CS 61C: Great Ideas in Computer Architecture

MIPS Instruction Formats

Instructor: Justin Hsia
Review of Last Lecture

• New registers: $a0-$a3, $v0-$v1, $ra, $sp
  – Also: $at, $k0-$k1, $gp, $fp, PC

• New instructions: slt, la, li, jal, jr

• Saved registers: $s0-$s7, $sp, $ra

Volatile registers: $t0-$t9, $v0-$v1, $a0-$a3

  – CalleR saves volatile registers it is using before making a procedure call

  – CalleE saves saved registers it intends to use
int factorial(int n) {
    if(n == 0) return 1;
    else return (n*factorial(n-1));
}

(A) This function must be implemented recursively

(B) We can write this function without using any saved or temporary registers

(C) We must save $ra on the stack since we need to know where to return to

(D) We could copy $a0 to $a1 to store n across recursive calls instead of saving it
Great Idea #1: Levels of Representation/Interpretation

**Higher-Level Language Program (e.g. C)**

```
Compiler
```

```
Assembly Language Program (e.g. MIPS)
```

```
Assembler
```

```
Machine Language Program (MIPS)
```

```
Machine Interpretation
```

```
Hardware Architecture Description (e.g. block diagrams)
```

```
Architecture Implementation
```

```
Logic Circuit Description (Circuit Schematic Diagrams)
```

```

temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

We are here.
Agenda

• **Stored-Program Concept**
• R-Format
• Administrivia
• I-Format
  – Branching and PC-Relative Addressing
• J-Format
• **Bonus:** Assembly Practice
• **Bonus:** Disassembly Practice
Big Idea: Stored-Program Concept

• Encode your instructions as binary data
  – Therefore, entire programs can be stored in memory to be read or written just like data

• Simplifies SW/HW of computer systems
  – Memory technology for data also used for programs

• Stored in memory, so both instructions and data words have addresses
  – Use with jumps, branches, and loads
Binary Compatibility

• Programs are distributed in binary form
  – Programs bound to specific instruction set
  – i.e. different versions for (old) Macs vs. PCs

• New machines want to run old programs (“binaries”) as well as programs compiled to new instructions

• Leads to “backward compatible” instruction sets that evolve over time
  – The selection of Intel 80x86 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set (Pentium 4); you could still run program from 1981 PC today
Instructions as Numbers (1/2)

• Currently all data we work with is in words (32-bit blocks)
  – Each register is a word in length
  – \texttt{lw} and \texttt{sw} both access one word of memory

• So how do we represent instructions?
  – Remember: computer only understands 1s and 0s, so “add \$t0, \$0, \$0” is meaningless.
  – MIPS wants simplicity: since data is in words, let instructions be in words, too
Instructions as Numbers (2/2)

• Divide the 32 bits of an instruction into “fields”
  – Each field tells the processor something about the instruction
  – Could use different fields for every instruction, but regularity leads to simplicity

• Define 3 types of instruction formats:
  – R-Format
  – I-Format
  – J-Format
Instruction Formats

• **I-Format**: instructions with immediates, 
  \( \text{lw/sw} \) (offset is immediate), and \( \text{beq/bne} \)
  — But not the shift instructions

• **J-Format**: \( j \) and \( jal \)
  — But not \( jr \)

• **R-Format**: all other instructions
Agenda

• Stored-Program Concept
• R-Format
• Administrivia
• I-Format
  – Branching and PC-Relative Addressing
• J-Format
• Bonus: Assembly Practice
• Bonus: Disassembly Practice
R-Format Instructions (1/4)

• Define “fields” of the following number of bits each: $6 + 5 + 5 + 5 + 5 + 5 + 6 = 32$

• Each field has a name:

• Each field is viewed as its own unsigned int
  – 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63
R-Format Instructions (2/4)

- **opcode** (6): partially specifies operation
  - Set at 0b000000 for all R-Format instructions
- **funct** (6): combined with opcode, this number exactly specifies the instruction

- How many R-format instructions can we encode?
  - opcode is fixed, so 64

- Why aren’t these a single 12-bit field?
  - We’ll answer this later
R-Format Instructions (3/4)

- **rs (5):** specifies register containing 1\textsuperscript{st} operand ("source register")
- **rt (5):** specifies register containing 2\textsuperscript{nd} operand ("target register" – name is misleading)
- **rd (5):** specifies register that receives the result of the computation ("destination register")

- **Recall:** MIPS has 32 registers
  - Fit perfectly in a 5-bit field (use register numbers)

- These map intuitively to instructions
  - e.g. `add dst, src1, src2` → `add rd, rs, rt`
  - Depending on instruction, field may not be used
R-Format Instructions (4/4)

• **shamt** (5): The amount a shift instruction will shift by
  – Shifting a 32-bit word by more than 31 is useless
  – This field is set to 0 in all but the shift instructions

• For a detailed description of field usage and instruction type for each instruction, see the MIPS Green Card
R-Format Example (1/2)

- **MIPS Instruction:**
  \[
  \text{add} \quad \$8,\$9,\$10
  \]

- **Pseudo-code (“OPERATION” column):**
  \[
  \text{add} \quad R[rd] = R[rs] + R[rt]
  \]

- **Fields:**
  \[
  \begin{align*}
  \text{opcode} &= 0 \quad \text{(look up on Green Sheet)} \\
  \text{funct} &= 32 \quad \text{(look up on Green Sheet)} \\
  \text{rd} &= 8 \quad \text{(destination)} \\
  \text{rs} &= 9 \quad \text{(first operand)} \\
  \text{rt} &= 10 \quad \text{(second operand)} \\
  \text{shamt} &= 0 \quad \text{(not a shift)}
  \end{align*}
  \]
R-Format Example (2/2)

- **MIPS Instruction:** `add $8, $9, $10`

Field representation (decimal):

