Chapter 2 & Appendix A
Instructions: Language of the Machine
Topics:

- Assembly language, assemblers
- MIPS R2000 Assembly language
  - Instruction set
  - MIPS design goals
  - Memory & registers
  - Instruction formats
  - Some MIPS instructions
- Advanced topics
  - Macros
  - Procedure calls
  - I/O
Introduction

Machine instructions:
- Language (commands) of the Machine
- Primitive (w.r.t. HLL)
  - No sophisticated control flow (Loops, If-statements)
- Restrictive
  - Arithmetic instructions operate on register operands only, two operands at a time (MIPS)
- Different m/c language dialects for different machines
- Once you learn one dialect, it is easy to learn the others
Introduction

- **Instruction set:**
  - The complete set of instructions (vocabulary) used by a machine

- **Instruction Set Architecture (ISA):**
  - An abstract interface between the hardware and the lowest-level software of a machine
  - Includes:
    - Necessary information to write correct machine-language programs
    - Specification of instructions, registers, memory size, ... etc.

- **We will concentrate on MIPS- ISA**
  - Used by NEC, Nintendo, Silicon Graphics, Sony, . . .
High-Level Language (HLL) Translation

- Compilers generate either machine language or assembly language object files.

Program → Compiler → Assembler → Linker → Computer

HLL program → Object code → Assembly language program

Program library → Executable code
Definitions

- **Assembler:**
  - A system program that translates a mnemonic assembly language into machine language (binary code / object code)

- **Macro:**
  - Allows extending assembly instructions through defining new (SW) instructions
  - Used for more structuring and information-hiding
  - Macro name can be followed by a formal argument list
  - Actual arguments can be passed to replace the formal argument list’s text

- **Macro expander**
  - Used to perform the expansion
  - The term is also used in connection with the C preprocessor, Lisp, or one of several special-purpose languages built around a macro-expansion facility
More Definitions

- Executable file:
  - Object files combined by the linker with library files

- Program library:
  - Pre-written collection of subroutines and functions stored in one or more files for linking with other programs
  - Usually in compiled form
  - Allows code reuse
  - Often supplied by the operating system or software development environment developer
  - May be general or special purpose
More Definitions

- **Linker:**
  - Converts the object files from user & library programs into executable file to form a complete executable.
  - Resolves all external & forward references
  - The linking may be static linking or, in some systems, dynamic linking (DLL)

- **Global label:**
  - Visible outside the file

- **External reference**
  - Referencing a label in another file

- **Forward reference**
  - Referencing a label that appears later in the same file
Assembly Language Translation

- How `.exe` files are generated from Assembly source file

```
Source file → Assembler → Object file

Source file → Assembler → Object file

Source file → Assembler → Object file
```

```
Executable file
Linked in the Program library

Non-executable machine language instructions
Assembly language instructions
```

Executable machine language instructions
**Object File Format**

- **Object file header**
  - Program or procedure name, Text Size, & Data size
- **Text segment**
  - Machine language instructions code (non-executable)
- **Data segment**
  - Binary data representation. Contains unresolved references
  - Identifies data & instructions that depend on absolute addresses
    - References should change if stored in a different place in memory
  - Correspondence of source program lines with instruction addresses
    - Used by the debugger
- **Relocation information**
  - Associates labels with addresses. Lists unresolved references
- **Symbol table**
  - Debugging information
Assembly Language

- Symbolic representation of the machine language of a specific processor

- Advantages
  - High execution speed
  - Smaller code size

- Disadvantages:
  - Machine specific
  - Long programs
  - Less programmer productivity
  - Difficult to read, understand, & debug
  - Lacks structure
Practical Approach

- **Hybrid approach:**
  - Most of the program written in HLL
  - Critical sections written in Assembly language

- **Expansion factor:**
  - Ratio of length of assembly to HLL programs
Assemblers

- Converts assembly language into machine code
- Input:
  - Assembly language program
- Output:
  - Object file containing
    - Non-executable machine instructions
    - Data
    - Bookkeeping info
- Two phases:
  - Get locations of labels and build the symbol table
  - Translate statements into equivalent binary code
- Symbol Table
  - Used to help resolve forward & external referencing to create the object file
Translation of a C-Program Into Assembly

- Example: C-Program

```c
#include <stdio.h>
int main (int argc, char *argv[]) {
    int i;
    int sum = 0;
    for (i=0; i<= 100; i=i+1)
        sum = sum +i*i;
    printf("The sum from 0 .. 100 is %d\n", sum);
}
```
Equivalent Assembly Program (No Labels)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Assembler Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>addiu</td>
<td>$29,$29, -32</td>
<td>#add immediate unsigned</td>
</tr>
<tr>
<td>sw</td>
<td>$31,20($29)</td>
<td># $29 = $sp stack pointer</td>
</tr>
<tr>
<td>sw</td>
<td>$4,32($29)</td>
<td># $31 = $ra return address</td>
</tr>
<tr>
<td>sw</td>
<td>$5,36($29)</td>
<td>#</td>
</tr>
<tr>
<td>sw</td>
<td>$0,24($29)</td>
<td># $31 = $ra return address</td>
</tr>
<tr>
<td>lw</td>
<td>$14,28($29)</td>
<td># onto stack</td>
</tr>
<tr>
<td>lw</td>
<td>$24,24($29)</td>
<td>#</td>
</tr>
<tr>
<td>multu</td>
<td>$14,$14</td>
<td># multiply unsigned</td>
</tr>
<tr>
<td>addiu</td>
<td>$8,$14,1</td>
<td># set $1 if &lt; immediate</td>
</tr>
<tr>
<td>slti</td>
<td>$1,$8,101</td>
<td># move from lo of register</td>
</tr>
<tr>
<td>sw</td>
<td>$8,28($29)</td>
<td>#</td>
</tr>
<tr>
<td>mflo</td>
<td>$15</td>
<td># load upper immediate</td>
</tr>
<tr>
<td>addu</td>
<td>$25,$24,$15</td>
<td># jump &amp; link</td>
</tr>
<tr>
<td>bne</td>
<td>$1,$0,-9</td>
<td>#</td>
</tr>
<tr>
<td>sw</td>
<td>$25,24($29)</td>
<td>#</td>
</tr>
<tr>
<td>lui</td>
<td>$4,4096</td>
<td>#</td>
</tr>
<tr>
<td>lw</td>
<td>$5,24($29)</td>
<td>#</td>
</tr>
<tr>
<td>jal</td>
<td>1048812</td>
<td>#</td>
</tr>
<tr>
<td>addiu</td>
<td>$4,$4,1072</td>
<td># jump register</td>
</tr>
<tr>
<td>lw</td>
<td>$31,20($29)</td>
<td>#</td>
</tr>
<tr>
<td>addiu</td>
<td>$29,$29,32</td>
<td>#</td>
</tr>
<tr>
<td>jr</td>
<td>$31</td>
<td>#</td>
</tr>
<tr>
<td>move</td>
<td>$2, $0</td>
<td>#</td>
</tr>
</tbody>
</table>
Equivalent Assembly Program (Labeled)

