Review of Last Lecture

- **Simplifying MIPS**: Define instructions to be the same size as data word (one word) so that they can use the same memory
  - Computer actually stores programs as a series of these 32-bit numbers
- **MIPS Machine Language Instruction**: 32 bits representing a single instruction

<table>
<thead>
<tr>
<th>R:</th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I:</td>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Agenda

- **I-Format**
  - Branching and PC-Relative Addressing
- **Administrivia**
- **J-Format**
- **Pseudo-instructions**
- **Bonus**: Assembly Practice
- **Bonus**: Disassembly Practice

I-Format Immediates

- **Immediate (16)**: Two’s complement number
  - All computations done in words, so 16-bit immediate must be extended to 32 bits
  - Green Sheet specifies ZeroExtImm or SignExtImm based on instruction
- Can represent $2^{16}$ different immediates
  - This is large enough to handle the offset in a typical `lw/sw`, plus the vast majority of values for `slti`

Dealing With Large Immediates

- How do we deal with 32-bit immediates?
  - Sometimes want to use immediates $\pm 2^{15}$ with `addi, lw, sw and slti`
  - Bitwise logic operations with 32-bit immediates
- **Solution**: Don’t mess with instruction formats, just add a new instruction
  - **Load Upper Immediate** (`lui`)
    - `lui reg, imm`
    - Moves 16-bit `imm` into upper half (bits 16-31) of `reg` and zeros the lower half (bits 0-15)
lui Example

- Want:  addiu $t0,$t0,0xABABCDCD
  - This is a pseudo-instruction!
- Translates into:
  
  \[
  \begin{align*}
  \text{lui} & \; \text{sat}, 0xABAB & \# \text{upper 16} \\
  \text{ori} & \; \text{sat}, \text{sat}, 0xCDCD & \# \text{lower 16} \\
  \text{addu} & \; \text{t0}, \text{t0}, \text{sat} & \# \text{move}
  \end{align*}
  \]

- Now we can handle everything with a 16-bit immediate!

Branching Instructions

- beq and bne
  - Need to specify an address to go to
  - Also take two registers to compare
- Use I-Format:
  
  \[
  \begin{array}{ccc}
  \text{opcode} & \text{rs} & \text{rt} & \text{immediate} \\
  \hline
  31 & 0 & 0 & 0 \\
  \end{array}
  \]
  - opcode specifies beq (4) vs. bne (5)
  - rs and rt specify registers
  - How to best use immediate to specify addresses?

Branching Instruction Usage

- Branches typically used for loops (if-else, while, for)
  - Loops are generally small (< 50 instructions)
  - Function calls and unconditional jumps handled with jump instructions (J-Format)
- Recall: Instructions stored in a localized area of memory (Code/Text)
  - Largest branch distance limited by size of code
  - Address of current instruction stored in the program counter (PC)

PC-Relative Addressing

- PC-Relative Addressing: Use the immediate field as a two’s complement offset to PC
  - Branches generally change the PC by a small amount
  - Can specify ±2^{15} addresses from the PC

- So just how much of memory can we reach?

Branching Reach

- Recall: MIPS uses 32-bit addresses
  - Memory is byte-addressed
- Instructions are word-aligned
  - Address is always multiple of 4 (in bytes), meaning it ends with 0b00 in binary
  - Number of bytes to add to the PC will always be a multiple of 4
- Immediate specifies words instead of bytes
  - Can now branch ±2^{15} words
  - We can reach 2^{16} instructions = 2^{18} bytes around PC

Branch Calculation

- If we don’t take the branch:
  - \( PC = PC + 4 = \text{next instruction} \)
- If we do take the branch:
  - \( PC = (PC+4) + (\text{immediate} \times 4) \)

- Observations:
  - immediate is number of instructions to jump (remember, specifies words) either forward (+) or backwards (−)
  - Branch from PC+4 for hardware reasons; will be clear why later in the course
Branch Example (1/2)
- MIPS Code:
  ```
  Loop:  
  beq $9,$0, End
  addu $8,$8,$10
  addiu $9,$9,-1  
  j Loop
  
  End:
  ```

- I-Format fields:
  - opcode = 4 (look up on Green Sheet)
  - rs = 9 (first operand)
  - rt = 0 (second operand)
  - immediate = 3

Questions on PC-addressing
- Does the value in branch immediate field change if we move the code?
  - If moving individual lines of code, then yes
  - If moving all of code, then no
- What do we do if destination is > 2^{15} instructions away from branch?
  - Other instructions save us
  ```
  beq $s0,$0,far   
  bne $s0,$0,next  
  # next instr --> j far
  ```
  ```
  next:    # next instr
  ```

