value can be a (signed) int, an unsigned int, a pointer (memory address), etc.
Outline

• Instruction set architecture
  (using MIPS ISA as an example)
• Operands
  – Register operands and their organization
  – Memory operands, data transfer, and addressing
  – Immediate operands
• Instruction format
• Operations
  – Arithmetic and logical (Sec 2.5)
  – Decision making and branches
  – Jumps for procedures
## MIPS Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands;</td>
</tr>
<tr>
<td>subtract</td>
<td>sub $1,$2,$3</td>
<td>$1 = $2 - $3</td>
<td>3 operands;</td>
</tr>
<tr>
<td>add immediate</td>
<td>addi $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant;</td>
</tr>
</tbody>
</table>
Bitwise Operations

- Up until now, we’ve done arithmetic (add, sub, addi) and memory access (lw and sw)
- All of these instructions view contents of register as a single quantity (such as a signed or unsigned integer)
- New perspective: View contents of register as 32 bits rather than as a single 32-bit number
- Since registers are composed of 32 bits, we may want to access individual bits rather than the whole.
- Introduce two new classes of instructions:
  - Logical Operators
  - Shift Instructions
MIPS Logical Operations

- Why logical operations
  - Useful to operate on fields of bit or individual bits

<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>
Logical Operators

- **Logical instruction syntax:**
  
  1  2  3  4
  
  or  $t0$, $t1$, $t2$

  1) operation name
  2) register that will receive value
  3) first operand (register)
  4) second operand (register) or immediate (numerical constant)

- **Instruction names:**
  - `and`, `or`: expect the third argument to be a register
  - `andi`, `ori`: expect the third argument to be immediate

- **MIPS Logical Operators are all bitwise, meaning that bit 0 of the output is produced by the respective bit 0’s of the inputs, bit 1 by the bit 1’s, etc.**
Use for Logical Operator And

- and operator can be used to set certain portions of a bit-string to 0s, while leaving the rest alone => mask

- Example:

  Mask: 1011 0110 1010 0100 0011 1101 1001 1010
  0000 0000 0000 0000 0000 1111 1111 1111

  The result of anding these two is:
  0000 0000 0000 0000 0000 1101 1001 1010

- In MIPS assembly: andi $t0,$t0,0xFFF
Use for Logical Operator Or

- or operator can be used to force certain bits of a string to 1s

- For example,

  \[ t0 = 0x12345678, \text{then after} \]

  \[ \text{ori } t0, \ t0, \ 0xFFFF \]

  \[ t0 = 0x1234FFFF \]

  (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are set to 1s)
Shift Instructions (1/3)

- Shift Instruction Syntax:

```
   1   2   3   4
sll   $t2,$s0,4
```

1) operation name
2) register that will receive value
3) first operand (register)
4) shift amount (constant)

- MIPS has three shift instructions:
  - `sll` (shift left logical): shifts left, fills empties with 0s
  - `srl` (shift right logical): shifts right, fills empties with 0s
  - `sra` (shift right arithmetic): shifts right, fills empties by sign extending
Shift Instructions (2/3)

- Move (shift) all the bits in a word to the left or right by a number of bits, filling the emptied bits with 0s.
- Example: shift right by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

- Example: shift left by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0011 0100 0101 0110 0111 1000 0000 0000
Shift Instructions (3/3)

- Example: shift right arithmetic by 8 bits

  0001 0010 0011 0100 0101 0110 0111 1000

- Example: shift right arithmetic by 8 bits

  1001 0010 0011 0100 0101 0110 0111 1000
Suppose we want to get byte 1 (bit 15 to bit 8) of a word in $t0. We can use:

- `sll $t0,$t0,16`
- `srl $t0,$t0,24`
Uses for Shift Instructions (2/2)

● Shift for multiplication: in binary
  – Multiplying by 4 is same as shifting left by 2:
    • \(11_2 \times 100_2 = 1100_2\)
    • \(1010_2 \times 100_2 = 101000_2\)
  – Multiplying by \(2^n\) is same as shifting left by \(n\)

● Since shifting is so much faster than multiplication (you can imagine how complicated multiplication is), a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:
  \(a *= 8;\)  \hspace{1cm} \text{(in C)}

would compile to:
  \text{sll } \$s0,\$s0,3 \hspace{1cm} \text{(in MIPS)}
## MIPS Logical Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and $1,$2,$3</td>
<td>$1 = $2 &amp; $3</td>
<td>3 reg. operands; Logical AND</td>
</tr>
<tr>
<td>or</td>
<td>or $1,$2,$3</td>
<td>$1 = $2</td>
<td>$3</td>
</tr>
<tr>
<td>nor</td>
<td>nor $1,$2,$3</td>
<td>$1 = \sim($2</td>
<td>$3)</td>
</tr>
<tr>
<td>and immediate</td>
<td>andi $1,$2,10</td>
<td>$1 = $2 &amp; 10</td>
<td>Logical AND reg, zero exten.</td>
</tr>
<tr>
<td>or immediate</td>
<td>ori $1,$2,10</td>
<td>$1 = $2</td>
<td>10</td>
</tr>
<tr>
<td>shift left logical</td>
<td>sll $1,$2,10</td>
<td>$1 = $2 &lt;&lt; 10</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srl $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right by constant</td>
</tr>
<tr>
<td>shift right arithm.</td>
<td>sra $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right (sign extend)</td>
</tr>
</tbody>
</table>
So Far...

