Instructions

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

Design goals: maximize performance and minimize cost, reduce design time

MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: \[ A = B + C \]
MIPS code: \[ add \, \$s0, \, \$s1, \, \$s2 \]

(associated with variables by compiler)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s0$</td>
<td>$s1$</td>
<td>$s2$</td>
</tr>
</tbody>
</table>
MIPS arithmetic

- Design Principle 1: **Simplicity favors regularity.**
  - In MIPS, all instructions have 3 operands.
  - In MIPS, operand order is fixed (destination first).
    - “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”

  C code:  
  ```c
  A = B + C + D;
  E = F - A;
  ```
  
  MIPS code:  
  ```
  add $t0, $s1, $s2 ; $t0 ← B + C
  add $s0, $t0, $s3 ; A ← $t0 + D
  sub $s4, $s5, $s0 ; E ← F - A
  ```

- In MIPS, operands must be registers, and only 32 (32-bit-wide) registers are provided.
  - Register: A limited number of special quick-access locations inside processor
  - In the previous example, $s0, $s1, $s2, ... are all registers, which are associated with the variables by compiler.

- Design Principle 2: **Smaller is faster.**
  Why? A very large number of registers would increase the clock cycle time simply because it takes electrical signals longer when they must travel further.
MIPS arithmetic

- What if a program consists of variables more than 32, such as an array.
- Answer: They will be placed in main memory (DRAM).

MIPS data transfer (between registers and memory)

- Since arithmetic instructions can only operate on registers; a machine language must include instructions to transfer data between memory and registers.
Memory organization

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

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>Address</th>
<th>Data (content)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>1</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>2</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>3</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>4</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>5</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>6</td>
<td>8 bits of data</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Memory organization

- Bytes are nice, but gradually, demands for wider data(bus) emerges.
  - The Intel 4004 chip used a 4-bit wide memory organization, named nibble. Later, Intel 8008 (1971) began to use a 8-bit wide memory organization, named byte.
  - For MIPS, a word is 32 bits or 4 bytes (double-word = 64 bits)
  - The address index is still byte-based; hence, indices increases as 0, 4, 8, 12, etc.

<table>
<thead>
<tr>
<th>Address</th>
<th>Data (content)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>4</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>8</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>12</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

For Microsoft-Intel-based machine,

- bit = binary digit with a value of “1” or “0”
- nibble = 4-bit wide memory location
- byte = 8-bit wide binary number
- word = 16-bit wide binary number
- double-word = 32-bit wide binary number
Memory organization

- $2^{32}$ bytes with byte addresses from 0 to $2^{32} - 1$
- $2^{30}$ words with byte addresses 0, 4, 8, ... $2^{32} - 4$
- Words must be aligned (named alignment restriction for MIPS)

**Question:** what are the least 2 significant bits of a word address?

- **Answer:** $(00)_2$

Addresses, as well as data, are often represented in hexadecimal format.

**Hexadecimal:** 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

<table>
<thead>
<tr>
<th>Address</th>
<th>Data (content)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>4</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>8</td>
<td>32 bits of data</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^{32} - 8$</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>$2^{32} - 4$</td>
<td>32 bits of data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Data (content)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000H</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>00000004H</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>00000008H</td>
<td>32 bits of data</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>FFFFFFF8H</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>FFFFFFFEH</td>
<td>32 bits of data</td>
</tr>
</tbody>
</table>
Byte order

- Little-endian
  - E.g. Intel x86 family, DEC VAX, DEC Alpha

  ![Little-endian example]

- Big-endian
  - E.g. IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

MIPS data transfer

- Load and store instructions
- Example:


  MIPS code: `lw $t0, 32($s3); $t0=Memory[$s3+32]`  
  `add $t0, $s2, $t0; $t0=$s2+$t0`  
  `sw $t0, 32($s3); Memory[$s3+32]=$t0`

<table>
<thead>
<tr>
<th>Temporary usage</th>
<th>h</th>
<th>Base address for array A[ ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s0</td>
<td>$s2</td>
<td>$s3</td>
</tr>
</tbody>
</table>

The constant 32 is referred to as offset or displacement.

- `lw = load word;`
- `sw = store word;`
So far we’ve learned

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

- **Instruction**

<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>

Register names for MIPS

- Can refer to registers by name (e.g., $s2, $t2) instead of number (e.g., $18, $10)
- **Store word has destination last**
- Remember arithmetic operands are registers, not memory! Can’t write:
  
  ```
  add 48($s3), $s2, 2($s3)
  ```
Consideration of small constants

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

- Solutions? Design Principle 3: Make the common case fast.
  - Put 'typical constants' in memory and load them.
    - Time consuming.
  - Create hard-wired registers (like $zero) for constants like one.
    - Limited numbers of constants can be provided.
  - Create specific instruction for operations of frequently used constants.
    E.g.,
    \[ \text{addi } $s3, $s3, 4 \]
    
    \[ \text{addi } = \text{add immediate} \]

Machine language

- MIPS instructions, like registers and words of data, are also 32 bits long
  - Example: add $t0, $s1, $s2
  - Registers have numbers, $t0=$8, $s1=$17, $s2=$18

- Instruction Format: consists of 6 fields
  
  \[
  \begin{array}{cccccccc}
  000000 & 10001 & 10100 & 01000 & 00000 & 100000 \\
  \text{op} & \text{rs} & \text{rt} & \text{rd} & \text{shamt} & \text{func} \\
  \end{array}
  \]

  - \text{op} = \text{opcode, } \text{rs} = \text{first source register, } \text{rt} = \text{second source register}
  - \text{rd} = \text{destination register, } \text{shamt} = \text{shift amount, } \text{func} = \text{function}

  - \text{The numerical version of instructions = machine language}
  - \text{A sequence of numerical version of instructions = machine code}
  - \text{The mnemonics of machine language such as add = assembly language.}
Machine language

- Consider the load-word and store-word instructions,
  - A problem occurs when an instruction needs longer fields than those shown above
    \[
    \text{lw } \text{t0}, \ 32(\text{s3})
    \]
    - Apparently, 5 or 6-bit field for offsets will be too small. Often, 12~16 bits are required. (See the statistics on later slides.)
  - Design principle 4: **Good design demands good compromise** (A compromise against design principle 1: Simplicity favors regularity)
    - Two options to solve the small-field problem.
      - To have variable-length instructions
      - To have fixed-length instructions but different formats.

