### 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 contents of a fixed location to $s0
add $s0            ; 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: \[ \text{add} \ $s0, \ $s1, \ $s2 \]

(associated with variables by compiler)

Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: \[ \text{add} \ $t0, \ $s1, \ $s2 \]
  - registers have numbers, \( t0=9, \ $s1=17, \ $s2=18 \)
- Instruction Format:

  \[
  \begin{array}{cccccccc}
  0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
  0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  \end{array}
  \]

  \begin{array}{c}
  \text{op} \\
  \text{rs} \\
  \text{rt} \\
  \text{rd} \\
  \text{shamt} \\
  \text{funct}
  \end{array}

Registers vs. Memory

- Arithmetic instructions operands must be registers, only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables

Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.

\[
\begin{array}{cccccccc}
0 & 8 & 16 & 24 & 32 & 40 & 48 & 56 \\
\text{8 bits of data} & \text{8 bits of data} & \text{8 bits of data} & \text{8 bits of data} & \text{8 bits of data} & \text{8 bits of data} & \text{8 bits of data} & \text{8 bits of data}
\end{array}
\]
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"  \(\rightarrow\) Little Endian
  - Most significant byte is byte "0"  \(\rightarrow\) Big Endian

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

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

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: \(\text{l} \ w \ \$t0, \ 32(\$s2)\)

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: \(\text{l} \ w \ \$t0, \ 32(\$s2)\)

Instructions

- Load and store instructions
- Example:
  - MIPS code:
    - \(\text{lw} \ \$t0, \ 32(\$s3)\)
    - \(\text{add} \ \$t0, \ \$s2, \ \$t0\)
    - \(\text{sw} \ \$t0, \ 32(\$s3)\)
- Store word has destination last
- Remember arithmetic operands are registers, not memory!

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 $a1, $a2, $a3 \</td>
<td>$a1 = $a2 + $a3</td>
</tr>
<tr>
<td>sub $a1, $a2, $a3 \</td>
<td>$a1 = $a2 - $a3</td>
</tr>
<tr>
<td>lw $a1, 100($s2) \</td>
<td>$a1 = \text{Memory}[$s2+100]</td>
</tr>
<tr>
<td>sw $a1, 100($s2) \</td>
<td>\text{Memory}[$s2+100] = $a1</td>
</tr>
<tr>
<td>jr $s3</td>
<td></td>
</tr>
</tbody>
</table>

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;
}
```

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

So far we’ve learned:

- MIPS
  - Loading words but addressing bytes
  - Arithmetic on registers only

Instructions

- Load and store instructions
- Example:
  - MIPS code:
    - \(\text{lw} \ \$t0, \ 32(\$s3)\)
    - \(\text{add} \ \$t0, \ \$s2, \ \$t0\)
    - \(\text{sw} \ \$t0, \ 32(\$s3)\)
- Store word has destination last
- Remember arithmetic operands are registers, not memory!
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 or 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
  - j exit
  - if (i!=j) beq $s4, $s5, Label1
  - f=g-h; sub $s3, $s4, $s5
  - exit: ...
- Can you build a simple for loop?

Branch Address Handling

- Instructions:
  - `bne $t4, $t5, Label`
  - `beq $t4, $t5, Label`
- Formats:
  - `op rs rt 16 bit address`
- 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

So far:

- Instruction
  - `add $s1, $s2, $s3`
  - `$s1 = $s2 + $s3`
  - `sub $s1, $s2, $s3`
  - `$s1 = $s2 - $s3`
  - `lw $s4, 100($s2)`
  - `Memory[$s2+100] = $s4`
  - `bne $s4, $s5, Label`
  - `beq $s4, $s5, Label`
  - `j Label`

- Formats:
  - `R op rs rt rd shamt funct`
  - `I op rs rt 16 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
    - `slt $t0, $s1, $s2`
    - `$t0 = 1`
  - else
    - `$t0 = 0`
- Can use this instruction to build "bit $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
Small constants are used quite frequently (50% of operands)
e.g., $A = A + 5$;
$B = B + 1$;
$C = C - 18$;

Solutions? Why not?
– put "typical constants" in memory and load them.
– create hard-wired registers (like $zero$) for constants like one.

MIPS Instructions:
- addi $29, $29, 4
- slti $8, $18, 10
- andi $29, $29, 6
- or $29, $29, 4

How do we make this work?

To summarize:

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>values for results and expression evaluation</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>

Stack Manipulation
- Register $29$ is used as stack pointer
- Stack grows from high address to low address
- Stack pointer should point to the last filled address
- Once entries are removed, stack pointer should be adjusted

Frame Pointer
- Stores the last address for the last frame
- When completing a subroutine, frame address can be used as the starting stack pointer value
We'd like to be able to load a 32-bit constant into a register.

Must use two instructions, new "load upper immediate" instruction:

\[
lui \$t0, 1010101010101010
\]

\[
0000000000000000 1010101010101010
\]

Then must get the lower order bits right, i.e.,

\[
ori \$t0, \$t0, 1010101010101010
\]

\[
0000000000000000 1010101010101010
\]

How about larger constants?

Then must get the lower order bits right, i.e.,

\[
ori \$t0, \$t0, 1010101010101010
\]

\[
0000000000000000 1010101010101010 1010101010101010
\]

\[
1010101010101010 1010101010101010
\]

\[
1010101010101010 1010101010101010
\]

Branch offset depends on expected jump address

Many compromise are made based on number of bits in instruction

LW/R2, Av(R1); Load memory from address (R1) + v

SW/R2, Av(R1); Store memory to address (R1) + v

R-Type – OPER R3, R2, R1; Perform R3 e.g. R2 OP R1

Five operations ADD, AND, OR, SLT, SUB

Four operation ADDI, ANDI, ORI, SLTI

B-Type – BC R2, R1, V; Branch if condition met to address PC+V

Two operation BNE, BEQ

Shift class – SHIFT TYPE R2, R1; Shift R1 of type and result to R2

One operation

Jump Class – JAL and JR (JAL can be used for Jump)

What are the implications of J vs JAL

Two instructions

Shift type requires four bits

Two fields Opcode

One level opcode

In our example it is more optimal, 16 op codes are sufficient

Each register takes 4 bits to encode

Shift type requires four bits

To pack instructions in 16 bits

V to 4 bits

Branch offset 4 bits

How many bits in jump address?

Only 12 bits jump address required
Trade Offs

- Only 4 bit immediate value
  - It is ok as constants are usually small
- Only 4-bit LW/SW address offset
  - This is real small
  - Good for small programs
- 12-bit jump address
  - Not a real limitation
- Branch offset 4 bits
  - Has constraints, but can be managed with jump
  - Depends on types of program
- Instructions are few
  - It is a quite a complete instruction set
- The instruction set is slightly redundant

Instruction Format

