MIPS Instruction Set Architecture (ISA)

Gabriel Hugh Elkaim
MAGIC NUMBER IS:
CISC vs. RISC

CISC: Complex Instruction Set Computing
- Lots of instructions of variable size
- Very memory optimized
- Very few registers

RISC: Reduced Instruction Set Computing
- Less instructions, all of a fixed width
- Optimized for speed
- Load/store architecture
What is “Modern”

CISC  RISC  RISCy  CISC
“Hello World” (K&R)

- The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:
  
  *Print the words*
  
  *hello, world*

- This is a big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went.

- With these mechanical details mastered, everything else is comparatively easy.
Hello World (1.3)

```
# Daniel J. Ellard -- 02/21/94 Modified by Max Dunne 09/27/17
# hello.asm-- A "Hello World" program.
# Registers used:
#   $v0 - syscall parameter and return value.
#   $a0 - syscall parameter-- the string to print.
.text
main:
    la $a0, hello_msg  # load the addr of hello_msg into $a0.
    li $v0, 4          # 4 is the print_string syscall.
    syscall
    li $v0, 10         # do the syscall.
    syscall
    li $v0, 10         # 10 is the exit syscall.
    syscall

# Data for the program:
.data
hello_msg: .asciiz "Hello World CMPE012L

# end hello.asm
```
Hello World (2.3)

# Daniel J. Ellard -- 02/21/94 Modified by Max Dunne 09/27/17
# hello.asm -- A "Hello World" program.
# Registers used:
#     $v0 - syscall parameter and return value.
#     $a0 - syscall parameter-- the string to print.
.text
main:
    la $a0, hello_msg
    li $v0, 4
    syscall
    li $v0, 10
    syscall

# Data for the program:
.data
hello_msg:.asciiz "Hello World CMPE012L\n"

# end hello.asm
# Hello World (3.3)

# Daniel J. Ellard -- 02/21/94 Modified by Max Dunne 09/27/17
# hello.asm-- A "Hello World" program.
# Registers used:
#   $v0 - syscall parameter and return value.
#   $a0 - syscall parameter-- the string to print.
.text
main:
  la $a0, hello_msg  # load the addr of hello_msg into $a0.
  li $v0, 4         # 4 is the print_string syscall.
  syscall           # do the syscall.
  li $v0, 10        # 10 is the exit syscall.
  syscall           # do the syscall.

# Data for the program:
.data
hello_msg: .asciiz "Hello World CMPE012L\n"

# end hello.asm
Instruction Set Architecture (ISA)

Programmer visible part of computer & operating system

- Memory organization (address space)
- Register set - how many, what size, how can they be used

- Instruction set
  - Operands (what commands I can actually give to the computer)
  - Data types
  - Addressing modes
MIPS Architecture

- 32 bit words
- 32 32-bit wide registers
- Byte addressable memory
- Many opcodes — small subset
  \[ 2^{10} = \text{Kilo} \]
  \[ 2^{20} = \text{Mega} \]
  \[ 2^{30} = \text{Giga} \]
  \[ 2^{40} = 4 \text{TB} \]

Amps nuts completes
Memory vs. Registers

Memory - Address Space
  - Addressability - byte (8 bits)
  - Address space - \(2^{32}\) 32 bit addresses

Registry
  - temporary storage, accessed in a single clock cycle
    - \(32 = 0 \ldots 31\)
  - specific uses, specific purposes
    - others not directly accessible
      - (PC, SP, registers like multiply)
Access Time of Storage

- SSD
- Cache
- RAM

[Diagram of storage devices]
More on Memory

- Memory is NOT directly manipulated
  - First access it (copy it to register)
  - Manipulate it
  - Write back to register

- text - data - either read or read only
- I/O - read only memory (Flash)
Table 1: Register Conventions

<table>
<thead>
<tr>
<th>CPU Register</th>
<th>Symbolic Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>zero</td>
<td>Always 0 (note 1)</td>
</tr>
<tr>
<td>r1</td>
<td>at</td>
<td>Assembler Temporary</td>
</tr>
<tr>
<td>r2 - r3</td>
<td>v0-v1</td>
<td>Function Return Values</td>
</tr>
<tr>
<td>r4 - r7</td>
<td>a0-a3</td>
<td>Function Arguments</td>
</tr>
<tr>
<td>r8 - r15</td>
<td>a0-a7</td>
<td>Temporary – Caller does not need to preserve contents</td>
</tr>
<tr>
<td>r16 - r23</td>
<td>s0-s7</td>
<td>Saved Temporary – Caller must preserve contents</td>
</tr>
<tr>
<td>r24 - r25</td>
<td>t8 - t9</td>
<td>Temporary – Caller does not need to preserve contents</td>
</tr>
<tr>
<td>r26 - r27</td>
<td>k0 - k1</td>
<td>Kernel temporary – Used for interrupt and exception handling</td>
</tr>
<tr>
<td>r28</td>
<td>gp</td>
<td>Global Pointer – Used for fast-access common data</td>
</tr>
<tr>
<td>r29</td>
<td>sp</td>
<td>Stack Pointer – Software stack</td>
</tr>
<tr>
<td>r30</td>
<td>s8 or fp</td>
<td>Saved Temporary – Caller must preserve contents OR Frame Pointer – Pointer to procedure frame on stack</td>
</tr>
<tr>
<td>r31</td>
<td>ra</td>
<td>Return Address (note 1)</td>
</tr>
</tbody>
</table>

