PH Chapter 2 Pt A

Instructions: MIPS ISA

Based on Text: Patterson Henessey

Publisher: Morgan Kaufmann

Edited by Y.K. Malaiya for CS470

Acknowledgements to V.D. Agarwal and M.J. Irwin
Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets
Designing a Computer

**FIVE PIECES OF HARDWARE**

- **Control**
- **Datapath**
  - Central Processing Unit (CPU) or "processor"
- **Input**
- **Memory**
- **Output**
Start by Defining ISA

- What is instruction set architecture (ISA)?
- ISA
  - Defines registers
  - Defines data transfer modes (instructions) between registers, memory and I/O
  - There should be *sufficient* instructions to efficiently translate any program for machine processing
- Next, define instruction set format – binary representation used by the hardware
  - Variable-length vs. fixed-length instructions
Types of ISA

- Complex instruction set computer (CISC)
  - Many instructions (several hundreds)
  - An instruction takes many cycles to execute
  - Example: Intel Pentium

- Reduced instruction set computer (RISC)
  - Small set of instructions
  - Simple instructions, each executes in one clock cycle – *almost*.
  - Effective use of pipelining
  - Example: ARM
MIPS: A RISC processor

RISC evolution

- The IBM 801 project started in 1975
  - Precursor to the IBM RS/6000 workstation processors which later influenced PowerPC
- The Berkeley RISC project started by Dave Patterson in 1980
  - Evolved into the SPARC ISA of Sun Microsystems
- The Stanford MIPS project started by John Hennessy ~1980
  - Hennessy co-founded MIPS Computer
- RISC philosophy: instruction sets should be simplified to enable fast hardware implementations that can be exploited by optimizing compiler
Original RISC view

- Fixed-length (32 bits for MIPS) instructions that have only a few formats
  - Simplifies instruction fetch and decode
  - Code density is sacrificed: Some bits are wasted for some instruction types
- Load-store/ Register-register architecture
  - Permits very fast implementation of simple instructions
  - Easier to pipeline (Chapter 6)
  - Requires more instructions to implement a HLL program
- Limited number of addressing modes
  - Simplifies EA calculation and thus speeds up memory access
- Few complex arithmetic functions
  - Instead more, simpler instructions are used
Pipelining of RISC Instructions

Although an instruction takes five clock cycles, one instruction can be completed every cycle.
The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See MIPS Reference Data tear-out card, and Appendixes B and E
MIPS Instruction Set (RISC)

- Instructions execute simple functions.
- Maintain regularity of format – each instruction is one word, contains opcode and arguments.
- Minimize memory accesses – whenever possible use registers as arguments.
- Three types of instructions:
  - Register (R)-type – only registers as arguments.
  - Immediate (I)-type – arguments are registers and numbers (constants or memory addresses).
  - Jump (J)-type – argument is an address.
Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination
  
  \[
  \text{add } a, b, c \quad # \quad a \text{ gets } b + c
  \]

- All arithmetic operations have this form

**Design Principle 1:** Simplicity favours regularity

- Regularity makes implementation simpler
- Simplicity enables higher performance at lower cost
Arithmetic Example

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

- **Compiled MIPS code:**
  
  ```mips
  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

- Arithmetic instructions use register operands
- MIPS has a $32 \times 32$-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- Assembler names
  - $t0, t1, \ldots, t9$ for temporary values
  - $s0, s1, \ldots, s7$ for saved variables
- Design Principle 2: Smaller is faster
  - c.f. main memory: millions of locations
Register Operand Example

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

  - \( f, \ldots, j \) in \( s0, \ldots, s4 \)

- **Compiled MIPS code:**

  - `add $t0, $s1, $s2`
  - `add $t1, $s3, $s4`
  - `sub $s0, $t0, $t1`
Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address
Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual bytes in memory
  - Alignment restriction - the memory address of a word must be on natural word boundaries (a multiple of 4 in MIPS-32)

- Big Endian: leftmost byte is word address
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

- Little Endian: rightmost byte is word address
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

```
little endian byte 0

msb  |  3  |  2  |  1  |  0  |

lsb  |  0  |  1  |  2  |  3  |

big endian byte 0
```
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:
  - Index 8 requires offset of 32
    - 4 bytes per word
  ```
lw  $t0, 32($s3)    # load word
add  $s1, $s2, $t0
  ```
  - Offset: 32
  - Base register: $s3
Memory Operand Example 2

- C code:
  \[ A[12] = h + A[8]; \]
  - \( h \) in \( $s2 \), base address of \( A \) in \( $s3 \)

- Compiled MIPS code:
  - Index 8 requires offset of 32
  - \[
    \begin{align*}
    &\text{lw} \quad \$t0, \quad 32(\$s3) \quad \# \text{ load word} \\
    &\text{add} \quad \$t0, \quad \$s2, \quad \$t0 \\
    &\text{sw} \quad \$t0, \quad 48(\$s3) \quad \# \text{ store word}
    \end{align*}
  \]
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
  addi $s3, $s3, 4

- No subtract immediate instruction
  - Just use a negative constant
    addi $s2, $s1, -1

- Design Principle 3: Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction
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
Aside: MIPS Register Convention

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
<th>Preserve on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>constant 0 (hardware)</td>
<td>n.a.</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2-3</td>
<td>returned values</td>
<td>no</td>
</tr>
<tr>
<td>$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 - $s7</td>
<td>16-23</td>
<td>saved values</td>
<td>yes</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$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 addr (hardware)</td>
<td>yes</td>
</tr>
</tbody>
</table>
Mr. Hennessy, an engineer who co-founded a semiconductor company, has used his talents, Silicon Valley connections and academic position to help win billions of dollars for Stanford. He has done well for himself, too.

Mr. Hennessy's November haul included a $75,000 retainer from Cisco Systems Inc., on whose board he sits, plus $133,000 in restricted Cisco stock, proceeds of $452,000 from selling stock in Atheros Communications Inc., where he is co-founder and chairman, and a $384,000 profit from the exercise of Google Inc. stock options. He sits on Google's board.
A Conversation with John Hennessy and David Patterson

**DP:** I got a really great compliment the other day when I was giving a talk. Someone asked, “Are you related to the Patterson, of Patterson and Hennessy?” I said, “I’m pretty sure, yes, I am.” But he says, “No, you’re too young.” So I guess the book has been around for a while.

**JH:** Another thing I’d say about the book is that it wasn’t until we started on it that I developed a solid and complete quantitative explanation of what had happened in the RISC developments. By using the CPI formula

\[
\text{Execution Time/Program} = \frac{\text{Instructions/Program}}{\text{Clocks/Instruction}} \times \frac{\text{Time/Clock}}{}
\]

we could show that there had been a real breakthrough in terms of instruction throughput, and that it overwhelmed any increase in instruction count.