- All instructions have allowed us to manipulate data.
- So we’ve built a calculator.
- In order to build a computer, we need ability to make decisions...
Outline

• Instruction set architecture
  (using MIPS ISA as an example)

• Operands
  – Register operands and their organization
  – Memory operands, data transfer, and addressing
  – Immediate operands

• Instruction format

• Operations
  – Arithmetic and logical
  – Decision making and branches (Sec. 2.6, 2.9)
  – Jumps for procedures
Decision Making Instructions

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed

- Branch Classifications
  - Two basic types of branches
    - Unconditional: Always jump to the specified address
    - Conditional: Jump to the specified address if some condition is true; otherwise, continue with the next instruction

- Destination addresses can be specified in the same way as other operands (combination of registers, immediate constants, and memory locations), depending on what is supported in the ISA
Addresses in Branches and Jumps

- Instructions:
  - `bne $t4,$t5,Label`  
    Next instruction is at Label if $t4 \neq $t5
  - `beq $t4,$t5,Label`  
    Next instruction is at Label if $t4 = $t5
  - `j Label`  
    Next instruction is at Label

- Formats:
  - **I**
    | op  | rs | rt | 16 bit address |
  - **J**
    | op  | 26 bit address |

- Addresses are not 32 bits
  - How do we handle this with load and store instructions?
Addresses in Branches

• Instructions:
  bne $t4,$t5,Label  Next instruction is at Label if $t4 \neq $t5
  beq $t4,$t5,Label  Next instruction is at Label if $t4 = $t5

• Formats:

| I | op | rs | rt | 16 bit address |

• Could specify a register (like lw and sw) and add it to address
  – use Instruction Address Register (PC = program counter)
  – most branches are local (principle of locality)

• Jump instructions just use high order bits of PC
  – address boundaries of 256 MB
Decision Making: Branches

Decision making: *if* statement, sometimes combined with *goto* and *labels*

beq register1, register2, L1 (beq: Branch if equal)

Go to the statement labeled L1 if the value in register1 equals the value in register2

bne register1, register2, L1 (bne: Branch if not equal)

Go to the statement labeled L1 if the value in register1 does not equal the value in register2

beq and bne are termed Conditional branches

What instruction format is beq and bne?
MIPS Decision Instructions

beq  register1, register2, L1

- Decision instruction in MIPS:
  beq  register1, register2, L1
  “Branch if (registers are) equal”
  meaning:
  if (register1==register2) goto L1

- Complementary MIPS decision instruction
  bne  register1, register2, L1
  “Branch if (registers are) not equal”
  meaning:
  if (register1!=register2) goto L1

- These are called **conditional branches**
MIPS has an unconditional branch:

\[
\text{j label}
\]

- Called a Jump Instruction: jump directly to the given label without testing any condition
- meaning:
  \[
  \text{goto label}
  \]

Technically, it’s the same as:

\[
\text{beq$0,0,\text{label}}
\]
since it always satisfies the condition

It has the j-type instruction format
Conditional Branch Instructions

- `beq` register1, register2, L1 #branch equal
- `bne` register1, register2, L1 #branch if not equal

Ex:

```plaintext
if (i==j) goto L1;
f = g+h;
L1: f = f-i;
```

assume f,g,h,i,j, stored in $s0...$s4

```
beq $s3, $s4, L1
add $s0, $s1, $s2
L1: sub $s0, $s1, $s3
```
Compiling an if-then-else

- Compile by hand
  
  ```c
  if (i == j) f = g + h;
  else f = g - h;
  ```

- Use this mapping:
  
  - f: $s0$
  - g: $s1$
  - h: $s2$
  - i: $s3$
  - j: $s4$

- Final compiled MIPS code:

  ```mips
  beq $s3,$s4,True
  sub $s0,$s1,$s2
  j Fin

  True:
  add $s0,$s1,$s2

  Fin:
  ```

  Note: Compiler automatically creates labels to handle decisions (branches) appropriately.
Inequalities in MIPS

- Until now, we’ve only tested equalities (== and != in C), but general programs need to test < and >
- Set on Less Than:
  ```
  slt   reg1,reg2,reg3
  ```
  meaning:
  ```
  if (reg2 < reg3)
      reg1 = 1;    # set
  else reg1 = 0;   # reset
  ```
- Compile by hand: if (g < h) goto Less;
  Let g: $s0, h: $s1
  ```
  slt $t0,$s0,$s1    # $t0 = 1 if g<h
  bne $t0,$0,Less    # goto Less if $t0!=0
  ```

MIPS has no “branch on less than” => too complex
Immediate in Inequalities

- There is also an immediate version of `slt` to test against constants: `slti`

  ```
  if (g >= 1) goto Loop
  ```

  ```
  Loop: . . .
  ```

  ```
  slti $t0,$s0,1  # $t0 = 1 if $s0 < 1 (g<1)
  beq $t0,$0,Loop # goto Loop if $t0 == 0
  ```

- Unsigned inequality: `sltu, sltiu`

  ```
  $s0 = FFFF FFFA_{hex},  \quad $s1 = 0000 FFFA_{hex}
  ```

  ```
  slt $t0, $s0, $s1  \Rightarrow $t0 = ? 1
  sltu $t1, $s0, $s1  \Rightarrow $t1 = ? 0
  ```
10/16 第二章 小考
Branches: Instruction Format

- Use I-format:

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
</table>

- opcode specifies beq or bne
- rs and rt specify registers to compare

- What can immediate specify? PC-relative addressing
  - Immediate is only 16 bits, but PC is 32-bit
    => immediate cannot specify entire address
  - Loops are generally small: < 50 instructions
    - Though we want to branch to anywhere in memory, a single branch only need to change PC by a small amount
  - How to use PC-relative addressing
    - 16-bit immediate as a signed two’s complement integer to be added to the PC if branch taken
    - Now we can branch +/- $2^{15}$ bytes from the PC?
Branches: Instruction Format

