**Stored Program Concept**

- Instructions are bits
- Programs are stored in memory — to be read or written just like data

- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the "next" instruction and continue

**Instructions:**

- Language of the Machine
- More primitive than higher level languages e.g., no sophisticated control flow
- Very restrictive e.g., MIPS Arithmetic Instructions

- We'll be working with the MIPS instruction set architecture
  - similar to other architectures developed since the 1980's
  - used by NEC, Nintendo, Silicon Graphics, Sony

*Design goals: maximize performance and minimize cost, reduce design time*

**Architecture Specification**

- Data types:
  - bit, byte, bit field, signed/unsigned integers logical, floating point, character
- Operations:
  - data movement, arithmetic, logical, shift/rotate, conversion, input/output, control, and system calls
- # of operands:
  - 3, 2, 1, or 0 operands
- Registers:
  - integer, floating point, control
- Instruction representation as bit strings

**Characteristics of Instruction Set**

- Complete
  - Can be used for a variety of application
- Efficient
  - Useful in code generation
- Regular
  - Expected instruction should exist
- Compatible
  - Programs written for previous versions of machines need it
- Primitive
  - Basic operations
- Simple
  - Easy to implement
- Smaller
  - Implementation

**Example of multiple operands**

- Instructions may have 3, 2, 1, or 0 operands
- Number of operands may affect instruction length
- Operand order is fixed (destination first, but need not that way)

```
add $s0, $s1, $s2 ; Add $s2 and $s1 and store result in $s0
add $s0, $s1     ; Add $s1 and $s0 and store result in $s0
add $s0       ; Add contents of a fixed location to $s0
add            ; Add two fixed locations and store result
```

**Where operands are stored**

- Memory locations
  - Instruction include address of location
- Registers
  - Instruction include register number
- Stack location
  - Instruction opcode implies that the operand is in stack
- Fixed register
  - Like accumulator, or depends on inst
  - Hi and Lo register in MIPS
- Fixed location
  - Default operands like interrupt vectors
MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: \[ A = B + C \]
MIPS code: `add $s0, $s1, $s2`

(associated with variables by compiler)

Memory Organization

- Bytes are nice, but most data items use larger “words”
- For MIPS, a word is 32 bits or 4 bytes.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>S16 bits of data</td>
<td>S16 bits of data</td>
<td>S16 bits of data</td>
<td>S16 bits of data</td>
<td>S16 bits of data</td>
<td>S16 bits of data</td>
<td>S16 bits of data</td>
<td>S16 bits of data</td>
<td>S16 bits of data</td>
<td>S16 bits of data</td>
<td>S16 bits of data</td>
<td>S16 bits of data</td>
</tr>
</tbody>
</table>

Registers hold 32 bits of data

- 2^{16} bytes with byte addresses from 0 to 2^{16}-1
- 2^{16} words with byte addresses 0, 4, 8, … 2^{16-4}
- Words are aligned
  i.e., what are the least 2 significant bits of a word address?

Addressing within a word

- Each word has four bytes
- Which byte is first and which is last
  - Two Choices
    - Least significant byte is byte “0” -> Little Endian
    - Most significant byte is byte “0” -> Big Endian

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

_registers hold 32 bits of data_
Instructions

- Load and store instructions
- Example:
  
  
  MIPS code:
  
  \[
  \text{lw} \ $t0, 32(\$a3) \\
  \text{add} \ $t0, \ $a2, \ $t0 \\
  \text{sw} \ $t0, 32(\$a3)
  \]

- Store word has destination last
- Remember arithmetic operands are registers, not memory!

Addressing

- Memory address for load and store has two parts
  - A register whose content are known
  - An offset stored in 16 bits
- The offset can be positive or negative
  - It is written in terms of number of bytes
  - It is but in instruction in terms of number of words
  - 32 byte offset is written as 32 but stored as 8
- Address is content of register + offset
- All address has both these components
  - If no register needs to be used then use register 0
    - Register 0 always stores value 0
  - If no offset, then offset is 0

Our First Example

- Can we figure out the code?

```c
swap(int v[], int k):
{ int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