.text
.globl main

main:
    subu $sp,$sp, 32  # increment stack by a stack frame
    sw $ra,20($sp)    # Save return address
    sd $a0,32($sp)    # pseudo-instruction (Save double-word)
    sw $0,24($sp)
    sw $0,28($sp)

loop:
    lw $t6,28($sp)   # Pseudo-instruction (Load address)
    mul $t7,$t6,$t6
    lw $t8,24($sp)
    addu $t9,$%8,$t7
    sw $t9,24($sp)
    addu $t0,$t6,1
    sw $t0,28($sp)
    bie $t0,100,loop # Jump & link
    la $a0,str
    lw $a1,24($sp)
    jal printf
    move $v0,$0
    lw $ra,20($sp)
    addu $sp,$sp,32
    jr $ra

.data
.globl str
.str: .asciiz "The sum from 0..100 is %d\n"
What Are Pseudo-Instructions?

- Instructions provided by an assembler, but not implemented in hardware
- Make assembly programming easier without complicating the hardware
- Macros can replace pseudo-instructions (Discussed later, if time allows)
Assembly vs. Machine Language

- **Assembly Language:**
  - Provides convenient symbolic representation
    - Easier than writing down numbers
    - Operands can be written in a different order than that in the instruction (e.g., destination first)
  - Provide 'pseudo-instructions'
    - Example:
      - `move $t0, $t1`
      - exists only in Assembly
      - would be implemented using
        - `add $t0,$t1,$zero`

- **Machine language:**
  - The underlying reality
    - e.g., destination is no longer first
  - When considering performance you should count real machine instructions, not pseudo-instructions
Linker & Loader

- **Linker**
  - Combines object files from separate modules
  - Resolves references among files
  - Searches for library routines needed by the program
  - Determines memory locations needed by each module
  - Adjusts absolute references accordingly

- **Loader**
  - Loads program into memory and provides actual memory references

![Diagram](image-url)
Memory is typically divided into 3 parts:

1. **Text segment:**
   - Holds program instructions

2. **Data segment**
   - **Static data:**
     - Size is known to the compiler
     - Lifetime is all program execution period
   - **Dynamic data**
     - Size is known at runtime
     - Allocated at runtime

3. **Stack segment**
   - Holds dynamic data

4. **Reserved**
   - Used by the operating system
Instruction Design

Instruction length:
- Variable length instructions:
  - Assembler needs to keep track of all instruction sizes to determine the position of the next instruction
- Fixed length instructions:
  - Require less housekeeping

Number of operands
- Depend on type of instruction
SPIM

- Software simulator for running MIPS R-Series processors’ programs
- Why use a simulator?
  - MIPS workstations
    - Not always available
    - Difficult to understand & program
  - Simulator
    - Better programming environment
    - Provide more features
    - Easily modified
- See Britton, online links, and Appendix A for more information about SPIM
MIPS Processors

- 3 processors:
  - Main processor:
    - For integer arithmetic
  - Coprocessor 0:
    - For exceptions, interrupts, & virtual memory system
  - Coprocessor 1:
    - For floating-point arithmetic

- SPIM Simulator simulates most of the functions of the above processors
MIPS Processors

- CPU
  - Registers
    - $0
    - $31
  - Arithmetic Unit
  - Multiply divide
  - Lo
  - Hi

- Coprocessor1 (FPU)
  - Registers
    - $0
    - $31
  - Arithmetic Unit

- Coprocessor 0 (traps & memory) registers
  - BadVAddr
  - Status
  - Cause
  - EPC

- Memory
MIPS Processors

- **Addressing modes:**
  - Describe the manner in which addresses for memory accesses are constructed
  - MIPS is a “Load-Store” architecture
    - Only load/store instructions access memory
  - Data should be aligned (usually multiple of 4 bytes)
  - More details in [MIPS Quick Reference](#)
Addressing Modes

- Register addressing
  - Operand is a register
  - Value is the contents of the register
- Base or displacement addressing
  - Operand is at the memory location whose address is the sum of a register and a constant in the instruction
- Immediate addressing
  - Operand is a constant within the instruction itself
- PC-relative addressing
  - Address is the sum of the PC and a constant in the instruction
- Pseudo-direct addressing
  - Jump address is the 26 bits of the instruction concatenated with the upper bits of the PC
Addressing Modes

1. Immediate addressing
   \[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{Immediate} \]

2. Register addressing
   \[ \begin{array}{c}
   \text{op} \quad \text{rs} \quad \text{rt} \quad \text{rd} \quad \ldots \quad \text{funt} \\
   \end{array} \rightarrow \text{Registers} \rightarrow \text{Register} \]

3. Base addressing
   \[ \begin{array}{c}
   \text{op} \quad \text{rs} \quad \text{rt} \quad \text{Address} \\
   \end{array} \rightarrow \text{Memory} \rightarrow \text{Byte} \quad \text{Halfword} \quad \text{Word} \rightarrow \text{Register} \]

