3. Instruction Set Architecture –
The MIPS architecture


Prof. Valeria Bertacco & Prof. Scott Mahlke
EECS Department
University of Michigan in Ann Arbor, USA

© V. Bertacco & S. Mahlke, 2007
The material in this presentation cannot be copied in any form without our written permission

Instruction Set Architecture (ISA) Design Lectures

- Lecture 2: Storage types and addressing modes
- Lecture 3: MIPS architecture
- Lecture 4: Calling functions / passing arguments
- Lecture 5: Translation software; libraries, VMs
Recap (Storage)

- Registers
  - Small array of storage locations in processor
  - Fast access
  - Direct addressing only
  - Digression: the PC register

- Memory
  - Large array of storage locations
  - Slow access
  - Many addressing modes (direct, indirect, double indirect, reg indirect, base+displmt)

Recap (Assembly/Machine Code)

- Computers store program instructions the same way they store data.
- Each instruction is encoded as a number
  - Opcode field: what instruction to perform.
  - Operand fields: what data to perform it on.

```
add     R2     3     R1
011011   010    011    001
```
LC-2Kx Processor

- 32-bit processor
  - Instructions are 32 bits
  - Integer registers are 32 bits
- 8 registers
- supports 65536 words of memory (addressable space)
- 8 instructions
  - add, nand, lw, sw, beq, jalr, halt, noop

Assembly Format

- Format:
  - Label <white> instr <white> field0 <white> field1 <white> field2 <white> comments
- Labels: Max of 6 characters, starts with a letter {A..Z,a..z} followed by letters or numbers.
- Instruction: add, nand, lw, sw, beq, jalr, noop, halt and "fill"

```assembly
lw 0    1    five
load reg1 with 5 (uses symbolic address)

lw 1    2    3
load reg2 with -1 (uses numeric address)

start add 1    2    1
decrement reg1

beq 0    1    2
goto end of program when reg1 equals 0

beq 0    0    start
go back to the beginning of the loop

noop

done halt
end of program

five .fill 5

neg1 .fill -1

stAddr .fill start
will contain the address of start (2)
```
**Instruction Encoding**

- Instruction set architecture defines the mapping of assembly instructions to machine code

```
add 1 2 3 // r3 = r1 + r2
```

---

**Instruction Formats**

- Positional organization of bits (Implies nothing about bit values!!)

- **R type instructions** (add, nand)

```
31-25  24-22  21-19  18-16  15-3  2-0
unused  opcode  regA  regB  unused  destR
```

- **I type instructions** (lw, sw, beq)

```
31-25  24-22  21-19  18-16  15-0
unused  opcode  regA  regB  offset
```
Bit Encodings

- Opcode encodings
  - add (000), nand (001), lw (010), sw (011), beq (100), jalr (101), halt (110), noop (111)

- Register values
  - Just encode the register number (r2 = 010)

- Immediate values
  - Just encode the values (Remember to give all the available bits a value!!)

Example Encoding - nand

nand 3 4 7 (r7 = r3 nand r4)

<table>
<thead>
<tr>
<th>31-25</th>
<th>24-22</th>
<th>21-19</th>
<th>18-16</th>
<th>15-3</th>
<th>2-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
<td>opcode</td>
<td>regA</td>
<td>regB</td>
<td>unused</td>
<td>destR</td>
</tr>
<tr>
<td>0000000</td>
<td>001</td>
<td>011</td>
<td>100</td>
<td>0000000000000</td>
<td>111</td>
</tr>
</tbody>
</table>

Convert to Hex → 0x005C0007
Convert to Dec → 6029319
Example Encoding - lw

- lw 5 2 -8 (r2 = M[r5 + -8])

<table>
<thead>
<tr>
<th>unused</th>
<th>opcode</th>
<th>regA</th>
<th>regB</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>010</td>
<td>101</td>
<td>010</td>
<td>1111111111111000</td>
</tr>
</tbody>
</table>

Convert to Hex → 0x00AAFFF8
Convert to Dec → 11206648

Mini-review: Representing Negative Numbers

- 2's complement representation

<table>
<thead>
<tr>
<th>Positive numbers:</th>
<th>Negative numbers:</th>
<th>Neg r5, #4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0000 0001</td>
<td>-1 1111 1111</td>
<td></td>
</tr>
<tr>
<td>2 0000 0010</td>
<td>-2 1111 1110</td>
<td></td>
</tr>
<tr>
<td>3 0000 0011</td>
<td>-3 1111 1101</td>
<td></td>
</tr>
<tr>
<td>4 0000 0100</td>
<td>-4 1111 1100</td>
<td></td>
</tr>
<tr>
<td>5 0000 0101</td>
<td>-5 1111 1011</td>
<td></td>
</tr>
<tr>
<td>6 0000 0110</td>
<td>-6 1111 1010</td>
<td></td>
</tr>
<tr>
<td>7 0000 0111</td>
<td>-7 1111 1001</td>
<td></td>
</tr>
<tr>
<td>8 0000 1000</td>
<td>-8 1111 1000</td>
<td></td>
</tr>
<tr>
<td>9 0000 1001</td>
<td>-9 1111 0111</td>
<td></td>
</tr>
</tbody>
</table>