Note 1: Hardware enforced, not just convention
Register File (2.2)

- Temporary $t6 ... $t9
- Procedures can destroy these (syscalls)
- Saved $s6 ... $s7
  - Must be saved by the caller, remain unmodified
- Arguments $a0 ... $a3 passed to subroutine remain live
- Returned $v0 ... $v1
Syntax of MIPS
Classes of MIPS Instructions
ALU: ADD (1.3)
ALU: ADD (2.3)
ALU: ADD (3.3)
Pseudo-Ops: ADDI (1.2)
Pseudo-Ops: ADDI (2.2)
Branch: BLT (1.2)
Branch: BLT (2.2)
ALU: SUB (1.2)
ALU: SUB (2.2)
ALU: AND
ALU: OR
ALU: XOR
ALU: NOR
ALU: MULT (1.2)
ALU: MULT (2.2)
ALU: DIV
ALU: Shifts (1.3)
ALU: Shifts (2.3)
ALU: Shifts (3.3)
Memory Access (1.3)
Memory Access (2.3)
Memory Access (3.3)
MIPS OpCodes and Usage

Gabriel Hugh Elkaim
Syntax of MIPS

- One instruction per line
- Comments are anything after 
- Comments may \textbf{NOT} span lines
- Some systems will support \texttt{C++} style comments /* */
Classes of MIPS Instructions

ALU - perform math on registers
  - ADD - op code

MEMORY MANIPULATION - copy data to/from registers to memory
  - LB

Flow control - control flow/delay, flushing, branch
  - BFEQ
ALU: ADD (1.3)
ALU: ADD (2.3)

\[ add \; R_d, \; R_s, \; R_t \rightarrow R_d = R_s + R_t \]

destination \hspace{2cm} things to be added together

* if carry out bit \( b_{31} \) of answer \rightarrow OVERFLOW
EXCEPTION

addu \hspace{2cm} "add unsigned"
ALU: ADD (3.3)

addi — add immediate

\[ R_d = R_s + \text{VALUE} \]

addiu — add immediate unsigned

\[ \text{addi} \quad \text{at}, \quad \$t1, \quad \text{OxDEADBEEF} \]

\[ \text{at} \rightarrow \text{DFXDBEEF} \]

\[ \text{lui} \quad \text{at}, \quad \text{OxDEADBEEF} \]

\[ \text{ori} \quad \text{at}, \quad \text{OxBEEF} \]
Pseudo-Ops: ADDI (1.2)

ADDI $t1, $t1, 0x12345678

```asm

; 0x12340000

```

ADDI $t1, $t1, 0x00005678

```asm

```

ADDI $t1, $t1, 0x0000

```asm

```

ADDI $t1, $t1, $t0

```asm

```

ADDI $t1, $t1, $t0

```asm

```

LDI $t2, 0x1234

```

ORI $t3, $t3, 0x8888

```asm

```

AND $t1, $t1, $t2

```asm

```asm

```
Pseudo-Ops: ADDI (2.2)

ADD
ADDI

\* ADDI

THROWN EXCEPTION ON OVERFLOW

\* ADDI

ADDU
ADDIU

\* ADDU

WILL NOT THROW EXCEPTION
ALU: SUB (1.2)