4. PC-relative addressing
   \[ \begin{array}{c}
   \text{op} \quad \text{rs} \quad \text{rt} \quad \text{Address} \\
   \end{array} \rightarrow \text{Memory} \rightarrow \text{Word} \rightarrow \text{PC} \]

5. Pseudodirect addressing
   \[ \begin{array}{c}
   \text{op} \quad \text{Address} \\
   \end{array} \rightarrow \text{Memory} \rightarrow \text{Word} \rightarrow \text{PC} \]
Memory Organization

- Memory is viewed as a large, single-dimensional array
- To access a word, memory address is supplied by instruction
- Memory address is an index to the array, starting at 0
Byte & Word Addressing

- Index points to a byte of memory
- Most data items use "words"
- Words are aligned at word boundaries
  - For MIPS, a word is 32 or 64 bits
  - They are usually called MIPS =32 & MIPS =64 respectively
  - We will only consider MIPS32
  - MIPS32 (4 bytes) can access
    - $2^{32}$ bytes with byte addresses from 0 to $2^{32}-1$, or
    - $2^{30}$ words with byte addresses 0, 4, 8, ..., $2^{32}-4$
Designing Instructions

- When designing an ISA, need to consider
  - Instruction length:
    - Variable length instructions:
      - Assembler needs to keep track of all instruction sizes to determine the position of the next instruction
    - Fixed length instructions:
      - Require less housekeeping
  - Number of operands
    - Depend on type of instruction
MIPS Instruction Formats

- MIPS has 3 instruction formats
  - R-type (Register) format
  - J-type (Jump) format
  - I-type (Immediate) format
- All MIPS instruction formats are 32 bits long
  - Example: `add $t0, $s1, $s2`
- Registers can be written in their symbolic or numeric forms
  - `$t0=8, $s1=17, $s2=18`
R-Format (Register) Instructions

- **op:** Operation code (6-bits)
- **rs:** 1\(^{st}\) source register (5-bits)
- **rt:** 2\(^{nd}\) source register (5-bits)
- **rd:** Destination register (5-bits)
- **shamt:** Shift amount (5-bits)
- **funct:** Function code (6-bits)

- The first (**op**) & last fields (**funct**), combined, indicate the type of instruction
- Second (**rs**) & third (**rt**) fields are the source operands
- Fourth field (**rd**) is the destination operand

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>18</td>
<td>8</td>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>

| 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>
## Registers Names & Numbers

<table>
<thead>
<tr>
<th>Name</th>
<th>Register#</th>
<th>Usage</th>
<th>Preserved on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>values for results &amp; expr. evaluation</td>
<td>no</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
<td>yes</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
<td>yes</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
SPIM

- Software simulator for running MIPS R-Series processors’ programs
- SPIM Simulator simulates most of the functions of three MIPS processors
- More about SPIM will be discussed in the labs
- Why use a simulator?
  - MIPS workstations
    - Not always available
    - Difficult to understand & program
  - Simulator
    - Better programming environment
    - Provide more features
    - Easily modified
MIPS Assembly Language Syntax

- Comments
  - Starts with *the sharp (pound) sign* `#`
  - Until end of line
- Identifiers
  - Alphanumeric, '_', and '.'
  - Can’t start with a digit
- Labels
  - At beginning of line, followed by `:`
MIPS Assembly Language Syntax

- **Numbers**
  - Decimal (default)
  - Hexadecimal
    - Start with “0x:”

- **Strings**
  - Enclosed in double quotes ‘ ” ’

- **Special characters**
  - `\n` new line
  - `\t` tab
  - `"` quote
  - . . .
MIPS Assembler Directives

- Used to give information to the assembler
- Starts with a period
- Describe data or references
- Examples:

  `.align n` # Align data on $2^n$ byte boundary
  `.ascii str` # Store strings without null termination
  `.asciiz str` # Store null-terminated string
  `.byte b1, ... , bn` # Store in successive bytes
  `.data <addr>` # Store in data segment
  `.double dl1, ..., dn` # Store floating-pt.double-precision
  `.extern sym size` # External symbol with ‘size’ bytes
  `.float fl1, ..., fn` # Store floating-pt. single precision
  `.globl sym` # sym can be referenced from other files
  `.half h1, ... , hn` # Half-words in successive memory
  `.space n` # Allocate n-bytes of space in data seg.
  `.text <addr>` # Put text items in text segment
  `.word w1, ... , wn` # Full word in successive memory
Assembler Directives Examples

`.ascii "The sum from 0..10 is %d\n"

Is equivalent to:

`byte 84, 104, 101, 32, 115, 117, 109, 32`
`byte 102, 114, 111, 109, 32, 48, 32, 46`
`byte 46, 32, 49, 48, 32, 105, 115, 32`
`byte 37, 100, 10, 0`

Note:
- 10 is ASCII code for line feed
- 0 is ASCII code for the NULL character

For more Assembler directives (see [MIPS Quick Reference](#))
MIPS Design

Goals
- Maximize performance
- Minimize cost
- Reduce design time

How can we reach these goals?