Graphical interpretation of instruction format width

| Variable: | \[\]
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
</tr>
<tr>
<td>[]</td>
</tr>
<tr>
<td>[]</td>
</tr>
<tr>
<td>[]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fixed:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hybrid:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
</tr>
<tr>
<td>[]</td>
</tr>
<tr>
<td>[]</td>
</tr>
</tbody>
</table>
**Instruction set format**

- **General decision rules on instruction width**
  - If code size is most important, use variable-length instructions.
  - If speed is most important, use fixed-length instructions.

---

**Machine language**

- **Introduce a new type of instruction format**
  - I-type for data transfer instructions
  - R-type for arithmetic instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Type</th>
<th>op (6 bits)</th>
<th>rs (5 bits)</th>
<th>rt (5 bits)</th>
<th>rd (5 bits)</th>
<th>shamt (5 bits)</th>
<th>funct (6 bits)</th>
<th>address (16 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td></td>
<td>0</td>
<td>32</td>
<td>n.a.</td>
</tr>
<tr>
<td>sub</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td></td>
<td>0</td>
<td>34</td>
<td>n.a.</td>
</tr>
<tr>
<td>lw</td>
<td>I</td>
<td>35</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
</tr>
<tr>
<td>sw</td>
<td>I</td>
<td>43</td>
<td>reg</td>
<td>reg</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>address</td>
</tr>
</tbody>
</table>

Hence, the offset for MIPS machine ranges 64K in size.

For I-type, rs = base-address register, rt = destination (target) register, and address = offset.
Machine language

- Example: `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>8</td>
<td>32</td>
</tr>
</tbody>
</table>

- Please self-practice the example on page 65.

Small note on instruction design

- Stack-based
  
  Add
  
  `tos = tos + next`

- Accumulator-based (1 register)
  
  Add A
  
  `acc = acc + Mem[A]`
  
  * The single register is called the **accumulator**, because all operations accumulate in this single register.

- Register-memory-based
  
  Add $r1, A
  
  `$r1 = r1 + Mem[A]`

- Load-store-based
  
  Add $r0, $r1, $r2
  
  `$r0 = r1 + r2`

Comparison among these four design philosophies are based on **Bytes per instruction?** **Number of Instructions?** **Cycles per instruction?**
Small note on instruction design

Comparing Number of Instructions

Code sequence for $C = A + B$ for four classes of instruction sets:
(A, B, C represents memories below.)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register (register-memory)</th>
<th>Register (load-store or register-register)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Load R1,A</td>
<td>Load R1,A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R1,B</td>
<td>Load R2,B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store C, R1</td>
<td>Add R3,R1,R2</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td>Store C,R3</td>
<td></td>
</tr>
</tbody>
</table>

If the design allows all the registers to be used for any purpose/operation, it is named “general-purpose-register” instruction set design.

Graduate dominance of general purpose register instruction set design

- In 1975-1995, all machines use general-purpose register design.
  - Note: Intel 8086 is called extended accumulator, dedicated register, or special-purpose register design, since registers are designed specifically for certain instructions. After Intel 80386, the dedication of registers to specific instructions is greatly removed; hence, Intel 80386 becomes more like a general purpose register instruction set design.

- Advantages of registers
  - Registers are faster than memory.
  - Registers are easier for a compiler/assembler to use.
  - Memory traffic is reduced so the program is sped up.
  - Code density is reduced since registers are named with fewer bits than memory locations.
### History of instruction set design

<table>
<thead>
<tr>
<th>Machine</th>
<th>Number of general-purpose registers</th>
<th>Architectural style</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDC 6600</td>
<td>8</td>
<td>Load-store</td>
<td>1963</td>
</tr>
<tr>
<td>IBM 701</td>
<td>1</td>
<td>Accumulator</td>
<td>1953</td>
</tr>
<tr>
<td>IBM 360</td>
<td>16</td>
<td>Register-memory</td>
<td>1964</td>
</tr>
<tr>
<td>DEC PDP-8</td>
<td>1</td>
<td>Accumulator</td>
<td>1965</td>
</tr>
<tr>
<td>DEC PDP-11</td>
<td>8</td>
<td>Register-memory</td>
<td>1970</td>
</tr>
<tr>
<td>Intel 8008</td>
<td>1</td>
<td>Accumulator</td>
<td>1972</td>
</tr>
<tr>
<td>Motorola 6800</td>
<td>2</td>
<td>Accumulator</td>
<td>1974</td>
</tr>
<tr>
<td>DEC VAX</td>
<td>16</td>
<td>Register-memory, memory-memory</td>
<td>1977</td>
</tr>
<tr>
<td>Intel 8086</td>
<td>1</td>
<td>Extended accumulator</td>
<td>1978</td>
</tr>
<tr>
<td>Motorola 68000</td>
<td>16</td>
<td>Register-memory</td>
<td>1980</td>
</tr>
<tr>
<td>Intel 80386</td>
<td>8</td>
<td>Register-memory</td>
<td>1985</td>
</tr>
<tr>
<td>MIPS</td>
<td>32</td>
<td>Load-store</td>
<td>1985</td>
</tr>
<tr>
<td>HP PA-RISC</td>
<td>32</td>
<td>Load-store</td>
<td>1986</td>
</tr>
<tr>
<td>SPARC</td>
<td>32</td>
<td>Load-store</td>
<td>1987</td>
</tr>
<tr>
<td>PowerPC</td>
<td>32</td>
<td>Load-store</td>
<td>1992</td>
</tr>
<tr>
<td>DEC Alpha</td>
<td>32</td>
<td>Load-store</td>
<td>1992</td>
</tr>
<tr>
<td>HP/Intel IA-64</td>
<td>128</td>
<td>Load-store</td>
<td>2001</td>
</tr>
<tr>
<td>AMD64(EMT64)</td>
<td>16</td>
<td>Register-memory</td>
<td>2003</td>
</tr>
</tbody>
</table>