```assembly
swap:
   muli $2, $5, 4
   add $2, $4, $2
   lw $15, 0($2)
   lw $16, 4($2)
   sw $15, 0($2)
   sw $16, 4($2)
   jr $31
```

So far we’ve learned:

- MIPS
  - loading words but addressing bytes
  - arithmetic on registers only

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory[$s2+100]</td>
</tr>
<tr>
<td>sw $s1, 100($s2)</td>
<td>Memory[$s2+100] = $s1</td>
</tr>
</tbody>
</table>

Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: add $t0, $s1, $s2
    - registers have numbers, $t0=9, $s1=17, $s2=18
- Instruction Format:

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>00001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>10000</td>
</tr>
</tbody>
</table>

- Can you guess what the field names stand for?

Machine Language

- Consider the load-word and store-word instructions.
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: \( \text{lw} \ $t0, 32(\$a2) \)
**Control**

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed
- MIPS conditional branch instructions:
  
  ```
  bne $t0, $t1, Label
  beq $t0, $t1, Label
  ```

  - Example: if (i==j) h = i + j;
  
  ```
  bne $s0, $s1, Label
  add $s3, $s0, $s1
  Label: ....
  ```

**Conditional Execution**

- A simple conditional execution
- Depending on i==j or i!=j, result is different

**Instruction Sequencing**

- MIPS unconditional branch instructions:
  
  ```
  j label
  ```

  - Example:
    
    ```
    f, g, and h are in registers $s3, $s4, and $s5
    if (i==j) 
    beq $s4, $s5, Label1
    else if (i==h) 
    beq $s4, $s5, Label2
    ```

    ```
    exit: ...
    ```

  - Can you build a simple for loop?

**So far:**

- Instruction Meaning
  
  ```
  add $s1, $s2, $s3 $s1 = $s2 + $s3
  sub $s1, $s2, $s3 $s1 = $s2 - $s3
  lw $s1, 100($s2) $s1 = Memory[$s2+100]
  sw $s1, 100($s2) Memory[$s2+100] = $s1
  bne $s4, $s5, L Next instr. is at Label if $s4 ° $s5
  beq $s4, $s5, L Next instr. is at Label if $s4 = $s5
  j L Next instr. is at Label
  ```

  - Formats:
    
    ```
    R op rs rt rd shamt funct
    I op 26 bit address
    J op 26 bit address
    ```

**Control Flow**

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:
  
  ```
  if $s1 < $s2 then
  $t0 = 1
  else
  $t0 = 0
  ```

- Can use this instruction to build "blt $s1, $s2, Label"
  — can now build general control structures
- Note that the assembler needs a register to do this,
  — there are policy of use conventions for registers

**Constants**

- Small constants are used quite frequently (50% of operands)
  
  ```
  addi $29, $29, 4
  slti $8, $18, 10
  andi $29, $29, 6
  ori $29, $29, 4
  ```

  - How do we make this work?
Other Issues

- Things we are not going to cover
  - support for procedures
  - linkers, loaders, memory layout
  - stacks, frames, recursion
  - manipulating strings and pointers
  - interrupts and exceptions
  - system calls and conventions
- Some of these we'll talk about later
- We've focused on architectural issues
  - basics of MIPS assembly language and machine code
  - we'll build a processor to execute these instructions.

Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

<table>
<thead>
<tr>
<th>Format</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- rely on compiler to achieve performance
  - what are the compiler's goals?
- help compiler where we can

Various Addressing Modes

Addresses in Branches and Jumps

- Instructions:
  - bne $t4,$t5,Label Next instruction is at Label if $t4 != $t5
  - beq $t4,$t5,Label Next instruction is at Label if $t4 = $t5
  - j Label Next instruction is at Label

- Formats:
  - op rs rt 16 bit address
  - op 26 bit address

- Addresses are not 32 bits
  - How do we handle this with load and store instructions?

Addresses in Branches

- Instructions:
  - bne $t4,$t5,Label Next instruction is at Label if $t4 != $t5
  - beq $t4,$t5,Label Next instruction is at Label if $t4 = $t5

- Formats:
  - I op rs rt 16 bit address
  - J op 26 bit address

- Addresses are not 32 bits
  - How do we handle this with load and store instructions?

To summarize:

<table>
<thead>
<tr>
<th>MIPS operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>Probe</td>
</tr>
<tr>
<td>BNE</td>
</tr>
<tr>
<td>BEQ</td>
</tr>
<tr>
<td>J</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction format</th>
<th>Use</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>I op rs rt 16 bit address</td>
<td>16 bit address</td>
<td></td>
</tr>
<tr>
<td>J op 26 bit address</td>
<td>26 bit address</td>
<td></td>
</tr>
</tbody>
</table>

- Could specify a register (like lw and sw) and add it to address
  - use Instruction Address Register (PC = program counter)
  - most branches are local (principle of locality)
- Jump instructions just use high order bits of PC
  - address boundaries of 256 MB