Principles
1. Simplicity favors regularity
2. Smaller is faster
3. Good design demands good compromises
4. Make common case fast
Principle 1: Simplicity Favors Regularity

- All instructions have the same size
- Each instruction has exactly 3 operands
  - 2 source operands
  - 1 destination operand
- Registers are used instead of variable memory locations to speed-up operations
- Register fields are always in the same place
  - Operand order is fixed (destination first, except for store
    Arithmetic instructions perform only one operation
- Each line contains at most one instruction
Principle 1: Simplicity Favors Regularity

- Effective use of registers is the key to program performance
- Each HLL statement breaks into several MIPS instructions
- Temporary variables are used for intermediate results
- Operands must be registers
- Only 32 registers provided (32-bit each)
- 2-char names for registers, following $ sign

Terminology:
- Temporary values stored in registers with prefix "$t"
- Memory (Storage) variables stored in registers with prefix "$s"
Principle 1: Simplicity Favors Regularity

- Three operands keeps the instruction logically simple

Examples:

C Code | Assembly Equivalent
--- | ---
A = b + c | add a, b, c
b = x * y | mul b, x, y
a = b + 42 | addi a, b, 42
Principle 1: Simplicity Favors Regularity

- A more complex example
  
  **C Code**
  
  \[ f = (g+h) - (i+j) \]
  
  **Assembly equivalent**
  
  `add $t0, g, h`
  
  `add $t1, i, j`
  
  `sub f, $t0, $t1`

- Notes
  
  - Consider the operator precedence
    
    - `( )` before `-`
  
  - This is a pseudo code
    
    - Cannot use the symbols `g` and `h`
    
    - Values should exist in some registers, then use register names or numbers
Principle 1:
Simplicity Favors Regularity

More Examples:
Using variable names

C code:
\[
A = B + C + D + E
\]

MIPS pseudocode:
\[
\begin{align*}
\text{add } & A, B, C & \quad \# \text{ add } B + C, \text{put result into } A \\
\text{add } & A, A, D & \quad \# \text{ put } B + C + D \text{ into } A \\
\text{add } & A, A, E & \quad \# \text{ put } B + C + D + E \text{ into } A
\end{align*}
\]

Syntax:
\[
\text{add rd, rs, rt} \quad \# \text{ destination, source1, source2}
\]

Exercise
Assume that A, B, C, D, & E are stored in registers $s0, \ldots, s4$, rewrite the code using registers’ names
Principle 1:
Simplicity Favors Regularity

More Examples:
- Using symbolic register names
  - C code:
    
    \[
    A = B + C + D; \\
    E = F - A; \\
    \]
  
  - MIPS code:
    
    \[
    add \; $t0, \; $s1, \; $s2 \\
    add \; $s0, \; $t0, \; $s3 \\
    sub \; $s4, \; $s5, \; $s0 \\
    \]
Principle 2: Smaller is Faster

- Only 32 registers are used
- What happens if we increase number of registers?
  - Allow more variables => faster
  - Increase length of clock cycle
    - Electronic signals take longer to travel farther
    - => slower
- We need to compromise
  - Use limited number of registers
  - Make effective use of the registers
  - Limit memory access to increase speed
  - Only data transfer instruction will access memory:
    - lw
    - sw
  - To access a word, memory address is needed
Principle 2: Smaller is Faster

- Why are registers faster?
- Where are the registers?
Principle 2: Smaller is Faster

- Data access is faster in register
- Data is more useful when in registers
  - Arithmetic deals with more than one register at a time
  - Data access retrieves one operand at a time, without operating on it
  => Registers have less access time & higher throughput
- MIPS compilers must use registers efficiently
Principle 2: Smaller is Faster

- What about programs with more variables than available registers or complex data structures?
  - **Spilling registers**: Most frequently used variables are kept, the rest are transferred back to memory.
  - **Arrays accessed via index** to reach different elements during run-time.
  - **Offset & base** register used for structures as well as arrays.
**Principle 2: Smaller is Faster**

- Data transfer instructions are used to transfer data between registers and memory
- They must supply a memory address
- Example
  - C Code
    - \( g = h + A[8] \)
  - Assumptions
    - Register \( s3 \) contains the base address of array \( A \)
    - 8 is the offset of the 8th element of the array
  - MIPS equivalent
    - \( \text{lw } t0, 8(s3) \)  # Temporary register \( t0 \) gets \( A[8] \)
    - \( \text{add } s1, s2, t0 \)  # \( g = h + A[8] \)
Principle 2: Smaller is Faster

- Alignment restriction
- In MIPS words must start at addresses that are multiples of 4
- Some computers use the address of the leftmost byte "**Big end**" and some computers use the rightmost byte "**Little end**"
- MIPS uses the Big Endian
  - See also
    - [Webopedia](#)
Principle 3: Make Common Case Faster

- Include PC-Relative addressing for conditional branches and immediate addressing for constant operands
- MIPS immediate instructions:
  - `addi $29, $29, 4` # add immediate
  - `slti $8, $18, 10` # set if less than immediate
  - `andi $29, $29, 6`
  - `ori $29, $29, 4`

- How could we implement this?
  - Make the constant as part of the instruction
    - Much faster than if they were loaded from memory
    - Constants are usually short to fit into the 16-bit field
Principle 3: Make Common Case Faster

Large constants Manipulation:

- How can we load a 32 bit constant into a register
  - Use two instructions one to load the upper half & another to load the lower half of the register

New instruction:

- **lui** (load upper immediate) instruction
  
  \[
  \text{lui } \$t0, 1010101010101010
  \]

- \$t0 filled with zeros

\[
\begin{array}{c}
1010101010101010 \\
0000000000000000 \\
\end{array}
\]
Principle 3:  
Make Common Case Faster

- Large constants Manipulation:
  - New instruction:
    - ori (or-immediate) instruction
    - Get the lower order bits right
      \[
      \text{ori } t0, t0, 1010101010101010
      \]

original \( t0 \) value

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

immediate value

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

ori

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

Final result in \( t0 \)
Principle 4:
Good Design Demands Compromise

- Compromise between providing for larger addresses & constants in instruction and keeping all instructions the same length
- Addresses needs more than 5-bits
  - Introduce a new type of instruction format for data transfer instructions (I-format)
  - We have two options:
    - Change instruction length for different types of instructions, or
    - Keep instruction length & change field format
  - Example: \( \text{lw} \ \$t0, 32(\$s2) \)

\[
\begin{array}{cccc}
35 & 18 & 8 & 32 \\
\text{op} & \text{rs} & \text{rt} & 16 \text{ bit address}
\end{array}
\]
Principle 4: Good Design Demands Compromise

- \textit{lw} instruction can load words within (\(+/-\) ) $2^{15}$ immediately
- The meaning of the field ($rt$) changes:
  - for \textit{lw}: destination register
  - for \textit{sw}: source register
- Each format is assigned a set of values of the op-field from which it recognizes how to treat the instruction (R- or I-format type) and how many operands are involved
Principle 4:
Good Design Demands Compromise

