# 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

```
Addr   Instruction
 800   Loop: sll  $t1,$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:
```

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

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?

# Code Answers

```
Addr  Instruction
 800  Loop: sll  $t1,$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:
```

**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.

```
int a,b,*c;
/* values initialized */
while(c[a] != b)  a++;
```

In English, what does this loop do?

Finds an entry in array c that matches b.

# Assembly Practice Question

```
Addr   Instruction
 800   Loop: sll $t1,$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 (1/4)

**Addr Instruction**

```
800   Loop: sll $t1,$s3,2
```

R:

| opcode | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|

```
804   addu  $t1,$t1,$s6
```

R:

| opcode | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|

```
808   lw    $t0,0($t1)
```

I:

| opcode | rs | rt | immediate |
|---|---|---|---|

```
812   beq   $t0,$s5, Exit
```

I:

| opcode | rs | rt | immediate |
|---|---|---|---|

```
816   addiu $s3,$s3,1
```

I:

| opcode | rs | rt | immediate |
|---|---|---|---|

```
820   j     Loop
```

J:

| opcode | target address |
|---|---|

```
      Exit:
```

# Assembly Practice Answer (2/4)

**Addr Instruction**

```
800  Loop: sll  $t1,$s3,2
```

| R: | | | | | | |
|---|---|---|---|---|---|---|
| R: | 0 | 0 | 19 | 9 | 2 | 0 |

```
804  addu  $t1,$t1,$s6
```

| | | | | | | |
|---|---|---|---|---|---|---|
| R: | 0 | 9 | 22 | 9 | 0 | 33 |

```
808  lw    $t0,0($t1)
```

| | | | | |
|---|---|---|---|---|
| I: | 35 | 9 | 8 | 0 |

```
812  beq   $t0,$s5, Exit
```

| | | | | |
|---|---|---|---|---|
| I: | 4 | 8 | 21 | 2 |

```
816  addiu $s3,$s3,1
```

| | | | | |
|---|---|---|---|---|
| I: | 8 | 19 | 19 | 1 |

```
820  j     Loop
```

| | | |
|---|---|---|
| J: | 2 | 200 |

```
     Exit:
```

# Assembly Practice Answer (3/4)

| Addr | Instruction | | |
|---|---|---|---|
| 800 | Loop: sll | $t1,$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: | | |

Encodings:

| Format | | | | | | |
|---|---|---|---|---|---|---|
| R: (800) | 000000 | 00000 | 10011 | 01001 | 00010 | 000000 |
| R: (804) | 000000 | 01001 | 10110 | 01001 | 00000 | 100001 |
| I: (808) | 100011 | 01001 | 01000 | 0000 0000 0000 0000 | | |
| I: (812) | 000100 | 01000 | 10101 | 0000 0000 0000 0010 | | |
| I: (816) | 001000 | 10011 | 10011 | 0000 0000 0000 0001 | | |
| J: (820) | 000010 | 00 0000 0000 0000 0000 1100 1000 | | | | |

# Assembly Practice Answer (4/4)

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

# Agenda

- I-Format
  - Branching and PC-Relative Addressing
- Administrivia
- J-Format
- Pseudo-instructions
- 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

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

# Disassembly Practice Answer (1/9)

| Address | Instruction |
| --- | --- |
| `0x00400000` | 00000000000000000001000000100101 |
| `...` | 00000000000001010100000000101010 |
| | 00010001000000000000000000000011 |
| | 00000000010001000001000000100000 |
| | 00100000101001011111111111111111 |
| | 00001000000100000000000000000001 |

1) Converted to binary

# Disassembly Practice Answer (2/9)

| Address | | Instruction |
|---|---|---|
| 0x00400000 | **R** | 000000 00000 00000 00010 00000 100101 |
| ... | **R** | 000000 00000 00101 01000 00000 101010 |
| | **I** | 000100 01000 00000 0000000000000011 |
| | **R** | 000000 00010 00100 00010 00000 100000 |
| | **I** | 001000 00101 00101 1111111111111111 |
| | **J** | 000010 00000100000000000000000001 |

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 | R | 0 | 0 | 0 | 2 | 0 | 37 |
| ... | R | 0 | 0 | 5 | 8 | 0 | 42 |
| | I | 4 | 8 | 0 | +3 | | |
| | R | 0 | 2 | 4 | 2 | 0 | 32 |
| | I | 8 | 5 | 5 | -1 | | |
| | J | 2 | 0x0100001 | | | | |

3) Convert to decimal

– Can leave target address in hex

# Disassembly Practice Answer (4/9)

| Address | Instruction |
|---|---|
| `0x00400000` | `or   $2,$0,$0` |
| `0x00400004` | `slt  $8,$0,$5` |
| `0x00400008` | `beq  $8,$0,3` |
| `0x0040000C` | `add  $2,$2,$4` |
| `0x00400010` | `addi $5,$5,-1` |
| `0x00400014` | `j    0x0100001` |
| `0x00400018` | |

4) Translate to MIPS instructions (write in addrs)

# Disassembly Practice Answer (5/9)

| Address | Instruction |
|---|---|
| `0x00400000` | `or    $v0,$0,$0` |
| `0x00400004` | `slt   $t0,$0,$a1` |
| `0x00400008` | `beq   $t0,$0,3` |
| `0x0040000C` | `add   $v0,$v0,$a0` |
| `0x00400010` | `addi  $a1,$a1,-1` |
| `0x00400014` | `j     0x0100001 # addr: 0x0400004` |
| `0x00400018` | |

4) Translate to MIPS instructions (write in addrs)

– More readable with register names

# Disassembly Practice Answer (6/9)

| Address | Instruction |
|---|---|
| 0x00400000 | `        or   $v0,$0,$0` |
| 0x00400004 | `Loop:  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 (7/9)

Address Instruction

```
      or   $v0,$0,$0    # initialize $v0 to 0
Loop: slt  $t0,$0,$a1   # $t0 = 0 if 0 >= $a1
      beq  $t0,$0,Exit  # exit if $a1 <= 0
      add  $v0,$v0,$a0  # $v0 += $a0
      addi $a1,$a1,-1   # decrement $a1
      j    Loop
Exit:
```

4) Translate to MIPS instructions (write in addrs)

– 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