Components of a Computer
Hierarchical Layers of Program Code

High-level language program (in C)

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

Assembly language program (for MIPS)

```
swap:
multi $2, $5, 4
add $2, $4, $2
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
jr $31
```

Binary machine language program (for MIPS)

```
0000000001010001000000000011000
00000000030011000000110000010001
1001100311001000000000000000000
10011001111010000000000000000000
101110011111010000000000000000000
1011100111111001000000000000000000
1000000000000000000000000000000000
```

Assembler

Compiler
Instruction Set

- The repertoire of instructions of a computer

- Different computers have different instruction sets
  - But with many aspects in common
The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
  - Founded in 1984 by …?
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
Instruction Set

- Stored-program concept
  - The idea that instructions and data of many types can be stored in memory as numbers, leading to stored-program computer

- Let us look into MIPS instruction set one by one to understand this
Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c

# a gets b + c
```
Arithmetic Operations

- Operand is a quantity on which an operation is performed

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

- How many operands in this instruction?

- All arithmetic operations have this form
Arithmetic Operations

- All arithmetic operations have same form
  - What if we want to add \( b, c, d, \) and \( e \) and put the result into \( a \)?
Design Principle 1

- All arithmetic operations have the same form
- *Design Principle 1*: Simplicity favors regularity
  - Regularity makes hardware implementation simpler
Arithmetic Example

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

- Compiled MIPS code: ?

- Hints:
  - Use sub instruction, e.g., \( a=b-c \), is sub a,b,c
  - Use two temporary variables t0 and t1
Arithmetic Example

- **C code:**
  
  \[ f = (g + h) - (i + j); \]

- **Compiled MIPS code:**
  
  ```assembly
  add t0, g, h   # temp t0 = g + h
  add t1, i, j   # temp t1 = i + j
  sub f, t0, t1  # f = t0 - t1
  ```
Register Operands

- The operands of arithmetic instructions must be from special location in hardware called *registers*.

- *Registers* are primitives of hardware design and are visible to programmers.
Register Operands

- Assembler names
  - $t0, t1, ..., t9 for temporary values
  - $s0, $s1, ..., $s7 for saved variables
Register Operand Example

- Compiler’s job to associate variables of a high-level program with registers

- C code:
  \[ f = (g + h) - (i + j); \]
  - \( f, \ldots, j \) in \$s0, \ldots, \$s4

- Compiled MIPS code?
Register Operand Example

- Compiled MIPS code:
  
  ```
  add $t0, $s1, $s2
  add $t1, $s3, $s4
  sub $s0, $t0, $t1
  ```
Register Operands

- MIPS has a $32 \times 32$-bit register file
  - Numbered 0 to 31
  - 32-bit data called a “word”

- Word is the natural unit of access, typically 32 bits, corresponds to the size of a register in MIPS

- There may be only 3 operands and they must be chosen from one of the 32 registers. Why only 32?
Design Principle 2

- Smaller is faster
  - Larger registers will increase clock cycle time --- electronic signals takes longer when they travel farther

- Design principles are not hard truths but general guidelines
  - 31 registers instead of 32 need not make MIPS faster
Memory Operands

- Programming languages, C, Java, …
  - Allow complex data structures like arrays and structures
  - They often contain many more data elements than the number of registers in a computer
  - Where are they stored?
    - Memory

- But, arithmetic operations are applied on register operands

- Hence, data transfer instructions are required to transfer data from memory to registers
  - Load values from memory into registers
  - Store result from register to memory
- Memory is like an array
- Data transfer instructions must supply the address (index/offset) of the memory (array)
Memory Operands

- Memory is byte addressed
  - Each address identifies an 8-bit byte

- Words are aligned in memory
  - Each word is 32 bits or 4 bytes
  - To locate words, addresses are in multiples of 4
- A is an array of words
- What is the offset to locate A[8]?
  - A[0] – 0
  - A[2]– 8
  - …
Memory Operands

- Why is memory not word-addressable?
Memory Operands

- Why is memory not word-addressable?

- Bytes are useful in many programs. In a word addressable system, it is necessary first to compute the address of the word containing the byte, fetch that word, and then extract the byte from the two-byte word. Although the processes for byte extraction are well understood, they are less efficient than directly accessing the byte. For this reason, many modern machines are byte addressable.
Memory Operands

- *Load* instruction
  - lw refers to *load word*
  - lw registerName, offset (registerWithBaseAddress)
  - lw $t0, 8 ($s3)
Memory Operand Example 1

- C code:
  \[ g = h + A[8]; \]
  - g in $s1
  - h in $s2
  - base address of A in $s3
  - A is an array of 100 words

- Compiled MIPS code?
Memory Operand Example 1

- **C code:**
  
  ```
  g = h + A[8];
  ```
  
  - g in $s1, h in $s2, base address of A in $s3

- **Compiled MIPS code:**
  
  ```
  lw $t0, 32($s3)    # load word
  add $s1, $s2, $t0
  ```
Memory Operand Example 2

- **C code:**
  \[
  \]
  - h in $s2, base address of A in $s3

- **Compiled MIPS code:**
Memory Operand Example 2

- **C code:**
  
  ```
  ```
  
  - `h` in `$s2`, base address of `A` in `$s3`

- **Compiled MIPS code:**
  
  ```
  lw  $t0, 32($s3)    # load word
  add $t0, $s2, $t0
  sw  $t0, 48($s3)    # store word
  ```
Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!
Immediate Operands

- Constant data specified in an instruction
  \( \text{addi } \$s3, \$s3, 4 \)
- No subtract immediate instruction
  - Just use a negative constant
    \( \text{addi } \$s2, \$s1, -1 \)
Design Principle 3

- Make the common case fast
  - Small constants are common: immediate operand avoids a load instruction
  - Allows us to avoid using memory meaning faster operations and lesser energy
The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers
    add $t2, $s1, $zero
Problems

- In the snippet of MIPS assembler code below, how many times is the memory accessed? How many times data is fetched from memory?