Administrivia
- Project 1 Part 1 due Sunday 2/17
  - No homework! (still have discussion and lab)
  - Solo project
- Midterm in 3 weeks (3/4)
  - Never too early to start looking at past exams
  - Midterm Review: Fri 3/1, 6-9pm, 2050 VLSB (this room!)
J-Format Instructions (1/4)

- For branches, we assumed that we won’t want to branch too far, so we can specify a \textit{change} in the PC.
- For general jumps (\texttt{j} and \texttt{jal}), we may jump to \textit{anywhere} in memory.
  - Ideally, we would specify a 32-bit memory address to jump to.
  - Unfortunately, we can’t fit both a 6-bit \texttt{opcode} and a 32-bit address into a single 32-bit word.

J-Format Instructions (2/4)

- Define two “fields” of these bit widths:
  \begin{tabular}{c|c}
    31 & 6  \\ \hline
    26 & 0  \\ \hline
  \end{tabular}

- As usual, each field has a name:
  \begin{tabular}{c|c|c}
    31 & \texttt{opcode} & \texttt{target. address}  \\ \hline
    0 &  \\ \hline
  \end{tabular}

- Key Concepts:
  - Keep \texttt{opcode} field identical to R-Format and I-Format for consistency.
  - Collapse all other fields to make room for large target address.

J-Format Instructions (3/4)

- We can specify $2^6$ addresses:
  - Still going to word-aligned instructions, so add \texttt{0b00} as last two bits (multiply by 4).
  - This brings us to 28 bits of a 32-bit address.
- Take the 4 highest order bits from the PC:
  - Cannot reach everywhere, but adequate almost all of the time, since programs aren’t that long.
  - Only problematic if code straddles a 256MB boundary.
- If necessary, use 2 jumps or \texttt{jr} (R-Format) instead.

J-Format Instructions (4/4)

- Jump instruction:
  - New PC = \{ (PC+4)[31..28], target address, 00 \}
- Notes:
  - \{, , \} means concatenation
  - \{ 4 bits, 26 bits, 2 bits \} = 32 bit address
    - Book uses || instead
  - Array indexing: [31..28] means highest 4 bits
  - For hardware reasons, use PC+4 instead of PC.

Question: When combining two C files into one executable, we can compile them independently and then merge them together.

When merging two or more binaries:

1) \textbf{Jump} instructions don’t require any changes
2) \textbf{Branch} instructions don’t require any changes

\begin{tabular}{c|c}
  1 & 2  \\ \hline
  a) & F \checkmark  \\ \hline
  b) & F \checkmark  \\ \hline
  c) & T \checkmark  \\ \hline
  d) & T \checkmark  \\ \hline
\end{tabular}

Agenda

- I-Format
  - Branching and PC-Relative Addressing
- Administrivia
- J-Format
- \textbf{Pseudo-instructions}
  - Bonus: Assembly Practice
  - Bonus: Disassembly Practice
Assembler Pseudo-Instructions

- Certain C statements are implemented unintuitively in MIPS
  - e.g. assignment \((a=b)\) via addition with 0
- MIPS has a set of “pseudo-instructions” to make programming easier
  - More intuitive to read, but get translated into actual instructions later
- Example:
  
  ```
  move dst, src translated into
  addi dst, src, 0
  ```

Assembler Pseudo-Instructions

- List of pseudo-instructions:
  - List also includes instruction translation
- Load Address (la)
  - ```
    la dst, label
  ```
  - Loads address of specified label into \(dst\)
- Load Immediate (li)
  - ```
    li dst, imm
  ```
  - Loads 32-bit immediate into \(dst\)
- MARS has additional pseudo-instructions
  - See Help (F1) for full list

Assembler Register

- Problem:
  - When breaking up a pseudo-instruction, the assembler may need to use an extra register
  - If it uses a regular register, it’ll overwrite whatever the program has put into it
- Solution:
  - Reserve a register (\(\$1\) or \(\$at\) for “assembler temporary”) that assembler will use to break up pseudo-instructions
  - Since the assembler may use this at any time, it’s not safe to code with it

MAL vs. TAL

- True Assembly Language (TAL)
  - The instructions a computer understands and executes
- MIPS Assembly Language (MAL)
  - Instructions the assembly programmer can use (includes pseudo-instructions)
  - Each MAL instruction becomes 1 or more TAL instruction
  - \(TAL \subset MAL\)

Summary

- I-Format: instructions with immediates, \(lw/sw\) (offset is immediate), and \(beq/bne\)
  - But not the shift instructions
  - Branches use PC-relative addressing
    
    | I:      | opcode | rs | rt | immediate |
    |--------|--------|----|----|----------|

- J-Format: \(j\) and \(jal\) (but not \(jr\))
  - Jumps use absolute addressing
    
    | J:      | opcode | target address |
    |--------|--------|----------------|

- R-Format: all other instructions
  
<table>
<thead>
<tr>
<th>R:</th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

BonuS Slides

You are responsible for the material contained on the following slides, though we may not have enough time to get to them in lecture.