• **Immediate** specifies *word address*
  - Instructions are word aligned (byte address is always a multiple of 4, i.e., it ends with 00 in binary)
    - The number of bytes to add to the PC will always be a multiple of 4
    - Specify the *immediate* in words (confusing?)
    - Now, we can branch +/- $2^{15}$ *words* from the PC (or +/- $2^{17}$ *bytes*),
  
• **Immediate** specifies **PC + 4**
  - Due to hardware, add *immediate* to (PC+4), not to PC
  - If branch not taken: **PC = PC + 4**
  - If branch taken: **PC = (PC+4) + (immediate*4)**
Branch Example

- MIPS Code:
  ```assembly
  Loop:       beq   $9, $0, End
              add   $8, $8, $10
              addi  $9, $9, -1
              j     Loop
  End:        sub   $6, $7, $8
  ```

- Branch is I-Format:
  ```markdown
<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>9</td>
<td>0</td>
<td>???</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
  - Number of instructions to add to (or subtract from) the PC, starting at the instruction *following* the branch
  - => immediate = 3
  ```
Branch Example

• MIPS Code:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq</td>
<td>$9, $0, End</td>
</tr>
<tr>
<td>add</td>
<td>$8, $8, $10</td>
</tr>
<tr>
<td>addi</td>
<td>$9, $9, -1</td>
</tr>
<tr>
<td>j</td>
<td>Loop</td>
</tr>
<tr>
<td>sub</td>
<td>$6, $7, $8</td>
</tr>
</tbody>
</table>

decimal representation:

| 4 | 9 | 0 | 3 |

binary representation:

| 000100 | 01001 | 00000 | 000000000000000011 |
Branch Example 2

- **MIPS Code:**
  
  Label: add $8,$8,$10  
  addi $9,$9,$-1  
  beq $9,$0,Label  
  sub $6,$7,$8

- **Branch is I-Format:**

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
</table>

  - opcode = 4 (look up in table)
  - rs = 9 (first operand)
  - rt = 0 (second operand)
  - immediate = ???
  
  - Number of instructions to add to (or subtract from) the PC, starting at the instruction following the branch
  
  => immediate = -3
Unconditional Branch Instructions and MIPS Control for *if-then-else*

- MIPS unconditional branch instructions:
  
  ```
  j label
  ```

- Example:

  ```
  if (i==j)                bne $s4, $s5, Else
    f=g+h;
  else
    f=g-h;
  ```

  ```
  add $s3, $s4, $s5
  j Lab2
  ```

  ```
  Else: sub $s3, $s4, $s5
  Exit:...
  ```
Unconditional Branch Instructions and MIPS Control for *if-then-else*

- **MIPS unconditional branch instructions:**
  
  \(j\) label

- **Example:**

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

  ```
  beq \$s4, \$s5, Lab1
  add \$s3, \$s4, \$s5
  j Lab2
  Lab1: sub \$s3, \$s4, \$s5
  Lab2: ...
  ```
set-on-less-than in MIPS

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:
  
  ```
  if $s1 < $s2 then
    $t0 = 1
  else
    $t0 = 0
  
  slt $t0, $s1, $s2
  ```

- Can use this instruction with beq/bne to build "blt $s1, $s2, Label"
  - blt => slt + bne/beq
  - can now build general control structures
  - Q. why not “blt” in MIPS?
    - Simplicity

- Note that the assembler needs a register to do this,
  - there are policy of use conventions for registers

- Constant operands are popular in comparisons
  - $zero always has 0
  - Other value: immediate version, slti
  
  ```
  slti $t0, $s2, 10  # $t0 = 1 if $s2 < 10
  ```
MIPS approach for $==$, $!=, <, <=, >, >=$

- Combine \texttt{slt}, \texttt{slti}, \texttt{beq}, \texttt{bne} and \$\texttt{zero}\$ to create all relative conditions
Observation on Branches

- Most conditional branches go a *short and constant distance*
- Fancy addressing modes not often used
- No use for auto-increment/decrement
- So in keeping with the RISC philosophy of *simplicity*, MIPS has only a few basic branch types.
INFO: Complete MIPS Branch Types

- **Conditional branch:**
  - `beq/bne reg1, reg2, addr`
  - If `reg1 =/≠ reg2`, jump to `PC+addr` (PC-relative)

- **Register jump:**
  - `jr reg`
  - Fetch address from specified register, and jump to it

- **Unconditional branch:**
  - `j addr`
  - Always jump to PC: `addr` (use “pseudodirect” addressing)
INFO: Branch Instructions Example

• Conditional branches
  – beq  R1, R2, L1  # if R1 = R2 go to L1
  – bne  R1, R2, L1  # if R1 ≠ R2 go to L1
  – These are I-type instructions

• Unconditional branches
  – jr    R8  # Jump based on register 8

• Test if < 0
  slt    R1, R16, R17  # R1 gets 1 if R16 < R17
  bne    R1, 0, less   # branch to less if R1 =\= 0
由於MIPS是32 bit機器，需要將26 bit的address轉成32 bit，需要從PC暫存器中拿取前面的4bit來湊出32bit
Compiling Other Control Statements

- **Loops:**
  - *for, while*: test before loop body; jump past loop body if false
  - *Do*: test condition at end of loop body; jump to beginning if true

- **switch**: (called “case” statements in some other languages)
  - Build a table of addresses
  - Use jr (or equiv. In non-MIPS processor)
  - Be sure to check for default and unused cases!
while (save[i] == k)
    i += 1;

Assume i, k use register $s3, $s5, base of array “save” is in $s6

    # index $t1 = 4 * i
    Loop: sll $t1, $s3, 2
    # add index to base
    add $t1, $t1, $s6
    # load array value
    lw $t0, 0($t1)
    # test if save[i] == k
    bne $t0, $s5, Exit
    # i = i+1
    add $s3, $s3, 1
    j Loop;