```
lw $v1, 0($a0)
addi $v0, $v0, 1
sw $v1, 0($a1)
sw $v1, 0($a1)
addi $a0, $a0, 1
```

- Write the MIPS assembly language instructions for the following C statement. Assume $f$ is stored in $s0$, $g$ in $s1$ and $h$ in $s2$. Use a minimal number of MIPS statements and a minimal number of registers.

```
f = g + (h – 5)
```
Stored-program concept

- The idea that instructions and data of many types can be stored in memory as numbers, leading to stored-program computer
Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!
- Register numbers
  - $t0 – t7$ are reg’ s $8 – 15$
  - $s0 – s7$ are reg’ s $16 – 23$
Example

add $t0, $s1, $s2

<table>
<thead>
<tr>
<th>special</th>
<th>$s1</th>
<th>$s2</th>
<th>$t0</th>
<th>0</th>
<th>add</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>18</td>
<td>8</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>

0000001000110010010000000000100000

<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>
Representing Instructions

- The layout or the form of representation of instruction is composed of fields of binary numbers.

- The numeric version of instructions is called machine language and a sequence of such instructions is called machine code.
Instruction types

- R format (for register)
  - Add, sub

- I-format (for immediate)
  - Immediate
  - Data transfer
## MIPS R-format Instructions

<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</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

### Instruction fields
- **op**: operation code (opcode)
- **rs**: first source register number
- **rt**: second source register number
- **rd**: destination register number
- **shamt**: shift amount (00000 for now)
- **funct**: function code (extends opcode)
## MIPS I-format Instructions

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>constant or 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>

- Immediate arithmetic and load/store instructions
  - rt: destination register number
  - rs: register number with address
  - Constant/ Address: offset added to base address in rs
Design Principle 4

- Ideally,
  - Keep all instructions of the same format and length
  - But this makes it difficult to address large memories
    - Compromise and allow different formats

- Principle 4: Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible
    - See example in page 98
Stored Program Computers

- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs
Logical Operations

- Instructions for bitwise manipulation

<table>
<thead>
<tr>
<th>Operation</th>
<th>C</th>
<th>Java</th>
<th>MIPS</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>Bitwise AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and, andi</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td></td>
<td></td>
<td>or, ori</td>
</tr>
<tr>
<td>Bitwise NOT</td>
<td>~</td>
<td>~</td>
<td>nor</td>
</tr>
</tbody>
</table>

- Useful for extracting and inserting groups of bits in a word
Shift Operations

- Shift left logical
  - Shift bits to the left and fill the empty bits with zeros
  - `sll $t2,$s0,3`

```
0000 0000 0000 0000 0000 0000 0000 0001
```
Shift Operations

- Shift left logical
  - Shift bits to the left and fill the empty bits with zeros
  - \texttt{sll}\ $t2,$s0,3

0000 0000 0000 0000 0000 0000 0000 0001

0000 0000 0000 0000 0000 0000 0000 1000
Shift Operations

- Shift left logical
  - Shift bits to the left and fill the empty bits with zeros
  - `sll $t2,$s0,3`

| 0000 0000 0000 0000 0000 0000 0000 0001 |
| 0000 0000 0000 0000 0000 0000 0000 1000 |

- `sll` by `i` bits multiplies by $2^i$
## Shift Operations

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<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>
Shift Operations

<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>

\[
\begin{array}{cccccc}
0 & 0 & 16 & 10 & 3 & 0 \\
$\text{s0}$ & $\text{t2}$ & & & & \\
\end{array}
\]

- `sll $t2,$s0,3`
- `shamt`: how many positions to shift
- Similarly, …
  - Shift right logical
**AND Operations**

- Mask bits in a word
  - Select some bits, clear others to 0

and $t0, t1, t2$
OR Operations

- Include bits in a word
  - Set some bits to 1, leave others unchanged

or $t0$, $t1$, $t2$
## NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - \( a \text{ NOR } b = \text{ NOT} \ (a \text{ OR } b) \)