- Constants & Immediate Operands:
  - Small constants are used frequently
    - 50% of operands in arithmetic operations
    - Also used in comparisons
  - Examples
    
    \[
    \begin{align*}
    A & = A + 5; \\
    B & = B + 1; \\
    C & = C - 18; \\
    \text{if} \ (x > 10) & \ldots
    \end{align*}
    \]
Principle 4:  
Good Design Demands Compromise

- How should they be manipulated?
  - Put 'typical constants' in memory and load them?
    - Slow
  - Create hard-wired registers (like $zero) for constants like one
    - Can’t be generalized for all constants
  - Use immediate operand inside the instruction
    - More practical, might need new format, or might use an already existing format type
Control Flow Instructions

- The ability to make decisions
- Change the control flow (i.e., "next" instruction to be executed)
- Types:
  - Conditional
  - Unconditional
- See Appendix for more comparison & branch instructions
- In high-level languages, you don’t have to write explicit labels
- Compilers create branches & labels that don’t appear in the HLL
Unconditional Branches

Forms:

- $j$ \textbf{label} # jump to label
- $jr$ \textbf{rs} # jump to addr stored in register
Conditional Branches

- Forms:
  - \texttt{beq} (Branch on equal)
  - \texttt{bne} (Branch on not equal)
  - \texttt{slt} (Set on less than)

- Examples:
  \begin{align*}
    \texttt{bne} & \quad \texttt{$t0,$t1,L} & \# \text{ go to L if } \texttt{$t0 \neq$t1} \\
    \texttt{beq} & \quad \texttt{$t0,$t1,L} & \# \text{ go to L if } \texttt{$t0 =$t1} \\
    \texttt{slt} & \quad \texttt{$t0,$t1,$t2} & \# \texttt{$t0 =1 if } \texttt{$t1<$t2, $t0=0 otherwise}
  \end{align*}
More Control Flow Instructions

- Branch-if-less-than

\[\text{slt } \$t0, \$s1, \$s2 \Leftrightarrow \text{if } \$s1 < \$s2 \text{ then} \]
\[\$t0 = 1\]
\[\text{else} \]
\[\$t0 = 0\]

- We can use this instruction to build

\[\text{blt } \$s1, \$s2, \text{Label}\]

- \textit{blt} is a pseudo-instruction meaning “branch if less than”

- We can now build general control structures

- Note that the assembler needs a register to do this
From C to MIPS – Array Manipulation

- Arrays with Constant Index
  - Equivalent MIPS code:
    - Assumptions:
      - $s3$ contains starting address of the array \( A \)
      - $s2$ contains the value of \( h \)

\[
\begin{align*}
\text{lw} & \quad \text{$t0,32($s3)} \quad \# \text{$t0$ gets A[8]} \\
& \quad \# \text{offset } = 8 \times 4 = 32 \\
\text{add} & \quad \text{$t0,$s2,$t0} \quad \# \text{Add } h \\
\text{sw} & \quad \text{$t0,32($s3)} \quad \# \text{store value back in A[8]}
\end{align*}
\]
From C to MIPS – Logical Operations

- Shifts
- Bitwise AND
- Bitwise OR
- Bitwise NOR
From C to MIPS – Logical Operations

- Shifts
  - Left/right (sll, srl)
  - Can be used to represent multiplication/division for multiples of 2

- Example
  
  ```
  sll $t2, $s0, 4  # reg $t2 = reg $s0 << 4 bits
  ```
From C to MIPS – Logical Operations

- **Bitwise AND**
  - Bit by bit operation
  - Leaves a 1 in the result only if both bits of the operands are 1

- **Example:**
  - **Assumption**
    - \( t2 = \) 0000 0000 0000 0000 0000 1101 0000 0000
    - \( t1 = \) 0000 0000 0000 0000 0011 1100 0000 0000
  - **Operation**
    - \( \text{and } t0, t1, t2 \) # reg \( t0 = \) reg \( t1 \) & reg \( t2 \)
  - **Result**
    - \( t0 = \) 0000 0000 0000 0000 0000 1100 0000 0000
From C to MIPS – Logical Operations

- **Bitwise OR**
  - Bit by bit operation
  - Leaves a 1 in the result only if any bit of the operands is 1
- **Example:**
  - **Assumption**
    - $t2 = 0000 0000 0000 0000 0000 1101 0000 0000$
    - $t1 = 0000 0000 0000 0000 0011 1100 0000 0000$
  - **Operation**
    - \texttt{or $t0, t1, t2 \# reg t0 = reg t1 | reg t2$}
  - **Result**
    - $t0 = 0000 0000 0000 0000 0011 1101 0000 0000$
From C to MIPS – Logical Operations

- Bitwise NOR
  - Bit by bit operation
  - Inverse of OR

Example:

- Assumption
  \[ t2 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 0000\ 0000 \]
  \[ t1 = 0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000 \]

- Operation
  \[ \text{and } t0, t1, t2 \quad \# \text{ reg } t0 = \sim (\text{reg } t1 | \text{reg } t2) \]

- Result
  \[ t0 = 1111\ 1111\ 1111\ 1111\ 1111\ 1100\ 0010\ 1111\ 1111 \]
Exercise:

What should change in the previous MIPS code for EACH OF the following C-statements?

\[
A[300] = h + A[300];
\]
\[
A[16] = h + A[8];
\]
\[
A[i] = h + A[i];
\]

Write the equivalent machine code in each case.
From C to MIPS – Array Manipulation

Arrays with Variable Index

C code:

\[ g = h + A[i]; \]

Equivalent MIPS code

Assumption: \$s4 contains \( i \)

# Multiply index by 4 due to byte addressing
# Store the value in \$t1
add \$t1, \$s4, \$s4 # \$t1 = 2 *i
add \$t1, \$t1, \$t1 # \$t1 = 4 *i