#### Small note on (possible) addressing modes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Add R4,R3</td>
<td>$R_4 = R_4 + R_3$</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R4,#3</td>
<td>$R_4 = R_4 + 3$</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4,100(R1)</td>
<td>$R_4 = R_4 + \text{Mem}[100 + R_1]$</td>
</tr>
<tr>
<td>Register indirect</td>
<td>Add R4,(R1)</td>
<td>$R_4 = R_4 + \text{Mem}[R_1]$</td>
</tr>
<tr>
<td>Indexed / Base</td>
<td>Add R3,(R1+R2)</td>
<td>$R_3 = R_3 + \text{Mem}[R_1 + R_2]$</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R1,(1001)</td>
<td>$R_1 = R_1 + \text{Mem}[1001]$</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R1,(R3)</td>
<td>$R_1 = R_1 + \text{Mem}[\text{Mem}[R_3]]$</td>
</tr>
<tr>
<td>Auto-increment</td>
<td>Add R1,(R2)+</td>
<td>$R_1 = R_1 + \text{Mem}[R_2]$; $R_2 = R_2 + d$</td>
</tr>
<tr>
<td>Auto-decrement</td>
<td>Add R1,–(R2)</td>
<td>$R_2 = R_2 - d$; $R_1 = R_1 + \text{Mem}[R_2]$</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1,100(R2)[R3]</td>
<td>$R_1 = R_1 + \text{Mem}[100 + R_2 + R_3 \cdot d]$</td>
</tr>
</tbody>
</table>

**Why Auto-increment/decrement? Scaled?**
Addressing mode statistics (ignoring register mode)

3 programs, measured on machine with all address modes (VAX)

--- Displacement: 42% avg, 32% to 55% 75%
--- Immediate: 33% avg, 17% to 43% 85%
--- Register deferred (indirect): 13% avg, 3% to 24%
--- Scaled: 7% avg, 0% to 16%
--- Memory indirect: 3% avg, 1% to 6%
--- Misc: 2% avg, 0% to 3%

75% displacement & immediate
88% displacement, immediate & register indirect

Displacement address size?

- Avg. of 5 SPECint92 programs v. avg. 5 SPECfp92 programs
- 12 - 16 bits of displacement needed
Immediate size?

- 50% ~ 60% fit within 8 bits
- 75% ~ 80% fit within 16 bits

Summary
- **Displacement**, **immediate** and **register indirect** are three most important addressing mode.
- **Displacement size** should be 12 ~ 16 bits.
- **Immediate size** should be 8 ~ 16 bits.

Stored program concept

- **Instructions are bits**
  - Programs are stored in memory — to be read or written just like data
  - Treating instructions in the same way as data greatly simplifies both the memory hardware and the software of computer systems.
**Stored program concept**

- 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

- Implicit in the stored-program idea is the need to have a register to hold the address of the current instruction being executed. In MIPS, this is named PC – program counter.

---

**Logical operations**

<table>
<thead>
<tr>
<th>Logical operations</th>
<th>C operators</th>
<th>Java operators</th>
<th>MIPS instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift left</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;&gt;</td>
<td>srl</td>
</tr>
<tr>
<td>Bit-by-bit AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and, andi</td>
</tr>
<tr>
<td>Bit-by-bit OR</td>
<td></td>
<td></td>
<td>or, ori</td>
</tr>
<tr>
<td>Bit-by-bit NOT</td>
<td>~</td>
<td>~</td>
<td>nor</td>
</tr>
</tbody>
</table>

- *Instructions that operate on fields of bits or individual bit within a word.*
Logical operations

- **Shift left logical** (`sll`) and **shift right logical** (`srl`)
  - `sll` moves all the bits in a word to the left, filling the emptied bits with 0s. (Similarly for `srl`)
  
  \[
  sll \ t2, \ s0, \ 8 \ ; \ t2 = \ s0 << 8 \text{ bits}
  \]

<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>0</td>
<td>16($s0)</td>
<td>10($t2)</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

* shamt = shift amount

Logical operations

- **And** (and) to mask bits
  
  \[
  \text{and} \ t0, \ t1, \ t2 \ ; \ t0 = \ t1 \ & \ t2
  \]

- **Or** (or) to set bits
  
  \[
  \text{or} \ t0, \ t1, \ t2 \ ; \ t0 = \ t1 \ | \ t2
  \]

- **MIPS does not have** not operator because not only requires one operand. Rather it uses nor to fulfill the work of not.
  
  \[
  \text{nor} \ t0, \ t1, \ t2 \ ; \ t0 = \ ~(t1 \ | \ t2)
  \]
Logical operations

- **And immediate** (andi)
  
  \[ \text{andi } \$s1, \$s2, 100 ; \$s1 = \$s2 \& 100 \]

- **Or immediate** (ori)
  
  \[ \text{or } \$s1, \$s2, 100 ; \$s1 = \$s2 | 100 \]

Control: Decision making instruction

What makes a computer different from a calculator?