Exit:
Switch Compilation Example

Compile the following:

```c
switch (k) {
    case 0: f = f + 1; break;
    case 1: f = f - 2; break;
    case 3: f = -f; break;
}
```

Note the gap (case 2);

(Assume k in r13)

```
#switch test
slti $14, $13, 0       # set r14 if r13 lt 0
bne $14, $0, Exit      # Go to Exit if k < 0
slti $14, $13, 4       # set r14 if k < 4
beq $14, $0, Exit      # Go to Exit if k = 4
add $14, $13, $13      # r14 = 2*k
add $14, $14, $14      # r14 = 4*k
lw $14, 1000 ($14)      # Base of table at 1000
jr $14                  # Jump to the address table
```

**Switch body**

```
L0: addi $8, $8, 1      # add 1 to r8 (f)
j Exit                  # jump to Exit (break)
L1: subi $8, $8, 2      # subtract 2 from r8
      j Exit              # Another break
L3: sub $8, $0, $8      # f = 0 - f
      j Exit              # Another break
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>address of L0</td>
</tr>
<tr>
<td>1004</td>
<td>address of L1</td>
</tr>
<tr>
<td>1008</td>
<td>address of Exit</td>
</tr>
<tr>
<td>1012</td>
<td>address of L3</td>
</tr>
</tbody>
</table>
INFO: Assembly Language vs. Machine Language

- Assembly provides convenient *symbolic representation*
  - much easier than writing down numbers
  - e.g., destination first

- Machine language is the underlying reality
  - e.g., destination is no longer first

- Assembly can provide 'pseudoinstructions'
  - e.g., “move $t0, $t1” exists only in Assembly
  - would be implemented using “add $t0,$t1,$zero”

- When considering performance you should *count real instructions*
## MIPS Jump, Branch, Compare

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch on equal</td>
<td>beq $1,$2,25</td>
<td>if ($1 == $2) go to PC+4+100</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Equal test; PC relative branch</em></td>
</tr>
<tr>
<td>branch on not eq.</td>
<td>bne $1,$2,25</td>
<td>if ($1!=$2) go to PC+4+100</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Not equal test; PC relative</em></td>
</tr>
<tr>
<td>set on less than</td>
<td>slt $1,$2,$3</td>
<td>if ($2 &lt; $3) $1=1; else $1=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Compare less than; 2’s comp.</em></td>
</tr>
<tr>
<td>set less than imm.</td>
<td>slti $1,$2,100</td>
<td>if ($2 &lt; 100) $1=1; else $1=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Compare &lt; constant; 2’s comp..</em></td>
</tr>
<tr>
<td>jump</td>
<td>j 2500</td>
<td>go to 10000 28-bit+4-bit of PC</td>
</tr>
</tbody>
</table>

這種寫法在組語是不可能存在的，只是為了讓我們知道原來的label所代表的值
So far

MIPS operands

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 registers</td>
<td>$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra</td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. $gp (28) is the global pointer, $sp (29) is the stack pointer, $fp (30) is the frame pointer, and $ra (31) is the return address.</td>
</tr>
<tr>
<td>$2^{30}$ memory words</td>
<td>Memory[0], Memory[4],..., Memory[42949672920]</td>
<td>Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays and spilled register, such as those saved on procedure calls.</td>
</tr>
</tbody>
</table>

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>$s1 = Memory[$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] = $s1</td>
<td>Data from register to memory</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>beq</td>
<td>$s1, $s2, L</td>
<td>if ($s1 == $s2) go to L</td>
<td>Equal test and branch</td>
</tr>
<tr>
<td></td>
<td>bne</td>
<td>$s1, $s2, L</td>
<td>if ($s1 != $s2) go to L</td>
<td>Not equal test and branch</td>
</tr>
<tr>
<td></td>
<td>slt</td>
<td>$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>j</td>
<td>j 2500</td>
<td>go to 10000</td>
<td>jump to target address</td>
</tr>
<tr>
<td></td>
<td>jr</td>
<td>jr $sra</td>
<td>go to $sra</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td></td>
<td>jal</td>
<td>jal 2500</td>
<td>$sra = PC + 4; go to 1000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>
Outline

• Instruction set architecture
  (using MIPS ISA as an example)

• Operands
  – Register operands and their organization
  – Immediate operands
  – Memory operands, data transfer, and addressing

• Instruction format

• Operations
  – Arithmetic and logical
  – Decision making and branches
  – Jumps for procedures (Sec. 2.7)
J-Format Instructions (1/3)

- For branches, we assumed that we won’t want to branch too far, so we can specify change in PC.
- For general jumps (j and jal), we may jump to anywhere in memory.
- Ideally, we could specify a 32-bit memory address to jump to.
- Unfortunately, we can’t fit both a 6-bit opcode and a 32-bit address into a single 32-bit word, so we compromise.
J-Format Instructions (2/3)

- Define “fields” of the following number of bits each:

<table>
<thead>
<tr>
<th>6 bits</th>
<th>26 bits</th>
</tr>
</thead>
</table>

- As usual, each field has a name:

<table>
<thead>
<tr>
<th>opcode</th>
<th>target address</th>
</tr>
</thead>
</table>

- Key concepts:
  - Keep opcode field identical to R-format and I-format for consistency
  - Combine other fields to make room for target address

- Optimization:
  - Jumps only jump to word aligned addresses
    - last two bits are always 00 (in binary)
    - specify 28 bits of the 32-bit bit address
Where do we get the other 4 bits?
- Take *the 4 highest order bits from the PC*
- Technically, this means that *we cannot jump to anywhere in memory*, but it’s adequate 99.9999…% of the time, since programs aren’t that long
- Linker and loader avoid placing a program across an address boundary of 256 MB

**Summary:**
- New PC = \( PC[31..28] \| \text{ target address (26 bits) } \| \text{ 00} \)
- Note: means concatenation
  - 4 bits || 26 bits || 2 bits = 32-bit address

If we absolutely need to specify a 32-bit address:
- Use \( jr \ $ra \quad \# \text{ jump to the address specified by } $ra \)
Procedures

• Six steps in the execution of a procedure
  – Place parameters in a place where the procedure can access them
  – Transfer control to the procedure
  – Acquire the storage resources needed for the procedure (local variables)
  – Perform the desired task
  – Place the result value in a place where the calling program can access it
  – Return control to the point of origin
Function Calls in the MIPS

- Function calls an essential feature of programming languages
  - The program calls a function to perform some task
  - When the function is done, the CPU continues where it left off in the calling program
- But how do we know where we left off?
Procedures

• Procedure/Subroutine

  A set of instructions stored in memory which perform a set of operations based on the values of parameters passed to it and returns one or more values

• Steps for execution of a procedure or subroutine

  ➢ The program (caller) places parameters in places where the procedure (callee) can access them
  ➢ The program transfers control to the procedure
  ➢ The procedure gets storage needed to carry out the task
  ➢ The procedure carries out the task, generating values
  ➢ The procedure (callee) places values in places where the program (caller) can access them
  ➢ The procedure transfers control to the program (caller)
Procedures

- int f1 (inti, intj, intk, intg)
  
  { 
  return 1; 
  }

- int f2 (ints1, ints2)

  { 
  add $3,$4, $3 
  i = f1 (3,4,5, 6); 
  add $2, $3, $3 
  }

- How to pass parameters & results?
- How to preserve caller register values?
- How to alter control? (i.e., go to callee, return from callee)
MIPS Procedures

- How to pass parameters & results
  - $a0-$a3: four argument registers. What if # of parameters is larger than 4? – push to the stack
  - $v0-$v1: two value registers in which to return values

- How to preserve caller register values?
  - Caller saved register
  - Callee saved register
  - Use stack

- How to switch control?
  - How to go to the callee
    - jal procedure_address(jump and link)
      - Store the the return address (PC+4 ) at $ra
      - set PC = procedure_address
  - How to return from the callee
    - Callee executes jr $ra
Calling a Function in the MIPS

- Use the *jal* (“jump and link”) instruction

  - *jal addr just j addr except*
    - The “*return address*” $(PC) + 4$ placed in $ra$ (R31)
    - This is the address of the *next instruction* after the jal
    - Use *jr $ra* to return
Instructions Supporting Procedure Calls

- **Parameter passing**
  - $a0 ~ a3$ are used for these
  - Q. what if parameters exceed four?
  - Spilling registers, place parameters in stack, $sp (R29)$

- **Transfer control: Jump and link**
  - `jal` procedure address
  - note: return address is stored in $ra (R31)$

- **Return value**
  - $v0 ~ v1$ for return values
  - Q. What if returns results exceed two?
  - Saving return address on stack
    - $sp (R29)$ is used as stack pointer

- **Return**
  - `jr $ra`
Procedure Call Example