```
31  9  10  8  0  32
```

Field representation (binary):

```
31 01001 01010 01000 00000 100000
```

hex representation: `0x012A4020`

decimal representation: `19,546,144`

Called a **Machine Language Instruction**
NOP (or NOOP)

• What is the instruction 0x00000000?  
  – opcode is 0, so is an R-Format

• Using Green Sheet, translates into:
  sll $0, $0, 0
  – What does this do? Nothing!

• This is a special instruction called nop (or noop) for “No Operation Performed”  
  – We’ll see its uses later in the course
Agenda

• Stored-Program Concept
• R-Format
• Administrivia
• I-Format
  – Branching and PC-Relative Addressing
• J-Format
• Bonus: Converting to Machine Code Practice
Administrivia

• Long weekend due to July 4, but HW2 due Fri, HW 3 due Sun

• Project 1 due 7/14
  – No homework next week
  – Will be released in the next two days

• I will be sitting in on discussions today
  – For TA critiques, so don’t mind me
Agenda

• Stored-Program Concept
• R-Format
• Administrivia
• I-Format
  – Branching and PC-Relative Addressing
• J-Format
• Bonus: Assembly Practice
• Bonus: Disassembly Practice
I-Format Instructions (1/4)

- What about instructions with immediates?
  - 5- and 6-bit fields too small for most immediates
- Ideally, MIPS would have only one instruction format (for simplicity)
  - Unfortunately here we need to compromise
- Define new instruction format that is partially consistent with R-Format
  - First notice that, if instruction has immediate, then it uses at most 2 registers
I-Format Instructions (2/4)

• Define “fields” of the following number of bits each: $6 + 5 + 5 + 16 = 32$ bits

• Field names:

• **Key Concept:** Three fields are consistent with R-Format instructions
  
  – Most importantly, $\text{opcode}$ is still in same location
I-Format Instructions (3/4)

• **opcode** (6): uniquely specifies the instruction
  – All I-Format instructions have non-zero opcode
  – R-Format has two 6-bit fields to identify instruction for consistency across formats
• **rs** (5): specifies a register operand
  – Not always used
• **rt** (5): specifies register that receives result of computation ("target register")
  – Name makes more sense for I-Format instructions
I-Format Instructions (4/4)

• **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 $slti$
I-Format Example (1/2)

• MIPS Instruction:
  \[ \text{addi } \$21,\$22,-50 \]

• Pseudo-code (“OPERATION” column)
  \[ \text{addi } R[rt] = R[rs] + \text{SignExtImm} \]

• Fields:
  \[ \begin{align*}
  \text{opcode} &= 8 & \text{(look up on Green Sheet)} \\
  rs &= 22 & \text{(register containing operand)} \\
  rt &= 21 & \text{(target register)} \\
  \text{immediate} &= -50 & \text{(decimal by default, can also be specified in hex)}
  \end{align*} \]
I-Format Example (2/2)

- **MIPS Instruction:** `addi $21, $22, -50`

  **Field representation (decimal):**
  
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>22</td>
<td>21</td>
<td>-50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  **Field representation (binary):**
  
  | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
  |-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
  | 001000 | 10110 | 10101 | 1111111111001110 |

  **hex representation:** `0x22D5 FFCE`

  **decimal representation:** `584,449,998`
Question: Which instruction has the same representation as \(35_{\text{ten}}\)?

**OPCODE/FUNCT:**
- subu 0/35
- lw 35/--
- addi 8/--

**Register names and numbers:**
- 0: $0$
- 8–15: $t0$–$t7$
- 16–23: $s0$–$s7$

(A) subu $s0,$s0,$s0
(B) lw $0,0($0)
(C) addi $0,$0,35
(D) subu $0,$0,$0

---

**Instruction Representation:**
- **opcode**
- **rs**
- **rt**
- **rd**
- **shamt**
- **funct**
- **offset**
- **immediate**
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 \( (\text{lui}) \)
  – \text{lui \ reg,imm}
  – Moves 16-bit \text{imm} into upper half (bits 16-31) of \text{reg} and zeros the lower half (bits 0-15)
lui Example

• Want: addi $t0,$t0,0xABABCD
  – This is a pseudo-instruction!

• Translates into:

  lui $at,0xabab  # upper 16
  ori $at,$at,0xcDCD  # lower 16
  add $t0,$t0,$at  # move

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

Only the assembler gets to use $at
Branching Instructions

• beq and bne
  – Need to specify an address to go to
  – Also take two registers to compare

• Use I-Format:

<table>
<thead>
<tr>
<th>31</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
</tr>
</tbody>
</table>

  – 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 $\pm 2^{15}$ 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 a 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 \(\pm 2^{15}\) 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
Branch Example (1/2)

• MIPS Code:

   Loop: \texttt{beq $9,$0, End}\n   \texttt{addu $8,$8,$10}\n   \texttt{addiu $9,$9,-1}\n   \texttt{j Loop}\n
   End: <some instr>

• I-Format fields:

   \texttt{opcode} = 4 \hspace{2cm} \text{(look up on Green Sheet)}
   \texttt{rs} = 9 \hspace{2cm} \text{(first operand)}
   \texttt{rt} = 0 \hspace{2cm} \text{(second operand)}
   \texttt{immediate} = 3

Start counting from instruction AFTER the branch
Branch Example (2/2)

• MIPS Code:

```mips
Loop:  beq $9,$0, End
       addu $8,$8,$10
       addiu $9,$9,-1
ej   Loop
   End:
```

Field representation (decimal):

```
  31  4  9  0  3
```

Field representation (binary):

```
  31  000100 01001 00000 00000000000000000011
```
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:

```assembly
beq $s0,$0,far
# next instr -->

bne $s0,$0,next

# next instr

j   far

next: # next instr
```
Get To Know Your Staff

• Category: Games
Agenda

• Stored-Program Concept
• R-Format
• Administrivia
• I-Format
  – Branching and PC-Relative Addressing
• J-Format
• Bonus: Assembly Practice
• Bonus: Disassembly Practice
J-Format Instructions (1/4)

• For branches, we assumed that we won’t want to branch too far, so we can specify a change in the PC

• For general jumps (j and jal), we may jump to anywhere in memory
  – Ideally, we would specify a 32-bit memory address to jump to
  – Unfortunately, we can’t fit both a 6-bit opcode and a 32-bit address into a single 32-bit word
J-Format Instructions (2/4)

- Define two “fields” of these bit widths:

```
31  6  26  0
```

- As usual, each field has a name:

```
31 31 0
```

- **Key Concepts:**
  - Keep `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 $2^{26}$ addresses
  – Still going to word-aligned instructions, so add $0b00$ 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 256MiB 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, 0b00\}

• 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) Jump instructions don’t require any changes
2) Branch instructions don’t require any changes
Summary

- The Stored Program concept is very powerful
  - Instructions can be treated and manipulated the same way as data in both hardware and software

- MIPS Machine Language 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>
<tbody>
<tr>
<td>I:</td>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
<tr>
<td>J:</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>target address</td>
</tr>
</tbody>
</table>

- Branches use PC-relative addressing,
  Jumps use absolute addressing
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 and the material will be touched upon in the following lecture.
Agenda

• Stored-Program Concept
• R-Format
• Administrivia
• I-Format
  – Branching and PC-Relative Addressing
• J-Format
• 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>800</td>
<td>Loop: sll $t1,$s3,2</td>
</tr>
<tr>
<td>804</td>
<td>804</td>
<td>addu $t1,$t1,$s6</td>
</tr>
<tr>
<td>808</td>
<td>808</td>
<td>lw $t0,0($t1)</td>
</tr>
<tr>
<td>812</td>
<td>812</td>
<td>beq $t0,$s5, Exit</td>
</tr>
<tr>
<td>816</td>
<td>816</td>
<td>addiu $s3,$s3,1</td>
</tr>
<tr>
<td>820</td>
<td>820</td>
<td>j Loop</td>
</tr>
</tbody>
</table>

Exit:
**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>int * (or any pointer)</td>
</tr>
<tr>
<td>808</td>
<td>lw $t0,0($t1)</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>812</td>
<td>beq $t0,$s5, Exit</td>
<td>int a,b,*c;</td>
</tr>
<tr>
<td>816</td>
<td>addiu $s3,$s3,1</td>
<td>/* values initialized */ while(c[a] != b) a++;</td>
</tr>
<tr>
<td>820</td>
<td>j Loop</td>
<td>In English, what does this loop do? Finds an entry in array c that matches b.</td>
</tr>
</tbody>
</table>
### Assembly Practice Question

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

Entry: Exit:
Assembly Practice Answer (1/4)