Neg r5, #4

4 = 0000 0100
 complement 1111 1011
+ 1 1111 1100 = -4
Class Problem 1

- Compute the encoding in Hex for:
  - add 3 7 3 \( (r3 = r3 + r7) \) (add = 000)
  - sw 1 5 67 \( (M[r1+67] = r5) \) (sw = 011)

<table>
<thead>
<tr>
<th>31-25</th>
<th>24-22</th>
<th>21-19</th>
<th>18-16</th>
<th>15-3</th>
<th>2-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
<td>opcode</td>
<td>regA</td>
<td>regB</td>
<td>unused</td>
<td>destR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31-25</th>
<th>24-22</th>
<th>21-19</th>
<th>18-16</th>
<th>15-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
<td>opcode</td>
<td>regA</td>
<td>regB</td>
<td>offset</td>
</tr>
</tbody>
</table>

Mini-review: What is a shift?

- C/C++
  - \( a = b >> 2; \) \( b = 01101110 \)
  - \( c = d << 4; \) \( b = 00011011 \)

- MIPS
  - slr $3, $2, 2
  - sll $5, $4, 4

- LC2K
  - ...

The University of Michigan
© V. Bertacco & S. Mahlke – 2007
MIPS Instruction Set

- Three main types of instructions:
  - **Arithmetic**
    - Add, subtract, multiply, divide
    - Logical: and, or, shift, rotate, etc.
    - Compare: equal, lt, le, ne, etc.
  - **Memory access**
    - Load, store
  - **Sequencing / control flow**
    - Jump, branch, function call, return

MIPS Arithmetic Instructions – type 1

- Format: three register operand fields
  - e.g., add $3, $4, $7

- C-code example: \( f = (g + h) - (i + j) \)

\[
\begin{align*}
\text{add} & \quad t0, g, h \\
\text{add} & \quad t1, i, j \\
\text{sub} & \quad f, t0, t1
\end{align*}
\]

\[
\begin{align*}
\text{add} & \quad $1, $3, $4 \\
\text{add} & \quad $2, $5, $6 \\
\text{sub} & \quad $7, $1, $2
\end{align*}
\]
MIPS Arithmetic Instructions – type 2

- Format: two register operand fields and an immediate constant (16 bit)
  - e.g., addi $3, $4, 10
- C-code example: \( f = g \times 10 \)

\[
\text{multi } \$7, \$5, 10
\]

What About Immediates Larger Than 16 Bits??

- Use 2 instructions
  - Load upper bits (load upper immediate, lui)
  - Add in lower bits (add immediate, addi)
- C-code example: \( f = 0x10002 \)

\[
\begin{align*}
\text{lui } \$7, 1 & \quad 0x10000 \\
\text{addi } \$7, \$7, 2 & \quad 0x10002
\end{align*}
\]
MIPS Arithmetic Instructions - recap

- abs rd, rs // absolute value
- add rd, rs, rt // add
- addi rd, rs, imm // add immediate
- and rd, rs, rt // logical AND of bits
- div rd, rs, rt // divide
- mult rd, rs, rt // multiply
- neg rd, rs // negate (pseudo-instruction)
- nor rd, rs, rt // logical NOR of bits
- sll rd, rs, imm // shift left logical
- sub rd, rs, rt // sub rt from rs
- subi rd, rs, imm // sub immediate from rs
- srl rd, rs, imm // shift right logical
- li rd, imm // load immediate (pseudo-instruction)

Class Problem 2

- Show the C and MIPS assembly for extracting the value in bits 15:10 from a 32-bit integer variable

Remember each hex digit is 4 bits
MIPS Memory Instructions

- Supports base + displacement mode only
  - Base is a register
  - Offset is a 16-bit immediate

- Format: 2 registers (dest, base) and 16-bit immediate (offset)
  - Example:
    \[ \text{lw} \; \$3, \; 1000(\$4) \] // load word
    Retrieves 32-bit value from memory location ($4+1000) and puts the result into $3

Load Instruction Sizes

How much data is retrieved from memory at the given address?