```c
int leaf_example (int g, int h, int i, int j)
{
    int f;
    f = (g+h) - (i+j);
    return f;
}
```
Assume g, h, i, j use $a0,..$a3, f uses $s0
Refer p. 11

leaf_example:

#push old values into stack to avoid damage
  addi $sp, $sp, -12;
  sw $t1, 8($sp)
  sw $t0, 4($sp)
  sw $s0, 0($sp)

#functional body
  add $t0, $s1, $s2
  add $t1, $s3, $s4
  sub $s0, $t0, $t1

#return value, copy f to return registers
  add $v0, $s0, $zero

#pop old values from stack
  lw $s0, 0($sp)
  lw $t0, 4($sp)
  lw $t1, 8($sp)
  addi $sp, $sp, 12
  jr $ra
Improve the Example

● Problem in previous example
  – A lot of saving and restoring temporary registers

● How to avoid it in MIPS registers convention
  – Temporary registers, $t0..$t9
    ● Value won’t be preserved in the procedure call
  – Saved registers, $s0..$s7
    ● Value must be preserved
    ● If used, these must be saved and stored
Difficulties with Function Calls

● This example works OK. But what if:
  – The function F calls another function?
  – The caller had something important in regs R6 and/or R7?
  – The called function calls itself, (nested procedure)?
    ● Register conflict

● Solution
  – Each version of a function should have its own copies of variables
  – These are arranged in a stack, as a pile of frames.
Procedure Call Stack (Frame)

Frame pointer points to the first word of the procedure frame
Procedure Call Stack (Frame)

Before the procedure call

during the procedure call

after the procedure call
Nested Procedures

- Problems:
  - Register conflicts

- Solutions:
  - Push all the other register that must be preserved onto the stack
  - Procedure
    - The caller pushed any argument register $a0-\text{a}3$ or temporary registers $t0-\text{t}9$ that are needed after the call
    - The callee push the return address $\text{ra}$ and any saved registers $s0-\text{s}7$ used by the callee
    - Stack push and store
Stack Examples for Nested Functional Calls

- Assume function A calls B, which calls C. Function C calls itself once:
Examples for Nested Functional Calls