```assembly
nor $t0, $t1, $zero
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t1</td>
<td>0000 0000 0000 0000 0011 1100 0000 0000</td>
</tr>
<tr>
<td>$t0</td>
<td>1111 1111 1111 1111 1100 0011 1111 1111</td>
</tr>
</tbody>
</table>
Hexadecimal Numbers

- Reading binary numbers are tedious
- So, hexadecimal representation is popular
- Base 16 is power of two and hence it is easy to replace each group of 4 binary digits
Hexadecimal

- **Base 16**
  - Compact representation of bit strings
  - 4 bits per hex digit

<table>
<thead>
<tr>
<th></th>
<th>0000</th>
<th>4</th>
<th>0100</th>
<th>8</th>
<th>1000</th>
<th>c</th>
<th>1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0001</td>
<td>5</td>
<td>0101</td>
<td>9</td>
<td>1001</td>
<td>d</td>
<td>1101</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>6</td>
<td>0110</td>
<td>a</td>
<td>1010</td>
<td>e</td>
<td>1110</td>
</tr>
<tr>
<td>2</td>
<td>0011</td>
<td>7</td>
<td>0111</td>
<td>b</td>
<td>1011</td>
<td>f</td>
<td>1111</td>
</tr>
</tbody>
</table>

- Example: eca8 6420 ?
- Example:
  0001 0011 0101 0111 1001 1011 1101 1111
Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- beq rs, rt, L1
  - if (rs == rt) branch to instruction labeled L1;
- bne rs, rt, L1
  - if (rs != rt) branch to instruction labeled L1;
- j L1
  - unconditional jump to instruction labeled L1
Compiling If Statements

- C code:
  
  ```c
  if (i==j) f = g+h;
  else f = g-h;
  ```

- f, g, ... in $s0, $s1, ...

![Diagram showing the compilation of an if statement](image-url)
Compiling If Statements

- C code:
  
  ```c
  if (i==j) f = g+h;
  else f = g-h;
  ```

- f, g, ... in $s0, $s1, ...

- Compiled MIPS code:
  
  ```
  bne $s3, $s4, Else
  add $s0, $s1, $s2
  j Exit
  Else: sub $s0, $s1, $s2
  Exit: ...
  ```

Assembler calculates addresses
RISC vs CISC
CISC Approach

- Complex Instruction Set Computer
- C code:
  \[ g = h + A[8]; \]
- CISC
  \[ \text{add a,32}<b> \]
- Achieved by building complex hardware that loads value from memory into a register and then adds it to register a and stores the results in register a
CISC vs RISC

- C code:
  \[ g = h + A[8]; \]

- CISC
  \[ \text{add a,32\textless b\textgreater} \]

- Compiled MIPS code:
  \[
  \text{lw}\quad $t0, 32($s3) \quad \# \text{ load word} \\
  \text{add}\quad $s1, $s2, $t0
  \]
CISC Advantages

- Compiler has to do little
  - Programming was done in assembly language
  - To make it easy, more and more complex instructions were added
- Length of the code is short and hence, little memory is required to store the code
  - Memory was a very costly real-estate
- E.g., Intel x86 machines powering several million desktops
RISC Advantages

- Each instruction needs only one clock cycle
- Hardware is less complex
RISC Roadblocks

- RISC processors, despite their advantages, took several years to gain market
  - Intel had a head start of 2 years before its RISC competitors
  - Customers were unwilling to take risk with new technology and change software products
  - They have a large market and hence, they can afford resources to overcome complexity
Fallacies

- Powerful instruction $\Rightarrow$ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions

- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code $\Rightarrow$ more errors and less productivity
Fallacies

- Backward compatibility $\Rightarrow$ instruction set doesn’t change
  - But they do accrete more instructions
Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
CISC vs RISC

- Very good slides on the topic
  - [https://www.cs.drexel.edu/~wmm24/cs281/lectures/pdf/RISCvsCISC.pdf](https://www.cs.drexel.edu/~wmm24/cs281/lectures/pdf/RISCvsCISC.pdf)
Patterson’s blog:

Interview with Hennessy
- [http://www-cs-aculty.stanford.edu/~eroberts/courses/soco/projects/risc/about/interview.html](http://www-cs-aculty.stanford.edu/~eroberts/courses/soco/projects/risc/about/interview.html)
Concluding Remarks

- **Design principles**
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises

- **Layers of software/hardware**
  - Compiler, assembler, hardware

- **MIPS: typical of RISC ISAs**
  - c.f. x86