**Addr**  **Instruction**

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

**R:**

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

**I:**

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
</table>

**J:**

<table>
<thead>
<tr>
<th>opcode</th>
<th>target address</th>
</tr>
</thead>
</table>

Exit:
### Assembly Practice Answer (2/4)

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction</th>
<th>R:</th>
<th>I:</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>Loop: sll $t1,$s3,2</td>
<td>0 0 19 9 2 0</td>
<td></td>
</tr>
<tr>
<td>804</td>
<td>addu $t1,$t1,$s6</td>
<td>0 9 22 9 0 33</td>
<td></td>
</tr>
<tr>
<td>808</td>
<td>lw $t0,0($t1)</td>
<td></td>
<td>35 9 8 0</td>
</tr>
<tr>
<td>812</td>
<td>beq $t0,$s5, Exit</td>
<td></td>
<td>4 8 21 2</td>
</tr>
<tr>
<td>816</td>
<td>addiu $s3,$s3,1</td>
<td></td>
<td>8 19 19 1</td>
</tr>
<tr>
<td>820</td>
<td>j Loop</td>
<td></td>
<td>2 200</td>
</tr>
</tbody>
</table>

**Exit:**
### Assembly Practice Answer (3/4)

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction</th>
<th>R:</th>
<th>I:</th>
<th>J:</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>Loop: sll $t1,$s3,2</td>
<td><img src="image" alt="Binary Representation" /></td>
<td><img src="image" alt="Binary Representation" /></td>
<td><img src="image" alt="Binary Representation" /></td>
</tr>
<tr>
<td>804</td>
<td>addu $t1,$t1,$s6</td>
<td><img src="image" alt="Binary Representation" /></td>
<td><img src="image" alt="Binary Representation" /></td>
<td><img src="image" alt="Binary Representation" /></td>
</tr>
<tr>
<td>808</td>
<td>lw $t0,0($t1)</td>
<td><img src="image" alt="Binary Representation" /></td>
<td><img src="image" alt="Binary Representation" /></td>
<td><img src="image" alt="Binary Representation" /></td>
</tr>
<tr>
<td>812</td>
<td>beq $t0,$s5, Exit</td>
<td><img src="image" alt="Binary Representation" /></td>
<td><img src="image" alt="Binary Representation" /></td>
<td><img src="image" alt="Binary Representation" /></td>
</tr>
<tr>
<td>816</td>
<td>addiu $s3,$s3,1</td>
<td><img src="image" alt="Binary Representation" /></td>
<td><img src="image" alt="Binary Representation" /></td>
<td><img src="image" alt="Binary Representation" /></td>
</tr>
<tr>
<td>820</td>
<td>j Loop</td>
<td><img src="image" alt="Binary Representation" /></td>
<td><img src="image" alt="Binary Representation" /></td>
<td><img src="image" alt="Binary Representation" /></td>
</tr>
</tbody>
</table>

**Exit:**
## Assembly Practice Answer (4/4)

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction</th>
<th>R:</th>
<th>I:</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>Loop: sll $t1,$s3,2</td>
<td>0x 0013 4880</td>
<td>0x 8D28 0000</td>
</tr>
<tr>
<td>804</td>
<td>addu $t1,$t1,$s6</td>
<td>0x 0136 4821</td>
<td>0x 1115 0002</td>
</tr>
<tr>
<td>808</td>
<td>lw $t0,0($t1)</td>
<td>0x 0800 00C8</td>
<td></td>
</tr>
<tr>
<td>812</td>
<td>beq $t0,$s5, Exit</td>
<td>0x 2273 0001</td>
<td></td>
</tr>
<tr>
<td>816</td>
<td>addiu $s3,$s3,1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>820</td>
<td>j Loop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Exit:

7/03/2013
Agenda

• Stored-Program Concept
• R-Format
• Administrivia
• I-Format
  – Branching and PC-Relative Addressing
• J-Format
• Bonus: Assembly Practice
• **Bonus: Disassembly Practice**
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400000</td>