$\text{SUB} \rightarrow \text{Subtraction} \quad R_d = R_s - R_t$

$\text{SUB} \quad d \rightarrow t, s \rightarrow t$

$\text{SUBU} \rightarrow \text{unsigned}

\text{will not throw exception}$
ALU: SUB (2.2)

\[
\text{SUB } \quad \begin{cases} 
\text{win on } 00b & \text{exception on } 11b \\
\text{SUB } \quad \begin{cases} 
\text{win on } 0b & \text{no } 01b \\
\text{SUB } \quad \begin{cases} 
\end{cases}
\end{cases}
\end{cases}
\]
ALU: AND

\[ \text{AND } R_d, R_s, R_6 \rightarrow R_d = R_s \text{ AND } R_6 \ (\text{emitted}) \]

\[ \text{ANDI } - \text{AND immediate} \]
\[ \text{andi } R_d, R_s, \text{ immediate} \]
\[ R_d = R_s \text{ AND } [\text{0000 immediate}] \]
ALU: OR

or — branch condition or
or $d_{to}, d_{to}, d_{to}$

or $R_d, R_s, R_r$

or $R_d, R_s, R_r$

or $R_d, R_s, R_r$

$R_d = R_s$ or $R_r$

or $R_d, R_s, R_r$

or immediate

or immediate

$R_d = R_s$ or [10000 mm]
ALU: XOR

xor - exclusive or

\[ R_d = R_s \oplus R_4 \]

xori - exclusive or with immediate

\[ R_d = R_s \oplus [\text{imm}] \]

\[ R_d = R_s \oplus [000000 \text{ imm}] \]
ALU: NOR

\[ \text{nor } R_d, R_s, R_t \]

\[ R_o = \overline{(R_s + R_t)} \]
ALU: MULT (1.2)

**MULT** - **MULTIPLY**

```
32 in
32 in  \times \rightarrow 64 bit out
```

```
mult Rj, Rk
muli Rj
mflo Rk
```

**MULT - UNSIGNED MULTIPLICATION**
ALU: MUL (2.2)

mul - multiply within 32 bits

mul $R_d, R_s, R_t$

$R_d = R_s \times R_t$ (keep bottom 32 bits)

mul
ALU: DIV

DIV - division

div R₂, R₁

lo = R₂ / R₁ (integer)

hi = R₂ % R₁ (remainder)

DIVU - unaligned division
ALU: Shifts (1.3)

SLL - Shift Left Logical
SLL Rd, Rs, S

SLV - Shift Left Variable
SLV Rd, Rs, Rj [4:0]
ALU: Shifts (2.3)

$sr1$ - shift right logical

$sr1 \rightarrow Rd, R3, sa$

$Ra \leftarrow Rj >> Rk (4:0)$
ALU: Shifts (3.3)

`sra - shift right arithmetic`

`sra Rd, Rs, Sa` (adds sign extension)

`srav - shift right arithmetic variable`
Memory Access (1.3)
Memory Access (2.3)
Memory Access (3.3)
Memory: LB (1.2)
Memory: LB (2.2)
Memory: LH
Memory: LW
Memory: SB (1.2)
Memory: SB (2.2)
Memory: SH
Memory: SW
Labels (1.2)
Labels (2.2)
Memory: LA (1.2)
## Memory Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100000</td>
<td></td>
</tr>
<tr>
<td>0x100002</td>
<td></td>
</tr>
<tr>
<td>0x100003</td>
<td></td>
</tr>
<tr>
<td>0x100004</td>
<td></td>
</tr>
<tr>
<td>0x100005</td>
<td></td>
</tr>
<tr>
<td>0x100006</td>
<td></td>
</tr>
<tr>
<td>0x100007</td>
<td></td>
</tr>
<tr>
<td>0x100008</td>
<td></td>
</tr>
<tr>
<td>0x100009</td>
<td></td>
</tr>
<tr>
<td>0x10000A</td>
<td></td>
</tr>
<tr>
<td>0x10000B</td>
<td></td>
</tr>
<tr>
<td>0x10000C</td>
<td></td>
</tr>
<tr>
<td>0x10000D</td>
<td></td>
</tr>
<tr>
<td>0x10000E</td>
<td></td>
</tr>
<tr>
<td>0x10000F</td>
<td></td>
</tr>
<tr>
<td>0x100010</td>
<td></td>
</tr>
<tr>
<td>0x100011</td>
<td></td>
</tr>
<tr>
<td>0x100012</td>
<td></td>
</tr>
</tbody>
</table>
Conditional Operators
Branches: B
Branches: BEQ
Branches: BNE
Branches: BGEZ/BGTZ/BLTZ
Branch: SLT
Branch: SLTU/SLTI/SLTIU
Conditional Structures: IF (1.4)
Conditional Structures: IF (2.4)
Conditional Structures: IF (3.4)
Conditional Structures: IF (4.4)
Conditional Structures: IF-ELSE (1.2)
Conditional Structures: IF-ELSE (2.2)
Conditional Structures: WHILE (1.2)
Conditional Structures: WHILE (2.2)
Conditionals: BREAK, CONTINUE
Conditional Structures: FOR (1.2)
Conditional Structures: FOR (2.2)
MIPS: Syscall (1.3)
MIPS: Syscall (2.3)
MIPS: Syscall (3.3)