iPhone games! (and general SDK) ⇒ Apple is (finally) releasing an iPhone Software Developer Kit on March 6th (?) That means iPhone games that use both touch and accelerometer input!

Outline
- Disassembly
- Pseudoinstructions
- “True” Assembly Language (TAL) vs. “MIPS” Assembly Language (MAL)

Decoding Machine Language
- How do we convert 1s and 0s to assembly language and to C code?
  Machine language ⇒ assembly ⇒ C?
- For each 32 bits:
  1. Look at opcode to distinguish between R-Format, J-Format, and I-Format.
  2. Use instruction format to determine which fields exist.
  3. Write out MIPS assembly code, converting each field to name, register number/name, or decimal/hex number.
  4. Logically convert this MIPS code into valid C code. Always possible? Unique?

Decoding Example (1/7)
- Here are six machine language instructions in hexadecimal:

  00001025
  0005402A
  11000003
  00441020
  20A5FFFF
  08100001

- Let the first instruction be at address 4,194,304,000 (0x00400000).
- Next step: convert hex to binary

Decoding Example (2/7)
- The six machine language instructions in binary:

- Next step: identify opcode and format
### Decoding Example (3/7)
- Select the opcode (first 6 bits) to determine the format:

<table>
<thead>
<tr>
<th>R</th>
<th>R</th>
<th>I</th>
<th>R</th>
<th>J</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>0000000000000000000100000010010</td>
<td>0000000000000101010000000010101</td>
<td>000100000000000000000000000011</td>
<td>00000000000000000000000000000011</td>
<td>00001000000000000000000000000001</td>
</tr>
</tbody>
</table>

- Look at opcode:
  - 0 means R-Format, 2 or 3 mean J-Format, otherwise I-Format.

- Next step: separation of fields

### Decoding Example (4/7)
- Fields separated based on format(opcode):

<table>
<thead>
<tr>
<th>R</th>
<th>R</th>
<th>I</th>
<th>R</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>5</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Next step: translate (“disassemble”) to MIPS assembly instructions

### Decoding Example (5/7)
- MIPS Assembly (Part 1):

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly instructions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400000</td>
<td>or $2,$0,$0</td>
</tr>
<tr>
<td>0x00400004</td>
<td>slt $8,$0,$5</td>
</tr>
<tr>
<td>0x00400008</td>
<td>beq $8,$0,$3</td>
</tr>
<tr>
<td>0x004000c0</td>
<td>add $2,$2,$4</td>
</tr>
<tr>
<td>0x0040010</td>
<td>addi $5,$5,-1</td>
</tr>
<tr>
<td>0x0040014</td>
<td>j x100001</td>
</tr>
</tbody>
</table>

- Better solution: translate to more meaningful MIPS instructions (fix the branch/jump and add labels, registers)

### Decoding Example (6/7)
- MIPS Assembly (Part 2):

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly instructions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400000</td>
<td>or $v0,$0,$0</td>
</tr>
<tr>
<td>0x00400004</td>
<td>slt $t0,$0,$a1</td>
</tr>
<tr>
<td>0x00400008</td>
<td>beq $t0,$0,Exit</td>
</tr>
<tr>
<td>0x004000c0</td>
<td>add $v0,$v0,$a0</td>
</tr>
<tr>
<td>0x0040010</td>
<td>addi $a1,$a1,-1</td>
</tr>
<tr>
<td>0x0040014</td>
<td>j Loop</td>
</tr>
<tr>
<td>0x0040018</td>
<td>Exit:</td>
</tr>
</tbody>
</table>

- Next step: translate to C code (must be creative!)

### Decoding Example (7/7)
- After C code (Mapping below)

<table>
<thead>
<tr>
<th>Before Hex:</th>
<th>After C code (Mapping below)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001025</td>
<td>$v0: product</td>
</tr>
<tr>
<td>0005402A</td>
<td>$a0: multiplicand</td>
</tr>
<tr>
<td>11000002</td>
<td>$a1: multiplier</td>
</tr>
<tr>
<td>00441020</td>
<td>product = 0;</td>
</tr>
</tbody>
</table>
| 20A5FF | while (multiplier > 0) {
| 08100001 | product += multiplicand;
| 0 } multiplier -= 1; |

| or $v0,$0,$0 |
| Loop: slt $t0,$0,$a1 |
| beq $t0,$0,Exit |
| add $v0,$v0,$a0 |
| addi $a1,$a1,-1 |
| j Loop |

- Demostrated Big 61C
  - Idea: Instructions are just numbers, code is treated like data

### Administrivia
- Midterm is next week! Day and location are still TBA
  - Old midterms online (link at top of page)
  - Lectures and reading materials fair game
  - Fix green sheet errors (if old book)
- Review session also TBA
  - Project 2 is due March 5 at 11:59PM
  - That’s Wednesday!
  - There was a file update. See spec page.
Review from before: \texttt{lui}

- So how does \texttt{lui} help us?
  - Example:
    
    \begin{verbatim}
    add $t0,$t0, 0xABABCDCD
    \end{verbatim}

    becomes:
    
    \begin{verbatim}
    lui $at, 0xABAB
    ori $at, $at, 0xCDCD
    add $t0,$t0,$at
    \end{verbatim}

  - Now each I-format instruction has only a 16-bit immediate.