```
15 12 11  8  7  4  3  0
  B-TYPE  REG 3  REG 2  REG 1
```
```
15 12 11  8  7  4  3  0
  I-TYPE  IMMED  REG 1  REG 0
```
```
15 12 11  8  7  4  3  0
  TYPE  IMMED  REG 1  REG 0
```
```
15 12 11  8  7  4  3  0
  TYPE  IMMED  REG 1  REG 0
```
```
15 12 11  8  7  4  3  0
  SHIFT  TYPE  REG 1  REG 0
```
```
15 12 11  8  7  4  3  0
  JR  NotUsed  NotUsed  REG 0
```
```
15 12 11  8  7  4  3  0
  JAL  JUMP ADDRESS
```

Operation for Each Instruction

<table>
<thead>
<tr>
<th></th>
<th>LW:</th>
<th>SW:</th>
<th>R/I/S-Type:</th>
<th>BR-Type:</th>
<th>JMP-Type:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>READ INST</td>
<td>1.</td>
<td>READ INST</td>
<td>1.</td>
<td>READ INST</td>
</tr>
<tr>
<td>2.</td>
<td>READ REG 1</td>
<td>2.</td>
<td>READ REG 1</td>
<td>2.</td>
<td>READ REG 1</td>
</tr>
<tr>
<td>READ REG 2</td>
<td>READ REG 2</td>
<td>READ REG 2</td>
<td>READ REG 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>ADD REG 1 + OFFSET</td>
<td>3.</td>
<td>OPERATE on REG 1 &amp; REG 2</td>
<td>3.</td>
<td>SUB REG 2 from REG 1</td>
</tr>
<tr>
<td>4.</td>
<td>READ MEM</td>
<td>4.</td>
<td>WRITE MEM</td>
<td>4.</td>
<td>WRITE MEM</td>
</tr>
<tr>
<td>5.</td>
<td>WRITE REG2</td>
<td>5.</td>
<td>WRITE DST</td>
<td>5.</td>
<td>WRITE DST</td>
</tr>
</tbody>
</table>

Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI
- Sometimes referred to as “RISC vs. CISC”
  - virtually all new instruction sets since 1982 have been RISC
  - VAX: minimize code size, make assembly language easy
    instructions from 1 to 54 bytes long!
- We’ll look at PowerPC and 80x86

PowerPC

- Indexed addressing
  - example: lw $t1,$a0+$s3 #$t1=Memory[$a0+$s3]
  - What do we have to do in MIPS?
- Update addressing
  - update a register as part of load (for marching through arrays)
  - example: lwu $t0,4($s3) #$t0=Memory[$s3+4];$s3=$s3+4
  - What do we have to do in MIPS?
- Others:
  - load multiple/store multiple
  - a special counter register “bc Loop”
  - decrement counter, if not 0 goes loop

80x86

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions
  (mostly designed for higher performance)
- 1997: MMX is added

“This history illustrates the impact of the “golden handcuffs” of compatibility
adding new features as someone might add clothing to a packed bag”
“an architecture that is difficult to explain and impossible to love”