# Base is stored in \$s3
# Get address of \( A[i] \)
add \$t1, \$t1, \$s3 #$t1=Address(A[i])

#Load \( A[i] \) into temporary register
lw \$t0, 0($t1) # \$t0 = A[i]

# Add \( A[i] \) to \( h \)
add \$s1, \$s2, \$t0 # \$s1 = h + A[i]

# \$s1 corresponds to \( g \)
From C to MIPS - If-Statement

- C-Code:
  
  ```
  if (i==j)
      h = i + j;
  ```

- Equivalent MIPS Code:
  
  ```
  bne  $s0, $s1, Label
  add  $s3, $s0, $s1
  Label: ....
  ```

- Assumptions:
  
  
  ```
  $s0 = i
  $s1 = j
  $s3 = h
  ```
### From C to MIPS - If-else statement

- **C statement**
  
  ```c
  if (i != j)
  f = g + h;
  else
  f = g - h;
  ```

- **Equivalent MIPS code:**
  ```mips
  beq $s0, $s1, Else
  add $s2, $s3, $s4
  j Exit
  Else: sub $s2, $s3, $s4
  Exit: ...
  ```

- **Assumptions:**
  
  - $s0 = i$
  - $s1 = j$
  - $s2 = f$
  - $s3 = g$
  - $s4 = h$
From C to MIPS – For Loops

C-Code

```
for (; i != h; i = i+j) g = g + a[i];
```

Equivalent MIPS code:

```
Loop:    add     $t1,$s3,$s3     # $t1 = 2 * i
       add     $t1,$t1,$t1     # $t1 = 4 * i
       add     $t1,$t1,$s5     # $t1 = addr(A[i])
       lw      $t0,0($t1)     # $t0 = A[i]
       add     $s1,$s1,$t0     #g = g +A[i]
       add     $s3,$s3,$s4     #i = i + j
       bne     $s3,$s2,Loop    #go to Loop if i≠h
```

Assumptions:

\[ \$s1 = g \quad \$s2 = h \]
\[ \$s3 = i \quad \$s4 = j \]
\[ \$s5 = \text{base address of array } A \]

Note: Check if this is not a do-while loop!!!
From C to MIPS – While Loop

- C-Code
  ```c
  while ( a[i] == k )
  i = i + j;
  ```

- MIPS Equivalent
  ```mips
  Loop:
  sll $t1, $s3, 2   # $t1 = 4 * i
  add $t1, $t1, $s6 # $t1=addr(A[i])
  lw $t0, 0($t1)  # $t0 = A[i]
  bne $t0, $s5, Exit  # goto Exit if A[i] ≠ k
  add $s3, $s3, $s4 # i=i+j
  j Loop # Loop back
  Exit: ... # Next statement
  ```

- Assumptions:
  
  - $s3 = i
  - $s4 = j
  - $s5 = k
  - $s6 = Base address of A
From C to MIPS - Less Than Test

C-Code:

```
if (a < b) goto Less;
```

Equivalent MIPS code:

```
slt $t0, $s0, $s1  # $t0=1 if $s0 < $s1
# ($s0=a, $s1=b)

bne $t0,$zero,Less  #goto Less if $t0≠0
```
From C to MIPS - Switch Statement

- The jump register \( (jr) \) instruction is used
- Unconditional jump to the address given in the register

Possibilities
- Convert it into a group of nested \texttt{if-then-else} statements
- Use a table of addresses (jump address table) for the instruction sequences and use an index to jump to the appropriate entry
From C to MIPS - Switch Statement

- **C-Code:**

  ```c
  switch(k)
  {
    case 0:  f = I + j; break; /* k=0 */
    case 1:  f = g + h; break; /* k=1 */
    case 2:  f = g - h; break; /* k=2 */
    case 3:  f = I - j; break; /* k=3 */
  }
  ```

- **Steps:**
  1. Check that k is within limits, otherwise exit
  2. From k, find out where to jump to (using index table)
  3. After statement execution, jump to Exit label (break)
From C to MIPS - Switch Statement

Assumptions:
$s0 = f$, $s1 = g$, $s2 = h$, $s3 = i$, $s4 = j$, $s5 = k$, $t2 = 4$

```
slt $t3, $s5, $zero  # test if k<0 ($s5=k)
bne $t3, $zero, Exit # go to Exit if k <0
slt $t3, $s5, $t2    # Test if k<4, $t2=4
beq $t3, $zero, Exit # go to Exit if k>=4
add $t1, $s5, $s5   # $t1 = 2*k
add $t1, $t1, $t1    # $t1=4*k=jump address
add $t1, $t1, $t4   #$t1=addr(JumpTable[k])
lw  $t0, 0($t1)     # $t0=JumpTable[K]
jr  $t0

L0: add $s0, $s3, $s4    # k=0 => f=i+j
    j   Exit
L1: add $s0, $s1, $s2  # k=1 => f=g+h
    j   Exit
L2: sub $s0, $s1, $s2  # k=2 => f=g-h
    j   Exit
L3: sub $s0, $s3, $s4  # k=3 => f=i-j
Exit: ...
```
Input/Output