They have been prepared in a way that should be easily readable.
Agenda

- I-Format
  - Branching and PC-Relative Addressing
- Administrivia
- J-Format
- Pseudo-instructions
- Bonus: Assembly Practice
- Bonus: Disassembly Practice

Assembly Practice

- Assembly is the process of converting assembly instructions into machine code
- On the following slides, there are 6-lines of assembly code, along with space for the machine code
- For each instruction,
  1) Identify the instruction type (R/I/J)
  2) Break the space into the proper fields
  3) Write field values in decimal
  4) Convert fields to binary
  5) Write out the machine code in hex
- Use your Green Sheet; answers follow

Code Questions

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction</th>
<th>Material from past lectures:</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>Loop: sll $t1,$s3,2</td>
<td>What type of C variable is probably stored in $s6?</td>
</tr>
<tr>
<td>804</td>
<td>addu $t1,$t1,$s6</td>
<td>Write an equivalent C loop using a⇒$s3, b⇒$s5, c⇒$s6. Define variable types (assume they are initialized somewhere) and feel free to introduce other variables as you like.</td>
</tr>
<tr>
<td>808</td>
<td>lw $t0,0($t1)</td>
<td></td>
</tr>
<tr>
<td>812</td>
<td>beq $t0,$s5, Exit</td>
<td></td>
</tr>
<tr>
<td>816</td>
<td>addiu $s3,$s3,1</td>
<td>In English, what does this loop do?</td>
</tr>
<tr>
<td>820</td>
<td>j Loop</td>
<td></td>
</tr>
</tbody>
</table>

Exit:

Material from past lectures: What type of C variable is probably stored in $s6?

int * (or any pointer)

Write an equivalent C loop using a⇒$s3, b⇒$s5, c⇒$s6. Define variable types (assume they are initialized somewhere) and feel free to introduce other variables as you like.

In English, what does this loop do?

Finds an entry in array c that matches b.

Assembly Practice Question

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>Loop: sll $t1,$s3,2</td>
<td></td>
</tr>
<tr>
<td>804</td>
<td>addu $t1,$t1,$s6</td>
<td></td>
</tr>
<tr>
<td>808</td>
<td>lw $t0,0($t1)</td>
<td></td>
</tr>
<tr>
<td>812</td>
<td>beq $t0,$s5, Exit</td>
<td></td>
</tr>
<tr>
<td>816</td>
<td>addiu $s3,$s3,1</td>
<td></td>
</tr>
<tr>
<td>820</td>
<td>j Loop</td>
<td></td>
</tr>
</tbody>
</table>

Exit:

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>Loop: sll $t1,$s3,2</td>
<td></td>
</tr>
<tr>
<td>804</td>
<td>addu $t1,$t1,$s6</td>
<td></td>
</tr>
<tr>
<td>808</td>
<td>lw $t0,0($t1)</td>
<td></td>
</tr>
<tr>
<td>812</td>
<td>beq $t0,$s5, Exit</td>
<td></td>
</tr>
<tr>
<td>816</td>
<td>addiu $s3,$s3,1</td>
<td></td>
</tr>
<tr>
<td>820</td>
<td>j Loop</td>
<td></td>
</tr>
</tbody>
</table>

Exit:

Assembly Practice Answer (1/4)

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>Loop: sll $t1,$s3,2</td>
<td></td>
</tr>
<tr>
<td>804</td>
<td>addu $t1,$t1,$s6</td>
<td></td>
</tr>
<tr>
<td>808</td>
<td>lw $t0,0($t1)</td>
<td></td>
</tr>
<tr>
<td>812</td>
<td>beq $t0,$s5, Exit</td>
<td></td>
</tr>
<tr>
<td>816</td>
<td>addiu $s3,$s3,1</td>
<td></td>
</tr>
<tr>
<td>820</td>
<td>j Loop</td>
<td></td>
</tr>
</tbody>
</table>

Exit:
Assembly Practice Answer (2/4)

