Today’s class

- More assembly language programming
Instruction formats

- Every MIPS instruction consists of a single 32-bit word aligned on a word boundary
- Three different instruction formats:
  - I type (immediate)
  - R type (register)
  - J type (jump)
### Instruction formats

<table>
<thead>
<tr>
<th>Register type</th>
<th>Immediate type</th>
<th>Jump type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>op</strong></td>
<td><strong>rs</strong></td>
<td><strong>rt</strong></td>
</tr>
<tr>
<td>Operation code</td>
<td>Source register specifier</td>
<td>Target register specifier</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Immediate type**
- **op** | **rs** | **rt** | **immediate**
- Operation code | Source register specifier | Target register specifier or branch condition | Immediate, branch displacement or address displacement |
- 6 | 5 | 16 |

**Jump type**
- **op** | **target**
- Operation code | Target address |
- 6 | 26 |
Parts of instruction

- **op** – 6-bit opcode \(2^6 = 64\) so 64 different operations are possible
- **rs** – 5-bit source register \(2^5 = 32\) so 32 different registers are possible
- **rd** – 5-bit destination register
- **rt** – 5-bit target register or branch condition
- **immediate** – 16-bit immediate value, branch displacement or address displacement
- **target** – 26-bit jump target address
- **shamt** – 5-bit shift amount
- **funct** – 6-bit function field (allows additional operations by combining with op field)
Register addressing

- Just indicate the register name/number as the operand
- Simplest and fastest addressing mode
- A form of direct addressing (the data is actually in the register)

Example:

```
add $t0, $t1, $t2
```
Register addressing

```
add $t0,$t1,$t2
```

function for add is 32
Base addressing

- A small constant is added to a pointer held in a register
- Gives you the ability to address fixed offsets from a base address of a data structure
- A form of indirect addressing since the operand is at the memory location whose address is in a register
- Examples (assuming a valid address is in $t2):
  
  ```
  lb $t0, ($t2)
  lw $t1, 4($t2)
  ```
Base addressing

[Diagram showing base addressing with a register and immediate values]

Op  rs  rt  immediate

Opcodes for lb is 32

lb $t0, ($t2)
Immediate addressing

- One operand is a constant within the instruction itself
- Does not require an extra memory access to fetch the operand
- Operand is limited to 16 bits in size
- Example:
  
  \[ \text{addi } \$t1, \$t1, 1 \]
Immediate addressing

```
001000 01001 01001 00000000000000001
```

```
addi $t1, $t1, 1
```

Op code for addi is 8
Immediate addressing

![Diagram of immediate addressing](image)

- Opcode for \( j \) is 2
- Destination 0x40003c. Add 2 zeros since instructions word-aligned
Program counter relative addressing

- Address is the sum of the program counter and a constant in the instruction
- Used for conditional branches
- Branch can only be 32768 instructions above or below the program counter because the offset is a 16-bit two’s complement number
- Example:

  \[ \text{beqz } t0, \text{strEnd} \]
Program counter relative addressing

![Diagram of program counter relative addressing]

- **Instruction**: `beqz $t0, strEnd`
- **Opcodes for `beq`**: 0b000100, 0b01000, 0b00000, 0b00000000000000100
- **Branch**: 4 instructions (16 bytes)

**Diagram Details**:
- **PC** (Program Counter)
- **Address Calculation**: `PC + address`
- **Opcode Fields**:
  - `op`: 6 bits
  - `rs`: 5 bits
  - `rt`: 5 bits
  - `address`: 16 bits
In-class exercise

- Write a program to sum the elements of an array
- Use the template sum.asm as a starting point
Shift instructions

- **sll** \( \text{reg}_{\text{dest}}, \text{reg}_{\text{src1}}, \text{src2} \)
  - Shift left logical
  - \( \text{reg}_{\text{src1}} \) shifted left by the distance specified in src2 and the result put in \( \text{reg}_{\text{dest}} \)
  - 0 fill the empty bits

- **srl** \( \text{reg}_{\text{dest}}, \text{reg}_{\text{src1}}, \text{src2} \)
  - Shift right logical
  - \( \text{reg}_{\text{src1}} \) shifted right by the distance specified in src2 and the result put in \( \text{reg}_{\text{dest}} \)
  - 0 fill the empty bits

- **sra** \( \text{reg}_{\text{dest}}, \text{reg}_{\text{src1}}, \text{src2} \)
  - Shift right arithmetic
  - \( \text{reg}_{\text{src1}} \) shifted right by the distance specified in src2 and the result put in \( \text{reg}_{\text{dest}} \)
  - sign fill the empty bits
Rotate instructions

- **rol** \( \text{reg}_{\text{dest}}, \text{reg}_{\text{src1}}, \text{src2} \)
  - Rotate left
  - \( \text{reg}_{\text{src1}} \) rotated left by the distance specified in \( \text{src2} \) and the result put in \( \text{reg}_{\text{dest}} \)

- **ror** \( \text{reg}_{\text{dest}}, \text{reg}_{\text{src1}}, \text{src2} \)
  - Rotate right
  - \( \text{reg}_{\text{src1}} \) rotated right by the distance specified in \( \text{src2} \) and the result put in \( \text{reg}_{\text{dest}} \)
Logical instructions

- and
- or
- xor
- nor
- not

All take 3 operands except not, which takes 2
Example program

- Look at uppercase.asm, which converts a string to all uppercase letters.
- Demo it.
The stack

- A stack is last in, first out data structure
- Special register, $sp$, points to top of the stack
- In MIPS, the stack is actually upside down, with the top of the stack growing toward lower memory addresses
Stack operations

- Push – add something to the stack, e.g. push the word in $t0 onto the stack
  
  sub $sp,$sp,4
  sw $t0,($sp)

- Pop – remove the top item from the stack, e.g. to pop the top of stack to $t0
  
  lw $t0,($sp)
  add $sp,$sp,4
Procedure calls

- Jump and link instruction (jal) puts the return address into register $ra, which is a special register to hold the return address.
- Procedure just needs to execute jr $ra at the end of the procedure to return.
If a procedure calls another procedure …

- The return address will be lost, as $ra will be overwritten
- The first procedure will have to save the return address in $ra on the stack before it calls the second procedure, and then restore it before returning

```
sub $sp, 4
lw  $ra,($sp)
sw  $ra,($sp)
add $sp, 4
jr  $ra
```
Passing parameters to procedures

- First four parameters are passed in the argument registers $a0..$a3
- Additional parameters are passed on the stack
- Return values are sent back in $v0 and $v1
Register usage in procedures

- A procedure may need to have local variables
- Best to keep these in registers for speed of access
- If one procedure calls another, it may lose the local variables’ values in the registers if the second procedure uses the registers as well
- Thus, need to save register values
- This can be done either by the calling procedure or the called procedure
MIPS conventions

- Registers $s0..s7$ are callee saves, they will be saved by the called procedure and restored before returning.

- Registers $t0..t9$ are caller saves, the procedure doing the calling must save them if there are values that are needed after the called procedure returns.
Stack frames

- A block of memory on the stack allocated at procedure entry to hold all local variables needed by the procedure

- A special register, $fp$, contains the address of the stack frame for the procedure

- Base mode addressing off $fp$ addresses local variables
Example program

- uppercaseProc.asm takes the string conversion to uppercase example and does the work in a procedure

- Demo