```c
int factorial (int n)
{
    if(n < 1) return 1;
    else return (n * factorial (n - 1));
}
```
Parameter n => $a0
factorial:
#push old values into stack to avoid damage
    addi $sp, $sp, -8;
sw $ra, 4($sp)
sw $a0, 0($sp)
#functional body
    slti $t0, $a0, 1           # test if n < 1
    beq  $t0, $zero, L1       # if n >= 1, go to L1
#return 1
    addi $v0, $zero, 1        # return 1
addi $sp, $sp, 8            #pop 2 items off stack
jr    $ra
#another return
L1:
  addi $a0, $a0, -1  # N >= 1, new factorial (n-1)
  jal  factorial
#pop values to restore
  lw  $a0, 0($sp)
  lw  $ra, 4($sp)
  addi $sp, $sp, 8
  mul  $v0, $a0, $v0  #return n * factorial(n-1)
  jr  $ra
INFO: Parameter Passing

- **Stack**
  - Ideal data structure for spilling registers

- **Caller save.** The calling procedure (caller) is responsible for saving and restoring any *registers* that must be *preserved across the call*. The called procedure (callee) can then modify any register without constraint.

- **Callee save.** The callee is responsible for saving and restoring any *registers that it might use*. The caller uses registers without worrying about restoring them after a call.
Stack Frames

- If a function needs more memory and/or may call others, it uses a stack frame, which holds:
  - *Automatic* variables (non-static variables declared within function)
  - *Arguments* to the function (just another type of local variable)
  - The “*return address*” (since $ra overwritten by call)
  - *Saved* registers from caller ($s0-$s7) if you need to use them
  - “*Spill*” registers, including $t0-$t9 when calling others
Layout of a Stack Frame

$fp \rightarrow$

- Argument 5
- Argument 6
- \ldots

Saved registers

Local variables

$sp \rightarrow$

Higher memory addresses

Stack grows

Lower memory addresses
Allocating Space for New Data on the Stack  
Details of Stack for Procedure Calls (1)

Procedure frame  
the segment of stack containing a procedure’s saved registers and local variables
Details of Stack for Procedure Calls (2)

- **Calling a Non-Leaf Function (Caller)**
  - Put arguments to the function in $a0$-$a3$
  - Save contents of $t0$-$9$ if they will be needed later
  - If more than 4 args, push them onto stack
  - jal to beginning of the function code
Details of Stack for Procedure Calls (3)

- Calling a Non-Leaf Function (Callee)
  - Push current $fp$ onto stack
  - Move $fp$ to top of frame (just below old sp)
  - Set $sp$ to ($fp$ – frame size)
    - Frame size is the same for every call of the same function
    - Known at compile-time
  - Use displacement addressing to get at local variables
  - Save $s0$-$s7$ (whichever you need to reuse) and $ra$ in frame
  - Save $a0$-$a3$ to frame if needed (e.g., calling another function)
Details of Stack for Procedure Calls (4)

● Returning from Non-Leaf Function (Callee)
  – Put return values (if any) in $v0 and $v1
  – Restore $s0-$s7 (whichever were saved) and $ra from frame
  – Restore sp to just above current fp
  – Restore old fp from stack frame
  – Jump to $ra (jr)
  – Caller can get return args in $v0 and $v1, if any
# Register Conventions in the MIPS

<table>
<thead>
<tr>
<th>Names</th>
<th>Regs</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>Constant 0</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>(Reserved for assembler)</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>Return values (NOT Preserved across the calls)</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>Args to functions (NOT Preserved across the calls)</td>
</tr>
<tr>
<td>$t0-$t9</td>
<td>8-15, 24-25</td>
<td>Temporaries (NOT Preserved across the calls)</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>Saved values (Preserved across the calls)</td>
</tr>
<tr>
<td>-</td>
<td>26-27</td>
<td>(Reserved for OS kernel)</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>Global pointer to global data</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>Stack pointer (Preserved across the calls)</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>Frame pointer (Preserved across the calls)</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>Return address (Preserved across the calls)</td>
</tr>
</tbody>
</table>
Other Storage: Global Variables

- In C/C++, “global variables” are
  - Variables declared outside of any functions
  - Static variables (inside or outside a function)
  - Static data members of a class (C++)

- Properties:
  - Only one copy of each (unlike automatic variables)
  - Initialization allowed (set value before main () starts)
  - All in one region of memory, accessed through $gp (r28)$
Other Storage: Dynamic Storage (Heap)

- In C/C++, the “heap” contains
  - Blocks of memory allocated by `malloc()` etc.
  - Objects created using the new keyword (C++)
- Properties:
  - Stored in a big chunk of memory between globals and stack
  - Controlled by the programming language’s library (e.g., libc)
  - Can be grown if needed
  - No dedicated reg. Like $gp; everything goes through pointers
Typical Layout of Program

$sp \rightarrow 7ff dfdf$

hex

$gp \rightarrow 1000 8000$

1000 8000

pc \rightarrow 0040 0000

stack

Dynamic data

Static data

Text

Reserved
What an Executable Program Looks Like

• When you execute a program, it is in the form of an “executable”
• The executable contains everything you need to run your program
  – Every function used, starting with main() – the “text segment”
  – Values of all initialized global variables – the “data segment”
  – Information about uninitialized globals
• Every function and every global variable has an absolute address in memory
Executing an Executable

● When you execute a program, the loader:
  – Allocates space for your program (details vary by OS)
  – Copies the text and data segments of the executable to memory
  – Jumps to a known starting address (specified in the executable)

● Once the executable starts running at that starting address, it
  – Initializes regs such as $gp and $sp; initializes heap (if used)
  – Sets uninitialized globals to 0 (if the language requires this)
  – Sets up command line args into data structure (e.g., argc/argv)
  – Does jal to start of main () function
### So far

#### MIPS operands

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 registers</td>