- We are not going to discuss MIPS I/O instructions, except what is necessary to display messages on the console window.
- See examples.
### MIPS I/O Example

```assembly
# Comments start with the #-character
# Labels are terminated with a colon ':'
# System commands begin with a period '.'
# PCSpim has a flaw.
# It should be terminated with an empty line
# Start with the data segment
.data
   First: .asciiz "Enter first number: "
   Second: .asciiz "Enter second number: "
   SumStr: .asciiz "First + Second = 
   Check: .asciiz "Continue? (0=No, 1 = Yes): 
   Newline: .asciiz "\n"

.align 2
.globl main
# indicating the beginning of the data segment
# indicating the starting label in the program
# Follow with the text segment
```
MIPS I/O Example - Continued

```assembly
.text
main:  la $a0, First # Load first string for printing
       li $v0,4     # Call system for printing a string
       syscall    # Perform the printing
       li $v0,5    # Tell the system that reading is required
       syscall    # Read a number
       move $t0, $v0 # Store the first number in $t0
       la $a0, Second # Load Second string for printing
       li $v0,4    # Tell the system that printing a string is required
       syscall    # Perform the printing
       li $v0,5    # Tell the system that reading is required
       syscall    # Read a number
       move $t1, $v0 # Store the second number in $t1
       add $t2, $t0, $t1 # Add the two numbers & store result in $t2
       la $a0, SumStr # Load result message string for printing
       li $v0,4    # Tell the system that printing a string is required
       syscall    # Perform the printing
       li $v0,1    # Tell the system that printing an integer is required
       move $a0, $t2 # Move the integer to $a0
       syscall    # Perform the printing
       la $a0, Newline # Load new line character for printing
       li $v0,4    # Tell the system that printing is required
       syscall    # Perform the printing

# Remember to have the extra carriage return after the last line of code
```
Another MIPS I/O Example

```assembly
#############################################################
# CS116: Computer Architecture: Program 2:             #
# Adding 2 numbers with I/O & looping according to user's request #
#############################################################
# Comments start with the #-character                   #
# Labels are terminated with a colon ':'                #
# System commands begin with a period '.'              #
# PCSpim has a flaw. It should be terminated with an empty line #
# Start with the data segment                          #

.data                                                # indicating the beginning of the data segment

First: .asciiz "Enter first number: "    # character string
Second: .asciiz "Enter second number: "   # Other types are ascii, .word, & .byte
SumStr: .asciiz "First + Second = "
Check: .asciiz "Continue? (0=No, 1=Yes): "
Newline: .asciiz "\n"

.align 2                                           # align to 2 to the power 2 (= 4) word boundary
.globl main                                       # indicating the starting label in the program

# Follow with the text segment
```
Another MIPS I/O Example - Continued

main:

.text

la $a0, First
li $v0,4
syscall
li $v0,5
syscall
move $t0, $v0
la $a0, Second
li $v0,4
syscall
li $v0,5
syscall
move $t1, $v0
add $t2, $t0, $t1
la $a0, SumStr
li $v0,4
syscall
li $v0,1
syscall
move $a0, $t2
syscall
la $a0, Newline
li $v0,4
syscall
la $a0, Check
li $v0,4
syscall
li $v0,5
syscall
bne $v0, $0, main
syscall

# Remember: have extra carriage return after the last line of code
Procedure Calls

Execution of a procedure follows 6 steps

1. Place parameters in a place where the procedure can access them
2. Transfer control to the procedure
3. Acquire storage resources to the procedure
4. Perform desired task
5. Place result in a place accessible by the calling program
6. Return control to the point of origin
Procedure Calls

- MIPS register convention for procedures
  - $a0-$a3: 4 arguments registers to pass parameters
  - $v0-$v1: 2 value registers to return values
  - $ra: return address register to return to point of origin
Procedure Calls

- MIPS instructions used with procedures
  - **jal**: Jump & Link
    - Jump to an address & save address of the following instruction in $ra register
  - **jr $ra**: Jump to return address
    - Jump to the address stored in $ra
Procedure Calls

- What if more than 4 arguments need to be transferred?
  - Put it onto the stack

- Stack:
  - Needs a pointer ($sp) to the most-recently allocated address, to show where the next procedure should be allocated
  - $sp grows from higher to lower address
    - Push: subtract from $sp
    - Pop: Add to $sp
Example: Leaf Procedure

- Leaf procedure doesn’t call other procedures
- C Code
  ```c
  int leaf_example (int g, int h, int I, int j)
  {
    int f;
    f = (g + h) - (I + j)
    return f;
  }
  ```
Example: Leaf Procedure

**MIPS Equivalent**

```
addi $sp, $sp, -12  # adjust stack to make room for 3 items
sw $t1, 8($sp)     # save $t1 on stack
sw $t0, 4($sp)     # save $t0 on stack
sw $s0, 0($sp)     # save $s0 on stack
add $t0, $a0, $a1  # $t0 contains g + h
add $t1, $a2, $a3  # $t1 contains I + j
sub $s0, $t0, $t1  # f = $t0 - $t1 = (g+h)-(I-j)
add $v0, $s0, $zero # return result to calling point = f = ($v0 = $s0+0)
lw $s0, 0($sp)      # restore $s0 for caller
lw $t0, 4($sp)      # restore $t0 for caller
lw $t1, 8($sp)      # restore $t1 for caller
addi $sp, $sp, 12   # adjust stack to delete 3 items
jr $ra               # jump back to calling routine
```
Procedure Call Frame

- Memory block associated with the call, usually saved onto stack

Includes
- Argument values
- Registers possibly modified by the procedure
- Local variables

Stack frame:
- Stack block used to hold a procedure call frame

Frame pointer ($fp$):
- Points to the first word in the frame

Stack pointer ($sp$):
- Points to last word of the frame
Procedure Calls

Before the call:

1. Pass the first 4 arguments to registers $a0-$a3. The system will take care of them.
2. Remaining arguments, if any, should be pushed onto stack.
3. Save caller-saved registers onto the stack as well, since the called function might use those registers and overwrite their contents.
4. Perform `jal` instruction
   - Jump to callee's first instruction
   - Save return address in $ra
Procedure Calls

Before execution of called procedure:
1. Allocate memory for a stack frame
2. Save callee-saved registers in the frame
3. Update frame pointer
Procedure Calls

Before returning from the procedure:
1. Place return value, if any, in $v0 register
2. Restore callee-saved registers by retrieving their saved contents from the stack
3. Pop stack frame to free the memory used by the procedure
4. Jump to the return address stored in $ra
Procedure Calls Review

Before returning from the procedure:
- $sp

Before executing the procedure:
- Saved return address
- Saved arguments registers (if any)
- Saved saves registers (if any)
- Local arrays and structures (if any)