Addr | Instruction
-----|-------------------
800  | Loop: sl $t1,$s3,2
804  | addu $t1,$t1,$s6
808  | lw $t0,0($t1)
812  | beq $t0,$s5, Exit
816  | addiu $s3,$s3,1
820  | j Loop

Exit:

Assembly Practice Answer (3/4)

Addr | Instruction
-----|-------------------
800  | Loop: sl $t1,$s3,2
804  | addu $t1,$t1,$s6
808  | lw $t0,0($t1)
812  | beq $t0,$s5, Exit
816  | addiu $s3,$s3,1
820  | j Loop

Exit:

Disassembly Practice

- Disassembly is the opposite process of figuring out the instructions from the machine code
- On the following slides, there are 6-lines of machine code (hex numbers)
- Your task:
  1) Convert to binary
  2) Use opcode to determine format and fields
  3) Write field values in decimal
  4) Convert fields MIPS instructions (try adding labels)
  5) Translate into C (be creative!)
- Use your Green Sheet; answers follow

Disassembly Practice Question

Address | Instruction
---------|-------------------
0x00400000 | 0x00001025
...        | 0x0005402A
0x11000003 | 0x00441020
0x20A5FFFF | 0x08100001

Agenda

- I-Format
  - Branching and PC-Relative Addressing
- Administrivia
- J-Format
- Pseudo-instructions
- Bonus: Assembly Practice
  - Bonus: Disassembly Practice
Disassembly Practice Answer (1/9)

Address Instruction
0x00400000 00000000000000000001000000100101 ...
00000000000000000000000000000011
00000000000000000000000000000010
00100000100000000000000000000011
00100000100000000000000000000010
00100000100000000000000000000001
00010000010000000000000000000000

1) Converted to binary

Disassembly Practice Answer (2/9)

Address Instruction
0x00400000 R 00000000000000000001000000100101 ...
R 00000000000000000001000000100101
I 00100000100000000000000000000011
I 00100000100000000000000000000010
J 00010000010000000000000000000000

2) Check opcode for format and fields...
   – 0 (R-Format), 2 or 3 (J-Format), otherwise (I-Format)

Disassembly Practice Answer (3/9)

Address Instruction
0x00400000 or $v0,$0,$0
0x00400004 slt $t0,$0,$a1
0x00400008 beq $t0,$0, Exit
0x00400010 add $v0,$v0,$a0
0x00400014 j Loop

3) Convert to decimal
   – Can leave target address in hex

Disassembly Practice Answer (4/9)

Address Instruction
0x00400000 or $v0,$0,$0
0x00400004 slt $t0,$0,$a1
0x00400008 beq $t0,$0, Exit
0x0040000C add $v0,$v0,$a0
0x00400010 addi $a1,$a1,-1
0x00400014 j 0x0100001
0x00400018

4) Translate to MIPS instructions (write in addrs)
   – More readable with register names

Disassembly Practice Answer (5/9)

Address Instruction
0x00400000 or $v0,$0,$0
0x00400004 slt $t0,$0,$a1
0x00400008 beq $t0,$0, Exit
0x0040000C add $v0,$v0,$a0
0x00400010 addi $a1,$a1,-1
0x00400014 j Loop
0x00400018 Exit:

4) Translate to MIPS instructions (write in addrs)
   – Introduce labels

Disassembly Practice Answer (6/9)
Disassembly Practice Answer (7/9)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>or $v0,$0,$0</td>
<td># initialize $v0 to 0</td>
</tr>
<tr>
<td>Loop: slt $t0,$0,$a1</td>
<td># $t0 = 0 if 0 &gt;= $a1</td>
</tr>
<tr>
<td>beq $t0,$0, Exit</td>
<td># exit if $a1 &lt;= 0</td>
</tr>
<tr>
<td>add $v0,$v0,$a0</td>
<td># $v0 += $a0</td>
</tr>
<tr>
<td>addi $a1,$a1,-1</td>
<td># decrement $a1</td>
</tr>
<tr>
<td>Exit: j Loop</td>
<td></td>
</tr>
</tbody>
</table>

4) Translate to MIPS instructions (write in addr)
– What does it do?

Disassembly Practice Answer (8/9)

/* a→$v0, b→$a0, c→$a1 */

a = 0;
while(c > 0) {
    a += b;
    c--;
}

5) Translate into C code
– Initial direct translation

Disassembly Practice Answer (9/9)

/* naïve multiplication: returns m*n */
int multiply(int m, int n) {
    int p; /* product */
    for(p = 0; n-- > 0; p += m) ;
    return p;
}

5) Translate into C code
– One of many possible ways to write this