  \textbf{Wouldn't it be nice if the assembler would do this for us automatically?}
  - If number too big, then just automatically replace \texttt{addi} with \texttt{lui}, \texttt{ori}, \texttt{add}.

\textbf{True Assembly Language (1/3)}

- \textbf{Pseudoinstruction:} A MIPS instruction that doesn’t turn directly into a machine language instruction, but into other MIPS instructions.

- What happens with pseudo-instructions?
  - They're broken up by the assembler into several “real” MIPS instructions.

- Some examples follow

\textbf{Example Pseudoinstructions}

- Register Move
  
  \begin{verbatim}
  move reg2,reg1
  \end{verbatim}

  Expands to:
  
  \begin{verbatim}
  add reg2,$zero,reg1
  \end{verbatim}

- Load Immediate
  
  \begin{verbatim}
  li reg,value
  \end{verbatim}

  If value fits in 16 bits:
  
  \begin{verbatim}
  addi reg,$zero,value
  \end{verbatim}

  else:
  
  \begin{verbatim}
  lui reg,upper 16 bits of value
  ori reg,$zero,lower 16 bits
  \end{verbatim}

\textbf{Example Pseudoinstructions}

- Load Address: How do we get the address of an instruction or global variable into a register?
  
  \begin{verbatim}
  la reg,label
  \end{verbatim}

  Again if value fits in 16 bits:
  
  \begin{verbatim}
  addi reg,$zero,label_value
  \end{verbatim}

  else:
  
  \begin{verbatim}
  lui reg,upper 16 bits of value
  ori reg,$zero,lower 16 bits
  \end{verbatim}

\textbf{Example Pseudoinstructions}

- \textbf{Rotate Right Instruction}
  
  \begin{verbatim}
  ror reg, value
  \end{verbatim}

  Expands to:
  
  \begin{verbatim}
  srl $at, reg, value
  sll reg, reg, 32-value
  or reg, reg, $at
  \end{verbatim}

- \textbf{“No OPeration” instruction}
  
  \begin{verbatim}
  nop
  \end{verbatim}

  Expands to instruction = 0_{\text{imm}}:
  
  \begin{verbatim}
  sll $0, $0, 0
  \end{verbatim}
Example Pseudoinstructions

- Wrong operation for operand
  
  \[
  \text{addu reg, reg, value} \# \text{should be addiu}
  \]

  If value fits in 16 bits, addu is changed to:
  \[
  \text{addiu reg, reg, value}
  \]

  else:
  \[
  \text{lui $at, upper 16 bits of value}
  \]
  \[
  \text{ori $at, $at, lower 16 bits}
  \]
  \[
  \text{addu reg, reg, $at}
  \]

- \text{How do we avoid confusion about whether we are talking about MIPS assembler with or without pseudoinstructions?}

Questions on Pseudoinstructions

- \text{Question:}
  - How does MIPS assembler / SPIM recognize pseudo-instructions?

- \text{Answer:}
  - It looks for officially defined pseudo-instructions, such as \text{ror} and \text{move}
  - It looks for special cases where the operand is incorrect for the operation and tries to handle it gracefully

Rewrite TAL as MAL

- \text{TAL:}
  
  \[
  \text{or} \quad \text{$v0, 0, 0$}
  \]
  
  \[
  \text{Loop:}
  \begin{align*}
  &\text{slt}\quad \text{$t0, 0, $a1} \\
  &\text{beq}\quad \text{$t0, 0, Exit} \\
  &\text{add}\quad \text{$v0, $v0, $a0} \\
  &\text{addi}\quad \text{$a1, $a1, -1} \\
  &\text{j}\quad \text{Loop}
  \end{align*}
  \]

  \[
  \text{Exit:}
  \]

- \text{MAL:}
  
  \[
  \text{li} \quad \text{$v0, 0$}
  \]
  
  \[
  \text{Loop:}
  \begin{align*}
  &\text{ble}\quad \text{$a1, zero, Exit} \\
  &\text{add}\quad \text{$v0, $v0, $a0} \\
  &\text{sub}\quad \text{$a1, $a1, 1} \\
  &\text{j}\quad \text{Loop}
  \end{align*}
  \]

  \[
  \text{Exit:}
  \]

Peer Instruction

- Which of the instructions below are \text{MAL} and which are \text{TAL}?

  i. \text{addi $t0$, $t1$, 40000}

  ii. \text{beq $s0$, 10, Exit}

  iii. \text{sub $s0$, $s1$, 1}
Peer Instruction Answer

• Which of the instructions below are MAL and which are TAL?

  i.  `addi $t0, $t1, 40000`

  40,000 > +32,767 => lui, ori

  ii. `beq $s0, 10, Exit`

    beq: both must be registers

    Exit: if > 2^9, then MAL

  iii. `sub $t0, $t1, 1`  

    sub: both must be registers;

    even if it was subi, there is no subi in TAL;

    generates addi $t0,$t1,-1

In Conclusion

• Disassembly is simple and starts by decoding opcode field.
  
  • Be creative, efficient when authoring C

• Assembler expands real instruction set (TAL) with pseudoinstructions (MAL)
  
  • Only TAL can be converted to raw binary
  
  • Assembler’s job to do conversion
  
  • Assembler uses reserved register $at
  
  • MAL makes it much easier to write MIPS