*Answer: The ability to make decisions.*

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

- **Example. MIPS conditional branch instructions:**
  
  \[
  \text{bne } \$t0, \$t1, \text{Label} \quad \text{branch to Label if not equal} \\
  \text{beq } \$t0, \$t1, \text{Label} \quad \text{branch to Label if equal}
  \]

  Notably, Label is a numeral offset with base address in **Program Counter (PC)**. **Program counter (PC)** is sometimes named **Instruction Register (IR)**.
Control: Decision making instruction

- Example:
  
  \[
  \text{if } (i==j) \ h = i + j;
  \]
  
  \[
  \text{bne } $s0, $s1, \text{Label}
  \text{add } $s3, $s0, $s1
  \]
  
  Label: ....

<table>
<thead>
<tr>
<th>(h)</th>
<th>(i)</th>
<th>(j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s3)</td>
<td>(s0)</td>
<td>(s1)</td>
</tr>
</tbody>
</table>

Control: Decision making instruction

- Example. MIPS unconditional branch instructions:

  \[
  j \ Label \quad \text{jump to Label}
  \]

J-type instruction

<table>
<thead>
<tr>
<th>Calculation of the jump target address</th>
</tr>
</thead>
<tbody>
<tr>
<td>26 bit Label-specified address</td>
</tr>
</tbody>
</table>

\[
\text{beq } $s4, $s5, \text{Lab1}
\text{add } $s3, $s4, $s5
\text{j \ Lab2}
\]

Lab1: \( \text{sub } $s3, $s4, $s5 \)

Lab2: ...

Example:

\[
\text{if } (i!=j) \ h=i+j;
\text{else} \ h=i-j;
\]

Po-Ning Chen@CM.NCTU

--- Po-Ning Chen ---
So far:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add $rd, $rs, $rt</code></td>
<td>$rd = $rs + $rt</td>
</tr>
<tr>
<td><code>sub $rd, $rs, $rt</code></td>
<td>$rd = $rs - $rt</td>
</tr>
<tr>
<td><code>lw $rt, 100($rs)</code></td>
<td>$rt = Memory[$rs+100]</td>
</tr>
<tr>
<td><code>sw $rt, 100($rs)</code></td>
<td>Memory[$rs+100] = $rt</td>
</tr>
<tr>
<td><code>bne $rs, $rt, Label</code></td>
<td>Next instr is at PC+4+Label if $rs ≠ $rt (I-type)</td>
</tr>
<tr>
<td><code>beq $rs, $rt, Label</code></td>
<td>Next instr is at PC+4+Label if $rs = $rt (I-type)</td>
</tr>
<tr>
<td><code>j Label</code></td>
<td>Next instr. is at Label</td>
</tr>
</tbody>
</table>

**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>
</table>

**I-type**

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

**J-type**

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

---

**PC-relative and pseudodirect addressing**

**Example.**

While (save[i] == k) i += 1;

<table>
<thead>
<tr>
<th>i</th>
<th>k</th>
<th>save</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s3$</td>
<td>$s5$</td>
<td>$s6$</td>
</tr>
</tbody>
</table>

Loop:
- `sll $t1, $s3, 2 ;$t1=4*i`
- `add $t1, $t1, $s6 ;$t1=addr of save[i]`
- `lw $t0, 0($t1) ;$t0=save[i]`
- `bne $t0, $s5, Exit ;if save[i]!=k, goto Exit`
- `addi $s3, $s3, 1 ;i=i+1`
- `j Loop ;goto Loop`

Exit:
PC-relative and pseudodirect addressing

Example (Continue)

Exit: 80012+4+2*4 = 80024
Loop: 20000*4+[80024/16]*16^7 = 80000

80000 0 0 19 9 4 0
80004 0 9 22 9 0 32
80008 35 9 8 0
80012 5 8 21 2
80016 8 19 19 1
80020 2 20000
80024 ...

Exit: 80012+4+2*4 = 80024
Loop: 20000*4+[80024/16]*16^7 = 80000

Control flow

We have: beq, bne, what about branch-if-less-than?

blt $s1, $s2, Label ; This command is implemented as:

slt $t0, $s1, $s2
bne $t0, $zero, Label ; $zero=0

Heeding von Neumann’s warning about the simplicity of the ”equipment,” the MIPS architecture does not include branch on less than because it is too complicated; either it would stretch the clock cycle or this instruction would take extra clock cycles per instruction. Two faster instructions are more useful.

if $s1 < $s2 then
$t0 = 1
else
$t0 = 0

(set on less than)
### Registers 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>
<tr>
<td>$at$</td>
<td>1</td>
<td>reserved for the assembler to handle, e.g., large constant</td>
</tr>
<tr>
<td>$k0$–$k1$</td>
<td>26–27</td>
<td>Reserved for the operating system</td>
</tr>
</tbody>
</table>

### Recall: Consideration of small constants

- Small constants are used quite frequently (50% of operands)

  e.g.,

  \[
  A = A + 5; \\
  B = B + 1; \\
  C = C - 18;
  \]

- Solutions? **Design Principle 3: Make the common case fast.**
  - Put 'typical constants' in memory and load them. 
    ```
    lw $t0, 0($a1) \\
    add $s3, $s3, $t0
    ```
  - Time consuming
  - Create hard-wired registers (like $zero) for constants like one. 
    - Limited numbers of constants can be provided.
  - Create specific instruction for operations of frequently used constants.