<td>0x00001025</td>
</tr>
<tr>
<td>...</td>
<td>0x0005402A</td>
</tr>
<tr>
<td></td>
<td>0x11000003</td>
</tr>
<tr>
<td></td>
<td>0x00441020</td>
</tr>
<tr>
<td></td>
<td>0x20A5FFFF</td>
</tr>
<tr>
<td></td>
<td>0x08100001</td>
</tr>
</tbody>
</table>
Disassembly Practice Answer (1/9)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400000</td>
<td>0000000000000000000000100000000100101</td>
</tr>
<tr>
<td>...</td>
<td>000000000000000000000010101010000000000101010</td>
</tr>
<tr>
<td></td>
<td>00010001000000000000000000000000000011</td>
</tr>
<tr>
<td></td>
<td>00000000010001000000100000001000000000</td>
</tr>
<tr>
<td></td>
<td>00100000101001011111111111111111</td>
</tr>
<tr>
<td></td>
<td>00001000000100000000000000000000001</td>
</tr>
</tbody>
</table>

1) Converted to binary
Disassembly Practice Answer (2/9)

2) Check opcode for format and fields...
   – 0 (R-Format), 2 or 3 (J-Format), otherwise (I-Format)
### Disassembly Practice Answer (3/9)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400000</td>
<td>R 0 0 0 2 0 37</td>
</tr>
<tr>
<td></td>
<td>R 0 0 5 8 0 42</td>
</tr>
<tr>
<td></td>
<td>I 4 8 0 +3</td>
</tr>
<tr>
<td></td>
<td>R 0 2 4 2 0 32</td>
</tr>
<tr>
<td></td>
<td>I 8 5 5 -1</td>
</tr>
<tr>
<td></td>
<td>J 2 0x0100001</td>
</tr>
</tbody>
</table>

3) Convert to decimal
   – Can leave target address in hex
### Disassembly Practice Answer (4/9)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Instruction</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400000</td>
<td>or</td>
<td>$2, $0, $0</td>
<td>0x00400004</td>
<td>slt</td>
</tr>
<tr>
<td>0x00400008</td>
<td>beq</td>
<td>$8, $0, $3</td>
<td>0x0040000C</td>
<td>add</td>
</tr>
<tr>
<td>0x00400010</td>
<td>addi</td>
<td>$5, $5, $-1</td>
<td>0x00400014</td>
<td>j</td>
</tr>
<tr>
<td>0x00400018</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4) **Translate to MIPS instructions (write in addrs)**
### Disassembly Practice Answer (5/9)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400000</td>
<td><code>or $v0,$0,$0</code></td>
</tr>
<tr>
<td>0x00400004</td>
<td><code>slt $t0,$0,$a1</code></td>
</tr>
<tr>
<td>0x00400008</td>
<td><code>beq $t0,$0,3</code></td>
</tr>
<tr>
<td>0x0040000C</td>
<td><code>add $v0,$v0,$a0</code></td>
</tr>
<tr>
<td>0x00400010</td>
<td><code>addi $a1,$a1,-1</code></td>
</tr>
<tr>
<td>0x00400014</td>
<td><code>j 0x0100001</code></td>
</tr>
</tbody>
</table>

#### 4) Translate to MIPS instructions (write in addrs)

- More readable with register names
Disassembly Practice Answer (6/9)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400000</td>
<td>or $v0,$0,$0</td>
</tr>
<tr>
<td>0x00400004</td>
<td>Loop: slt $t0,$0,$a1</td>
</tr>
<tr>
<td>0x00400008</td>
<td>beq $t0,$0,Exit</td>
</tr>
<tr>
<td>0x0040000C</td>
<td>add $v0,$v0,$a0</td>
</tr>
<tr>
<td>0x00400010</td>
<td>addi $a1,$a1,-1</td>
</tr>
<tr>
<td>0x00400014</td>
<td>j Loop</td>
</tr>
<tr>
<td>0x00400018</td>
<td>Exit:</td>
</tr>
</tbody>
</table>

4) Translate to MIPS instructions (write in addrs)
   – Introduce labels
### Disassembly Practice Answer (7/9)

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

**Exit:**

4) Translate to MIPS instructions (write in addrs)

— What does it do?
Disassembly Practice Answer (8/9)

/* $v0$ → $a$, $a0$ → $b$, $a1$ → $c$ */
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