<td>$s0$-$s7$, $t0$-$t9$, $zero$, $a0$-$a3$, $v0$-$v1$, $gp$, $fp$, $sp$, $ra$</td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero$ always equals 0. $gp$ (28) is the global pointer, $sp$ (29) is the stack pointer, $fp$ (30) is the frame pointer, and $ra$ (31) is the return address.</td>
</tr>
<tr>
<td>$2^{30}$ memory words</td>
<td>Memory [0], Memory [4], …, Memory[42949672920]</td>
<td>Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled register, such as those saved on procedure calls.</td>
</tr>
</tbody>
</table>

### 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>lw</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>sw</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>beq</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>bne</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>slt</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>j</td>
<td>j 2500</td>
<td>go to 10000</td>
<td>jump to target address</td>
</tr>
<tr>
<td></td>
<td>jr</td>
<td>jr $ra$</td>
<td>go to $ra$</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td></td>
<td>jal</td>
<td>jal 2500</td>
<td>$ra = PC + 4$; 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
Outline

• Instruction set architecture
  (using MIPS ISA as an example)

• Operands
  – Register operands and their organization
  – Immediate operands
  – Memory operands, data transfer, and addressing

• Instruction format

• Operations
  – Arithmetic and logical
  – Decision making and branches
  – Jumps for procedures
  – Communicating with People (Sec. 2.8)
Communicating with People

- For communication
  - Use characters and strings

- Characters
  - 8-bit (one byte) data for ASCII
    
    ```
    lb $t0, 0($sp) ; load byte
    ```
    - Load a byte from memory, placing it in the rightmost 8-bits of registers

    ```
    sb $t0, 0($gp) ; store byte
    ```
    - Takes a byte from the rightmost 8-bits of a register and writes it to the memory

  - Unicode in Java (16-bits)
    
    ```
    lh $t0, 0($sp) ; load halfword
    ```
    - Load a byte from memory, placing it in the rightmost 16-bits of registers

    ```
    sh $t0, 0($gp) ; store halfword
    ```
    - Takes a byte from the rightmost 16-bits of a register and writes it to the memory
Q. Impact of Word Alignment to Byte/Halfword Storage

• MIPS software tries to keep the stack aligned to word address
  – A char variable will occupy four bytes, even though it requires less
  – Solution
    • Software will pack C string in 4 bytes per word, Java string in 2 halfwords per word
Outline

- Instruction set architecture (using MIPS ISA as an example)
- Operands
  - Register operands and their organization
  - Immediate operands
  - Memory operands, data transfer, and addressing
- Instruction format
- Operations
  - Arithmetic and logical
  - Decision making and branches
  - Jumps for procedures
  - Communicating with People
  - MIPS Addressing for 32-Bit Immediates and Addresses (2.9)
MIPS Addressing Mode

• Addressing mode
  – A method that help you identify and find where the operand is
  – What you learned now
    ● Register addressing
    ● Immediate addressing
    ● Base or displacement addressing

\[
\text{lw } \$t0, 32(\$s3)
\]
Review: Handle 32-bit Constants in MIPS

• We'd like to be able to load a 32 bit constant into a register

• Must use *two instructions*, new "load upper immediate" instruction

\[
lui \ $t0, \ 1010101010101010 \quad \text{filled with zeros}
\]

\[
\begin{array}{c|c}
1010101010101010 & 0000000000000000 \\
\end{array}
\]

• Then must get the lower order bits right, i.e.,

\[
ori \ $t0, \ $t0, \ 1010101010101010 \quad \text{filled with zeros}
\]

\[
\begin{array}{c|c|c}
1010101010101010 & 0000000000000000 & 0000000000000000 \\
0000000000000000 & 1010101010101010 \\
\end{array}
\]

Either compiler or assembler to break and then reassemble this
So $at is reserved for assembler
Addresses in Branches and Jumps

**Instructions:**

- **I**
  
  \[
  \text{bne \ $s0,\$s1, Exit}
  \]

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit address</th>
</tr>
</thead>
</table>

- **J**
  
  \[
  \text{j 1000}
  \]

<table>
<thead>
<tr>
<th>op</th>
<th>26 bit address</th>
</tr>
</thead>
</table>

**Q.** What’s the destination address of next instruction? And How far do you can jump (or branch)?
Addresses in Branches and Jumps

• Destination Address
  – MIPS uses *PC-relative address (relative to PC+4, +/- 2^{15})* for all conditional branches
    Next PC = (PC +4) + (16-bit address <<2)
  – MIPS uses *long addresses (26-bits) (pseduodirect addressing)* for both jump and jump-and-link instructions
    Next PC =\{PC[31:28], (26-bit address <<2)\}
  – Note. PC-relative addressing refer to the number of words to the next instruction instead of number of bytes (*word address*)
  – 16-bit field => 18-bit byte address displacement
  – 26-bit field => 28-bit byte address displacement
How Far Do You Can Jump or Branch?

• Formats:

<table>
<thead>
<tr>
<th>I</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit address</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>op</td>
<td></td>
<td></td>
<td>26 bit address</td>
</tr>
</tbody>
</table>

• Branch limitation: +/-2^{15}, (2^{18} = 256KB address boundaries)
  – Is it enough: most branches are local (principle of locality)
  – How about larger space? Branch + Jump
    beq $s0, $s1, L1
    bne $s0, $s1, L2
    j L1
    L2: ..........

• Jump limitation: +/-2^{25}, (2^{28} = 256MB address boundaries)
  – How about larger space? Jump registers (32-bit value)
    • jr $s0
Addressing in Branches and Jumps

- **J-type**

  | 6 bits | 26 bits |

- **I-type**

  | 6 bits | 5 bits | 5 bits | 16 bits |

  - Program counter = Register + Branch address
    - PC-relative addressing
      - We can branch within $\pm 2^{15}$ words of the current instruction.
  - Conditional branches are found in loops and in if statements, so they tend to branch to a nearby instruction.
J-type

- 26-bit field is sufficient to represent 32-bit address?
  - PC is 32 bits
    - The lower 28 bits of the PC come from the 26-bit field
      - The field is a word address
      - It represents a 28-bit byte address
    - The higher 4 bits
      - Come from the original PC content