  E.g.,

  ```
  add $s3, $s3, 4 ;I-type, the constant can be 16-bit in size 
  ; addi = add immediate
  ```
Consideration of small constants

- Constant operands are also popular in comparisons.
  \[
  \text{slti } t0, s2, 10 : t0 = 1 \text{ if } s2 < 10
  \]

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

Consideration of large constants

- What if the constant requires more than 16 bits.
- Must use two instructions, new "load upper immediate" instruction
  
  ```
  lui $t0, 1010101010101010
  ```

  filled with zeros

  | 1010101010101010 | 0000000000000000 |
Consideration of large constants

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

\[
\text{ori } \$t0, \$t0, 1010101010101010
\]

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

Control flow

- Example. MIPS jump register instruction:

\[
\text{jr } \$t0 ; \text{jump to the address in } [\$t0]
\]
Assembly language vs. machine language

- Assembly provides convenient symbolic representation
  - Much easier than writing down numbers
    - Which one is easier to memorize for human?
      \[
      \text{add } \text{t0}, \text{s2}, \text{t0} \\
      \text{or } \text{00000010010000100000000100000B}
      \]
    - Which one is easier for machine to recognize?
      \[
      \text{add } \text{t0}, \text{s2}, \text{t0} \\
      \text{or } \text{00000010010000100000000100000B}
      \]

Assembly language vs. machine language

- Machine language is the underlying reality
  - In human/assembler, one may presume “destination first” in instruction.
  - However, in machine language, destination is not necessarily first, depending on which way would benefit the hardware design.

<table>
<thead>
<tr>
<th>R-type</th>
<th>I-type</th>
<th>J-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td>op</td>
<td>26-bit address</td>
<td></td>
</tr>
</tbody>
</table>
Assembly language vs. machine language

- 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

--- Po-Ning Chen ---

In the sequel, we will discuss:
- support for procedures
- linkers, loaders, memory layout
- stacks, frames, recursion
- manipulating strings and pointers
- interrupts and exceptions
- system calls and conventions

- Some of these, we’ll talk more about later
- We’ll talk about compiler optimizations when we hit chapter 4.
Supporting procedures in computer hardware

- **Support for procedure in MIPS**
  - $a0-$a3: four argument registers that are used to pass parameters
  - $v0-$v1: two value registers that are used to return values
  - $ra: one return address register that is used to the point of origin

  ```
  jal ProcedureAddress ; jump and link
  \ (*i.e., jump to ProcedureAddress
  and store the return address,
  (PC+4), at $ra
  
  At the end of the subroutine, we therefore put
  jr $ra
  ```

- **Steps to call Procedure X**
  - Place parameters in $a0-$a3
  - `jal Xaddress ; J-type`

  ```
  PC_{31:28} 26 bit Label-specified address \text{\texttt{00}}
  ```
  
  - Execute, and put the return values in $v0-$v1
  - `jr $ra`

- **What if we need some local parameters.**
- **Solution:** Stack – a last-in-first-out queue.
By convention, stack “grows” from higher addresses to lower addresses

- $sp = stack pointer

```
addi $sp, $sp, -12  ; make room for 3 items
sw $t1, 8($sp)
sw $t0, 4($sp)
sw $s0, 0($sp)

add $t0, $a0, $a1  ; $t0 = g+h
add $t1, $a2, $a3  ; $t1 = i+j
sub $s0, $t0, $t1  ; $s0 = (g+h)-(i+j)

add $v0, $s0, $zero ; $v0 = $s0+0
lw $s0, 0($sp)
lw $t0, 4($sp)
lw $t1, 8($sp)
addi $sp, $sp, 12
jr $ra
```

```
int leaf_example(int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```
Supporting procedures in computer hardware

- Procedures that do not invoke other procedures (including themselves, which causes a “recursive” call) are called leaf procedures.
- How nested procedures work?
- Answer: Push all the necessary registers into stack before invoking other procedures.

Nested procedure

```
Fact: addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)

slti $t0, $a0, 1  ;test for n<1
beq $t0, $zero, L1  ;if n>=1, go to L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr $ra

L1: addi $a0, $a0, -1  ;n-1
jal Fact

lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
mul $v0, $a0, $v0  ;return n*fact(n-1)
jr $ra
```

```
int fact(int n)
{
    if (n<1) return(1);
    else return (n*fact(n-1));
}
```
$fp = \text{frame pointer, a frame (of variables) used by the procedure}$

- Sometimes, a procedure may require a local array, or structure, or a large number of local variables.
- Question is how to deal with such situation?
- Answer: Using stack space.
  - In such case, the procedure will save $sp$ into $fp$ on a call (and use $sp$ to refer to various items in an, e.g., array) and $sp$ is restored using $fp$.

- The usage of $fp$ is not truly necessary. For example, the C compiler in MIPS/Silicon Graphics does not use $fp$; so there is an extra $s8$=$30$ in the machine.

$gp = \text{global pointer}$

- C has two storage classes: automatic and static.
  - Automatic variables are local to a procedure, while static variables exist across processors.

- MIPS software reserves another register, called the global pointer ($gp$), to refer to the memory used to store static variables.
Policy of use conventions

- Data structures like linked lists tend to grow during their life times.
- The segment for such data structures is traditionally called the heap.
- The previous allocation allows the stack and heap to grow towards each other, thereby allowing the efficient use of memory.
- C uses malloc() to allocate space on the heap, and uses free() to release space from the heap.
Number system

- **ASCII** (American Standard code for information interchange)
  - 7-bit code for alphabetic numerical characters
  - Bit 8 is sometimes used as parity-check bit or used to form the extended ASCII code
  - Extended ASCII code can be displayed, but cannot necessarily be printed out.

```
| X0 | X1 | X2 | X3 | X4 | X5 | X6 | X7 | X8 | X9 | XA | XB | XC | XD |XE |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

(Note that upper- and lowercase letters differ by exactly 32.)