- \[ \text{lw} \; \$3, \; 1000(\$4) \]
  - retrieve a word (32 bits) from address ($4+1000)

- \[ \text{lh} \; \$3, \; 1000(\$4) \]
  - retrieve a halfword (16 bits) from address ($4+1000)

- \[ \text{lb} \; \$3, \; 1000(\$4) \]
  - retrieve a byte (8 bits) from address ($4+1000)
Sign/Zero Extension

- Registers in MIPS are 32 bits!
- So what happens when you load 8 or 16 bits?
  - Sign extend if the load is signed
    
    $0xFE \xrightarrow{} 0x0000001F \xrightarrow{} 0xFFFFFFFFE$
  
  - Zero extend if the load is unsigned
    
    $0x1F \xrightarrow{} 0x0000001F \xrightarrow{} 0x000000FE$

MIPS Memory Instructions - recap

- Load instructions
  - `lb` \ load byte signed (load 8 bits, sign extend)
  - `lbu` \ load byte unsigned (load 8 bits, zero extend)
  - `lh` \ load halfword (load 16 bits, sign extend)
  - `lhu` \ load halfword unsigned (load 16 bits, zero extend)
  - `lw` \ load word (load 32 bits, no extension)

- Store instructions (No sign/zero extension for stores)
  - `sb` $3$, 1000($4) \ store 8 LSBs of $3$ to M[$4+1000]$
  - `sh` $3$, 1000($4) \ store 16 LSBs of $3$ to M[$4+1000]$
  - `sw` $3$, 1000($4) \ store all 32 bits of $3$ to M[$4+1000]$
Load Instruction in Action

- **lb $3, 1000($4) // load byte**
  
  Retrieves 8-bit value from memory location ($4+1000) and puts the result into $3 (sign extended)

  
  Calculate address:
  
  $2500 + 1000 = 3500$

  
  Sign extend 254 (0xFE) to 32 bit → 0xFFF FFFE

  
  Calculate address:
  
  $2504 + 1000 = 3504$
Example Code Sequence

What is the final state of memory once you execute the following instruction sequence?

lw $4, 100($0)
lb $3, 102($0)
sw $3, 100($0)
sb $4, 102($0)

Example Code Sequence – insn 1

What is the final state of memory once you execute the following instruction sequence?

lw $4, 100($0)
lb $3, 102($0)
sw $3, 100($0)
sb $4, 102($0)
Example Code Sequence – insn 2

What is the final state of memory once you execute the following instruction sequence?

```
lw $4, 100($0)
lb $3, 102($0)
sw $3, 100($0)
sb $4, 102($0)
```

```
<table>
<thead>
<tr>
<th>register file</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3 0xFFFF FFFF</td>
<td>0x02</td>
</tr>
<tr>
<td>$4 0x0203 FF05</td>
<td>0x03</td>
</tr>
</tbody>
</table>
```

Example Code Sequence – insn 3

What is the final state of memory once you execute the following instruction sequence?

```
lw $4, 100($0)
lb $3, 102($0)
sw $3, 100($0)
sb $4, 102($0)
```

```
<table>
<thead>
<tr>
<th>register file</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3 0xFFFF FFFF</td>
<td>0x02</td>
</tr>
<tr>
<td>$4 0x0203 FF05</td>
<td>0x03</td>
</tr>
</tbody>
</table>
```

The University of Michigan
© V. Bertacco & S. Mahlke – 2007
**Example Code Sequence – insn 4**

What is the final state of memory once you execute the following instruction sequence?

```
lw $4, 100($0)
lb $3, 102($0)
sw $3, 100($0)
sb $4, 102($0)
```

```
0xFF
0xFF
0x05
0xFF
```

**Class Problem 3**

What is the final state of memory once you execute the following instruction sequence?

```
lhu $3, 100($0)
lb $4, 102($0)
sw $3, 100($0)
sh $4, 102($0)
```

```
0xFF
0xFF
0x77
0xFF
```

![Memory Diagram](image)
Converting C to Assembly

- Memory layout \(\rightarrow\) memory addresses
- Branches
- Procedure calls
- Expression trees
- Register allocation
- This topic (along with procedure calls) is the focus of the next lecture, but let's do a brief warmup

Converting C to assembly – example 1

Write MIPS Assembly Code for the following C Expression:

\[ C: \quad a = b + names[i] \]

Assume that \(a\) is in $1$, \(b\) is in $2$, \(i\) is in $3$, and the array \(names\) starts at address 1000 and holds 32-bit integers.

\[
\begin{align*}
\text{mult} & \quad 5, 3, 4 & \quad \text{// calculate array offset} \\
\text{lw} & \quad 4, 1000 ($5) & \quad \text{// load names[i]} \\
\text{add} & \quad 1, 2, 4 & \quad \text{// calculate b + names[i]}
\end{align*}
\]
Converting C to assembly – example 2

Write MIPS Assembly Code for the following C Expression:

class { int a; char b, c; } y;
y.a = y.b + y.c;

Assume that a pointer to y is in $1.

lb $2, 4 ($1)   // load y.b
lb $3, 5 ($1)   // load y.c
add $4, $2, $3  // calculate y.b+y.c
sw $4, 0 ($1)   // store y.a