- An address boundary of 256 MB (64 million instructions)
# Addressing Modes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>addi R4,R4,3</td>
<td>R4 ← R4+3</td>
</tr>
<tr>
<td>Register</td>
<td>add R4,R4,R3</td>
<td>R4 ← R4+R3</td>
</tr>
<tr>
<td>Base/Displacement</td>
<td>lw R4,100(R1)</td>
<td>R4 ← Mem[100+R1]</td>
</tr>
<tr>
<td>PC-relative</td>
<td>beq R1, R2, L1</td>
<td></td>
</tr>
<tr>
<td>Pseudodirect</td>
<td>j L2</td>
<td></td>
</tr>
</tbody>
</table>
MIPS Addressing Mode

1. Immediate addressing
   - op | rs | rt | Immediate

2. Register addressing
   - op | rs | rt | rd | ... | funct
   - Registers
     - Register

3. Base addressing
   - op | rs | rt | Address
   - Register
   - Memory
     - Byte
     - Halfword
     - Word

4. PC-relative addressing
   - op | rs | rt | Address
   - PC
   - Memory
     - Word

5. Pseudodirect addressing
   - op
     - Address
   - PC
   - Memory
     - Word
# MIPS Operands

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 registers</td>
<td>$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at</td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.</td>
</tr>
<tr>
<td>2^30 memory words</td>
<td>Memory[0], Memory[4], ..., Memory[4294967292]</td>
<td>Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.</td>
</tr>
</tbody>
</table>

# 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><strong>Arithmetic</strong></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></td>
<td>add immediate</td>
<td>addi $s1, $s2, 100</td>
<td>$s1 = $s2 + 100</td>
<td>Used to add constants</td>
</tr>
<tr>
<td><strong>Data transfer</strong></td>
<td>load word</td>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Word 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>Word from register to memory</td>
</tr>
<tr>
<td></td>
<td>load byte</td>
<td>lb $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Byte from memory to register</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>sb $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Byte from register to memory</td>
</tr>
<tr>
<td></td>
<td>load upper immediate</td>
<td>lui $s1, 100</td>
<td>$s1 = 100 * $s2^16</td>
<td>Loads constant in upper 16 bits</td>
</tr>
<tr>
<td><strong>Conditional branch</strong></td>
<td>branch on equal</td>
<td>beq $s1, $s2, 25</td>
<td>if($s1 == $s2) go to PC + 4 + 100</td>
<td>Equal test; PC-relative branch</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>bne $s1, $s2, 25</td>
<td>if($s1 != $s2) go to PC + 4 + 100</td>
<td>Not equal test; PC-relative</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></td>
<td>set less than immediate</td>
<td>slti $s1, $s2, 100</td>
<td>if($s2 &lt; 100) $s1 = 1; else $s1 = 0</td>
<td>Compare less than constant</td>
</tr>
<tr>
<td><strong>Unconditional jump</strong></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 $ra</td>
<td>go to $ra</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>jal 2500</td>
<td>$ra = PC + 4; go to 10000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>
Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

<table>
<thead>
<tr>
<th>R</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td>16 bit address</td>
</tr>
<tr>
<td>J</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26 bit address</td>
</tr>
</tbody>
</table>

- rely on compiler to achieve performance
  - what are the compiler's goals?
- help compiler where we can
2.10 Translating and Starting a Program
Starting A Program

Compiler

Transforms the C program into an assembly language program.

Assembler

Assembly language program

Object: Machine language module

Object: Library routine (machine language)

Linker

Executable: Machine language program

Loader

Memory
IA-32 instruction Formats

- a. **JE EIP + displacement**
  
  ```
<table>
<thead>
<tr>
<th>JE</th>
<th>Condition</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
  ```

- b. **CALL**
  
  ```
<table>
<thead>
<tr>
<th>CALL</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
  ```

- c. **MOV EBX, [EDI + 45]**
  
  ```
<table>
<thead>
<tr>
<th>MOV</th>
<th>d w</th>
<th>n/m</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 1 1</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
  ```

- d. **PUSH ESI**
  
  ```
<table>
<thead>
<tr>
<th>PUSH</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 3</td>
<td></td>
</tr>
</tbody>
</table>
  ```

- e. **ADD EAX, #6765**
  
  ```
<table>
<thead>
<tr>
<th>ADD</th>
<th>Reg w</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 3 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
  ```

- f. **TEST EDX, #42**
  
  ```
<table>
<thead>
<tr>
<th>TEST</th>
<th>w</th>
<th>Postbyte</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 1 8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
  ```

**IA-32 variable-length encoding vs. MIPS fixed-length encoding**
Summary: MIPS ISA

- 32-bit fixed format instructions (3 formats)
- 32 32-bit GPR (R0 = zero), 32 FP registers, (and HI LO)
  - partitioned by software convention
- 3-address, reg-reg arithmetic instructions
- Memory is byte-addressable with a single addressing mode: base+displacement
  - 16-bit immediate plus LUI
- Decision making with conditional branches: beq, bne
  - Often compare against zero or two registers for =
  - To help decisions with inequalities, use: “Set on Less Than” called slt, slti, sltu, sltui
- Jump and link puts return address PC+4 into link register $ra (R31)
- Branches and Jumps were optimized to address to words, for greater branch distance
Summary: MIPS ISA

- Immediates are extended as follows:
  - logical immediate: zero-extended to 32 bits
  - arithmetic immediate: sign-extended to 32 bits
  - Data loaded by lb and lh are similarly extended:
    lbu, lhu are zero extended; lb, lh are sign extended
- Simplifying MIPS: Define instructions to be same size as data (one word), so they can use same memory
- Stored Program Concept: Both data and actual code (instructions) are stored in the same memory
- Instructions formats are kept as similar as possible

<table>
<thead>
<tr>
<th></th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate

- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast

- Instruction set architecture
  - a very important abstraction indeed!