Number system

```
| X0 | X1 | X2 | X3 | X4 | X5 | X6 | X7 | X8 | X9 | XA | XB | XC | XD |XE |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |

(Note that upper- and lowercase letters differ by exactly 32.)
Number system

- For convenience of ASCII code handling (which are based on 8-bit or byte), byte movement operations are provided.

\[
\begin{align*}
\text{lb} & \quad \$t0, \ 0(\$sp) \\
\text{sb} & \quad \$t0, \ 0(\$gp)
\end{align*}
\]

- The MIPS instruction set also has explicit instructions to load and store halfwords (rightmost 16 bits of a word)

\[
\begin{align*}
\text{lh} & \quad \$t0, \ 0(\$sp) \\
\text{sh} & \quad \$t0, \ 0(\$gp)
\end{align*}
\]

Number system

- How to handle a string? Three solutions:
  - Attach the length of a string at the beginning of the string.
  - Use a separate associated variable to store the length of a string.
  - Use an end-of-string character.
    - C language uses this option for which the end-of-string = NUL.
    - For example, “Cal” = 67, 97, 108, 0.
Unicode

- Unicode is a universal encoding of the alphabet of most human language.
  - Java uses Unicode for characters.

Addressing modes supported by MIPS

1. Immediate addressing

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>Immediate (4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $29, $29, 4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

addi $29, $29, 4
2. Register addressing

\[
\text{add } \$s1, \$s2, \$s3
\]

3. Base or displacement addressing

\[
\text{lw } \$s1, 100(\$s2)
\]
Addressing modes supported by MIPS

4. PC-relative addressing (E.g., Fig. 2.13)

\[ \text{beq } \$s4, \$s5, \text{Label} \]

Addressing modes supported by MIPS

5. Pseudodirect addressing (E.g., Fig. 2.13)

\[ \text{j Label} \]
Decoding machine code

<table>
<thead>
<tr>
<th>R-format</th>
<th>addi</th>
<th>jal</th>
<th>beq</th>
<th>bne</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>format</th>
<th>add</th>
<th>sub</th>
<th>and</th>
<th>or</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>format</th>
<th>sll</th>
<th>srl</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>format</th>
<th>jr</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>format</th>
<th>slt</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>format</th>
<th>jal</th>
</tr>
</thead>
</table>

* Self-practice the example on page 102.
**Pseudo instructions for assembler**

- Pseudoinstructions for assembler
  - Example. `blt ← slt and bne`
  - Example. `move $t0, $t1 ← add $t0,$zero,$t1`

- In short, pseudoinstructions benefit those who program assembly language directly, yet programming using real instructions certainly can achieve a better performance.

---

**Starting a program**

A translation hierarchy for C
Starting a program

- **Object file**
  - A combination of machine language instructions, data and information needed to place instructions properly in memory.
  - Example. Obj file in Unix
    - **Object file header**: Size and position of pieces of object file
    - **Text segment**: Machine language code
    - **Data segment**: Static and dynamic data needed in the program
    - **Relocation information**: Identify instructions and data that depend on absolute addresses when the program is loaded into memory
    - **Symbol table**: Labels used in branches and data transfer instructions, other than absolute addresses
    - **Debugging information**: A concise description of how the modules were compiled.

Example of OBJ file

<table>
<thead>
<tr>
<th>Object file header</th>
<th>Name</th>
<th>Procedure A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test size</td>
<td>200kb</td>
<td></td>
</tr>
<tr>
<td>Data size</td>
<td>30kb</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text segment</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$@</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$@</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data segment</th>
<th>O</th>
<th>C (1)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Relocation information</th>
<th>Address</th>
<th>Instruction type</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$@</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$@</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol table</th>
<th>Label</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object file header</th>
<th>Name</th>
<th>Procedure 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test size</td>
<td>200kb</td>
<td></td>
</tr>
<tr>
<td>Data size</td>
<td>30kb</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text segment</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$@</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$@</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data segment</th>
<th>O</th>
<th>C (1)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Relocation information</th>
<th>Address</th>
<th>Instruction type</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$@</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$@</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol table</th>
<th>Label</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Starting a program

- Linker or link editor
  - The linker, based on relocation information and symbol table, finds the old (internal and external) addresses/references and replace them with the new (internal and external) addresses/references.

  - After linking, an executable file is produced, containing no unresolved references, relocation information, symbol table, and debugging information.

  - Notably, library routines may still have unresolved addresses.
Executable example after linking

<table>
<thead>
<tr>
<th>Executable file header</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Text size</td>
<td>Data size</td>
</tr>
<tr>
<td></td>
<td>300 hex</td>
<td>50 hex</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test segment</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0040 0000 hex</td>
<td>lw $a0, 8000 hex (4gp)</td>
</tr>
<tr>
<td></td>
<td>0040 0004 hex</td>
<td>jal 40 0100 hex</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0040 0100 hex</td>
<td>sw $al, 8002 hex (4gp)</td>
</tr>
<tr>
<td></td>
<td>0040 0104 hex</td>
<td>jal 40 0000 hex</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data segment</th>
<th>Address</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1000 0000 hex</td>
<td>(x)</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1000 0020 hex</td>
<td>(y)</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dynamically linked libraries (DLLs)

- **Disadvantage of static linked libraries** (described above)
  - The library itself should become part of the executable code.
  - The library itself may be large, while only a relatively small portion is used by program.
    - E.g., the standard C library is 2.5MB.

- **DLL**
  - The library is not linked and loaded until the program is run.

- The early version of DLL still linked **all** routines of the library.

- The newly **lazy procedure linkage version** of DLLs links routines only after it is called.
**DLL via lazy procedure linkage**

First call to DLL routine

1. Call a routine that puts an ID in a register to identify the desired library routine.
2. Jump to the dynamic linker-loader to find the desired routine.
3. The linker-loader, after finding the desired routine, remaps it and changes the address in the indirect jump location to point to that routine.

- \( l_i l_i = \text{load immediate, a pseudoinstruction (cf. A-57)} \)

---

**Starting a Java Program**

- Java is compiled first to instructions that are easy to interpret, named the **Java bytecode** instruction set.
- Then a software interpreter, called a Java Virtual Machine (JVM), can execute Java bytecodes.
- Java bytecode is **portable**, but has the drawback of **low performance**.
- To achieve better performance, the JVM can invoke the **Just-In-Time (JIT) compiler** that selectively compiles bytecode into native machine language of the machine on which the program is running.
  - So JIT compiler operates at runtime, translating the bytecode segments into the native code of the computer on the fly.
Starting a Java Program

- Java program
  - Java compiler
    - Class files (Java bytecodes)
    - Java library routines (machine language)
    - Just In Time compiler
      - invoke
      - Java virtual machine
        - Complied Java methods (machine language)

How compilers optimize

- **High-level optimizations are “transformations”**
  - E.g., **procedure inlining transformation** that replaces a call to a function by the body of the function.
  - E.g., **loop unrolling transformation** in which multiple copies of the loop body are made.

- **Three classes** of optimizations can be defined.
  - **Local optimization** that works in a single basic block
  - **Global optimization** that works across multiple blocks
  - **Global register allocation** that allocates variables to registers.

Basic block: A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning) (cf. page 75).
How compilers optimize

- Example of local optimization
  - **Local common subexpression elimination**
    
    \[ x[i] = x[i] + 4 \]

    The address calculation for \(x[i]\) occurs twice (neither \(x\) nor \(i\) changes in the second calculation) so the calculation can be re-used.

- **Global common subexpression elimination** is possible, when it is applied across two or more blocks.

How compilers optimize

- Some other optimizations (that can be applied **locally** and **globally**)
  - **Strength reduction**: Replace complex operations by simple operations. E.g., replace \(\text{mult}\) by a shift left.
  - **Constant propagation** (or constant folding): Find constants in code and propagate them.
  - **Copy propagation**: Propagate values that are simply “copies”, enabling other optimizations such as common subexpression elimination. E.g, replace \(A\) by \(X\), when \(A = X\) occurs in code.
  - **Dead store elimination**: Eliminates the store to values that are no longer used.
  - **Dead code elimination**: Eliminate unused code that cannot affect the final result of the program.
How compilers optimize

- Two more global optimizations
  - **Code motion**: Move code inside a loop, which is loop invariant, out of the loop (so that the code is computed only once instead of as many times as the loop repeats)
  - **Induction variable elimination**: Replace array index by pointer index.

- There are more optimization approaches....
- For a true example, please read Section 2.12 (page 2.12-3).

---

### Induction variable elimination: Array versus pointer