Before the call:
- $fp

High address
- $fp

Low address
- $sp

After returning from the procedure:
- $sp
- $fp
Nested Procedures

- One procedure calls another, or calls itself (recursion)
- Example: Factorial
  - C Code
    ```c
    int fact(int n)
    {
        if (n < 1)
            return 1;
        else return (n * fact(n-1));
    }
    ```
Nested Procedures

Example: Factorial

MIPS Code

```
fact:
        addi  $sp, $sp, -8    # adjust stack for 2 items
        sw    $ra, 4 ($sp)   # save return address
        sw    $a0, 0 ($sp)   # save argument n
        slti  $t0, $a0, 1    # test for n < 1
        beq   $t0, $zero, L1 # if n >= 1, goto L1
        addi  $v0, $azero, 1 # return 1
        addi  $sp, $sp, 8    # pop 2 items off stack
        jr     $ra           # return to after jal
L1:    addi  $a0, $a0, -1 # N >=1: argument gets (n-1)
        jal   fact          # call fact with (n-1)
        lw    $a0, 0($sp)    # return from jal: restore argument n
        lw    $ra, 4 ($sp)   # restore return address
        addi  $sp, $sp, 8    # adjust stack pointer to pop 2 items
        mul   $v0, $a0, $v0  # return n * fact (n-1)
        jr     $ra           # return to caller
```
Procedure Calls Example

- Example: Factorial
  - Main calls Fact(10)
  - Stack frame during call of \textbf{fact(7)}
Allocating New Data on Stack

- Stack is used to store variables local to the procedure that don’t fit in registers.
- Some MIPS software use frame pointer $fp to point to the first word of the frame of a procedure to allow reference for local variables.
- $fp offers a stable base register within a procedure for local memory reference.
Allocating Space on the Heap

- **0000 0000_{hex}:**
  - First part of the low end is reserved by the system

- **0040 0000_{hex}:**
  - Followed by the text segment

- **1000 0000_{hex}:**
  - Static data are above the text segment used for constants & other static variables

- **1000 8000_{hex}:**
  - Heap hosts dynamic data structures (e.g. linked lists)
  - Stack starts in high-end of memory & grows down
  - Stack & heap grow in opposite directions
Communicating with People

- **ASCII** Characters:
  - 8 bits
  - See Fig. 2.21, p. 91

- **UNICODE**:
  - 16 bits
3 Choices for String Representation

- First position of string gives its length
- Accompanying variable has the length of the string
- Last position has end of string character
  - Used by C, Java, & MIPS
  - Null-terminating string
Byte/Word Addressing

- Byte addressing instructions
  \[ \text{lb} \ $t0, \ 0($sp) \quad \# \text{read byte (8 bits) from source} \]
  \[ \text{sb} \ $t0, \ 0($sp) \quad \# \text{write byte to destination} \]

- Half-word addressing instructions
  \[ \text{lh} \ $t0, \ 0($sp) \quad \# \text{read half word (16 bits) from source} \]
  \[ \text{sh} \ $t0, \ 0($sp) \quad \# \text{write half word to destination} \]
String Copy Procedure

C Code

```c
void strcpy (char x[], char y[])
{
    int i;
    i = 0;
    While ((x[i] = y[i]) != '\0')  /* copy & test byte */
        i += 1;
}
```

MIPS Code

```assembly
Strcpy:
    addi $sp, $sp, -4  # adjust stack for 1 more item
    sw $s0, 0($sp)     # save $s0
    add $s0, $zero, $zero  # i = 0 + 0
    L1: add $t1, $s0, $a1  # address of y[i] in $t1
        lb $t2, 0($t1)  # $t2 = y[i]
        add $t3, $s0, $a0  # address of x[i] in $t3
        sb $t2, 0($t3)  # x[i] = x[i]
        beq $t2, $zero, L2  # if y[i] == 0, go to L2
        addi $s0, $s0, 1  # i = i + 1
        j L1  # go to L1
    L2: lw $s0, 0($sp)  # y[i] == 0: end of string; restore old $s0
        addi $sp, $sp, 4  # pop 1 word off stack
        jr $ra  # return
```
32-Bit Immediate Addressing

See Principle 3
Review - Loading Programs for Execution

1. Determine **size** of text & data segments from executable file header
2. Create enough **address space** for program’s text & data segments, in addition to a “stack segment”
3. **Copy** both **instruction** & **data** segments into address space
4. **Copy arguments** onto stack
5. **Initialize Instruction register** & **stack pointer**
6. **Copy arguments** from stack to **registers**
7. **Call** program’s main routine
8. When returning from main program, terminate with exit system call
Review - MIPS instruction Formats

- Simple instructions all 32 bits wide
- Very structured
- Addresses are not 32 bits
- Only three instruction formats

<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>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>16 bit address</td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26 bit address</td>
</tr>
</tbody>
</table>
Review - Branch instructions

\[ \text{bne } \$t4,\$t5,\text{Label} \quad \# \text{ Next instruction is at Label if } \$t4 \neq \$t5 \]
\[ \text{beq } \$t4,\$t5,\text{Label} \quad \# \text{ Next instruction is at Label if } \$t4 = \$t5 \]

- **Formats:**

  \[
  \begin{array}{c|c|c|c}
  \hline
  \text{op} & \text{rs} & \text{rt} & \text{16 bit address} \\
  \hline
  \end{array}
  \]

- We could specify a register (like \textit{lw} and \textit{sw}) and add it to address
  - Most branches are local (\textit{principle of locality})
  - Use Instruction Address Register (PC = program counter)
- Jump instructions just use high order bits of PC
  - address boundaries of 256 MB
Review - Addressing

1. Register addressing:
   - Operands are registers

2. Base (Displacement addressing):
   - Operand location = register + constant (offset) in the instruction

3. Immediate addressing:
   -Operand is a constant within the instruction

4. PC-relative addressing:
   - Address = PC (program counter) + constant in the instruction

5. Pseudo addressing:
   - Jump address = 26 bits of the instruction + upper bits of the PC

   A single operation can use more than one addressing mode (e.g. add, addi)
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