```
clear1(int array[], int size)
{
    int i;
    for (i=0;i<size;i=i+1)
        array[i]=0;
}
```

```
clear2(int *array, int size)
{
    int *p;
    for (p=&array[0];p<&array[size];p=p+1)
        *p=0;
}
```
Induction variable elimination: Array versus pointer

<table>
<thead>
<tr>
<th>array</th>
<th>size</th>
<th>i or p</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a0</td>
<td>$a1</td>
<td>$t0</td>
</tr>
</tbody>
</table>

```
mv $t0, $zero ; i=0
loop1:
    add $t1, $t0, $t0
    add $t1, $t1, $t1
    add $t2, $a0, $t1
    sw $zero, 0($t2) ; array[i]=0
    addi $t0, $t0, 1 ; i=i+1
    slt $t3, $t0, $a1
    bne $t3, $zero, loop1 ; if (i<size) goto loop1

    mv $t0, $a0 ; p=&array[0]
    add $t1, $a1, $a1
    add $t1, $t1, $t1
    add $t2, $a0, $t1 ; &array[size]

    loop2: sw $zero, 0($t0) ; Memory[p]=0
            addi $t0, $t0, 4 ; p=p+1, where "1" means a word
            slt $t3, $t0, $t2
            bne $t3, $zero, loop2 ; if (p<&array[size]) goto loop2.
```

An example to put it all together

- Can we figure out the code?

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

```
swap:
    sll $t1, $a1, 2 ; $t1=k*4
    add $t1, $a0, $t1 ; $t1=v+(k*4)
    lw $t0, 0($t1) ; $t0=v[k]
    lw $t2, 4($t1) ; $t2=v[k+1]
    sw $t2, 0($t1) ; v[k]=t2
    sw $t0, 4($t1) ; v[k+1]=$t0
    jr $ra
```
Milestones for Intel IA-32 Instructions

- 1978: The Intel 8086/8088 is announced (16 bit architecture)
  - Each register has a dedicated use; so it is not considered a general-purpose register architecture.
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions, +protected mode
- 1985: The 80386 extends to 32 bits, +new addressing modes, +virtual-8086 mode
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions, superscalar, two-level cache
- 1997-1999: 57 new MMX (Multi Media Extensions) instructions are added, Pentium II, Pentium II Xeon, Celeron
- 2001: Pentium 4 and Xeon, NetBurst Micro-Architecture, add another new 140 instructions (SSE2)

- 2003: AMD extends the IA-32 architecture to increase address space to 64 bits, widens all registers to 64 bits and other changes (AMD64)
- 2004: Intel capitulates and embraces AMD64 (calls it EM64T) and adds more media extensions
Milestones for Intel IA-32 Instructions

This history illustrates the impact of the “golden handcuffs” of compatibility

“adding new features as someone might add clothing to a packed bag”

“an architecture that is difficult to explain and impossible to love”

Real stuff: Intel IA-32 Instructions

- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    - e.g., “base or scaled index with 8 or 32 bit displacement”
IA-32 Registers and Data Addressing

- Registers in the 32-bit subset that originated with 80386

<table>
<thead>
<tr>
<th>Register</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>AL, AH, BL, BH</td>
</tr>
<tr>
<td>ECX</td>
<td>CL, CH, DL, DH</td>
</tr>
<tr>
<td>EDX</td>
<td>DL, DH</td>
</tr>
<tr>
<td>EBX</td>
<td>DL, DH</td>
</tr>
<tr>
<td>ESP</td>
<td>AL, AH, BL, BH</td>
</tr>
<tr>
<td>EBP</td>
<td>CL, CH, DL, DH</td>
</tr>
<tr>
<td>ESI</td>
<td>CL, CH, DL, DH</td>
</tr>
<tr>
<td>EDI</td>
<td>DL, DH</td>
</tr>
<tr>
<td>CS</td>
<td>Code segment pointer</td>
</tr>
<tr>
<td>SS</td>
<td>Stack segment pointer (top of stack)</td>
</tr>
<tr>
<td>DS</td>
<td>Data segment pointer 0</td>
</tr>
<tr>
<td>ES</td>
<td>Data segment pointer 1</td>
</tr>
<tr>
<td>FS</td>
<td>Data segment pointer 2</td>
</tr>
<tr>
<td>GS</td>
<td>Data segment pointer 3</td>
</tr>
<tr>
<td>EIP</td>
<td>Instruction pointer (PC)</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>Condition codes</td>
</tr>
</tbody>
</table>

IA-32 Register Restrictions

- Registers are not “general purpose” – note the restrictions below

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
<th>Register restrictions</th>
<th>MIPS equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register indirect</td>
<td>Address is in a register</td>
<td>not ESP or EBP</td>
<td>$R31, $R8(16)</td>
</tr>
<tr>
<td>Data plus scaled index</td>
<td>The address is $base + $offset x $index</td>
<td>any 16-bit index, not ESP</td>
<td>$R16, $R17, 8</td>
</tr>
<tr>
<td>Data plus scaled index with 0 or 31-bit displacement</td>
<td>The address is $base + $offset x $index + $disp, where $index has the value 0, 1, 2, or 3.</td>
<td>any 8-bit index, not ESP</td>
<td>$R8, 8($8)</td>
</tr>
</tbody>
</table>
**IA-32 Typical Instructions**

- Four major types of integer instructions:
  1. Data movement including *mov, push, pop*
  2. Arithmetic and logical (destination register or memory)
  3. Control flow (use of condition codes / flags)
  4. String instructions, including string move and string compare

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>JE EIP + displacement</td>
<td>(bits)</td>
</tr>
<tr>
<td>CALL</td>
<td>Offset</td>
</tr>
<tr>
<td>MOV EBX, EDX + 48</td>
<td>6 1 8 8</td>
</tr>
<tr>
<td>TEST EDX, EDX</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

**IA-32 instruction Formats**

- Typical formats: (notice the different lengths)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>Offset</td>
</tr>
<tr>
<td>MOV EBX, EDX + 48</td>
<td>6 1 8 8</td>
</tr>
<tr>
<td>TEST EDX, EDX</td>
<td>Immediate</td>
</tr>
</tbody>
</table>
IA-32 instruction Formats

- **Complexity:**
  - Instructions from 1 to 17 bytes long
  - One operand must act as both a source and destination.
  - One operand can come from registers, memory, immediate.
  - Complex addressing modes
    - e.g., “base or scaled index with 8 or 32 bit displacement”
    - Although it supports almost all known addressing modes, some restrictions on the use of registers are applied in specific addressing mode.
Summary

- Instruction complexity is only one variable
  - Lower instruction count vs. higher clock-per-instruction (CPI) / lower clock rate
- Intel has a 16-bit microprocessor two years before its competitor's more elegant architectures, such as Motorola 68000.
- This head start led to the selection of the 8086 as the CPU for the IBM PC.
- Intel engineers generally acknowledge that the 80x86 is more difficult to build than machines like MIPS.
- What the 80x86 lacks in style is made up in quantity. I.e., Intel can afford more resources to help overcome the added complexity.
- The saving grace is that the most frequently used 80x86 architectural components are not too difficult to implement; so compilers must avoid the portions of the architecture that are hard to implement fast.

Key summary in this chapter

- Design Principles:
  - Simplicity favors regularity
  - Smaller is faster
  - Make the common case fast
    - Common case statistics from gcc and spice

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>MIPS examples</th>
<th>Frequency gcc</th>
<th>Frequency spice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add, sub, addi</td>
<td>48%</td>
<td>50%</td>
</tr>
<tr>
<td>Data transfer</td>
<td>lw, sw, lb, lb, lui</td>
<td>33%</td>
<td>41%</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>beq, bne, slt, slti</td>
<td>17%</td>
<td>8%</td>
</tr>
<tr>
<td>Jump</td>
<td>j, jr, jal</td>
<td>2%</td>
<td>1%</td>
</tr>
</tbody>
</table>

- Good design demands compromises
Fallacies

- More powerful instructions mean higher performance?
  - Perform 32-bit memory-to-memory on a 133-MHz Pentium
  - MOVS is 1.5 times slower than normal load-to-integer-register-then-store-back-to-memory with code replications.
  - MOVS is 2.0 times slower than normal load-to-floating-point-register-then-store-back-to-memory with code replications.

Suggestive exercises

- 2.2, 2.3, 2.4, 2.6, 2.13, 2.20
- 2.29, 2.32, 2.34, 2.37, 2.38, 2.39, 2.40, 2.49, 2.50, 2.51

- Should you have questions on exercises, ask the teaching assistants (助教).
Answers to Check yourself

- Page 52: 3, 2, 1.
- Page 59: 2.
- Page 66: 2.
- Page 77: 1st answer is all of the above, 2nd answer is 1.
- Page 88: 1, 3 (for 2, local dynamic variables in C are handled automatically on the stack).
- Page 95: 1 (strings are special types in Java; Java strings are not null terminated, and so operations is faster in Java).
- Page 104: 1st answer is 4, 2nd answer is 6, 3rd answer is 4.