Other architectures

Example. Accumulator-based machines

A single register, called the accumulator, stores the operand before the operation, and stores the result after the operation.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load x</td>
<td># into acc from memory</td>
</tr>
<tr>
<td>Add y</td>
<td># add y from memory to the acc</td>
</tr>
<tr>
<td>Store z</td>
<td># store acc to memory as z</td>
</tr>
</tbody>
</table>

Can we have an instruction

Add z, x, y # z := x + y, (x, y, z in memory) ?

For some machines, YES, not in MIPS.

MIPS is a load-store architecture. What is a load-store architecture?
**Load-store architecture**

Only load and store instructions access the memory, all other instructions use registers as operands. What is the motivation? Primary motivation is speedup – registers are faster.

**Reduced Instruction Set Computers (RISC)**

- The instruction set has only a small number of frequently used instructions. This lowers processor cost, without much impact on performance.
- All instructions have the same length.
- Load-store architecture.

Non-RISC machines are called CISC (Complex Instruction Set Computer). Example: Pentium
Another classification

3-address    add r1, r2, r3    (r1 ← r2 + r3)
2-address    add r1, r2         (r1 ← r1 + r2)
1-address    add r1             (to the accumulator)
0-address or stack machines    (see below)

Example of stack architecture

Push x
Push y
Push z
Add
Multiply
Pop z

Computes \( z = x \times (y + z) \)
Computer Arithmetic

How to represent negative integers? The most widely used convention is 2’s complement representation.

\[
\begin{align*}
+14 &= 0, 1 1 1 0 \\
-14 &= 1, 0 0 1 0
\end{align*}
\]

Largest integer represented using n-bits is \( +2^{n-1} - 1 \)
Smallest integer represented using n-bits is \( -2^{n-1} \)

Review binary-to decimal and binary-to-hex conversions.
Review BCD (Binary Coded Decimal) and ASCII codes.
How to represent fractions?
### Overflow

\[+12 = 0, 1 1 0 0\] \[+12 = 0, 1 1 0 0\]
\[+2 = 0, 0 0 1 0\] \[+7 = 0, 0 1 1 1\]
\[\text{add} \quad \quad \quad \quad \quad \quad \text{add}\]
\[+14 = 0, 1 1 1 0\] \[? = 1, 0 0 1 1\]

Addition of a positive and a negative number does not lead to overflow. How to detect overflow?

Some machines use the following instructions:

Add the operands \hspace{2em} \#sets condition codes
B (overflow) OVERFLOW \hspace{2em} \# conditional branch

\begin{array}{|c|c|c|c|}
\hline
Z & N & V & C \\
\hline
\end{array}

Condition code register

Not MIPS
Exceptions

MIPS instructions that detect overflow cause an exception (also called an interrupt), and transfer control to a predefined address to invoke a routine for handling the exception.

L: add $t0, $t1, $t2
Overflow! Return address (L+4) is saved in EPC
Next instruction

Exception handler routine
Ra ← EPC
ir ra

Exceptions cause unscheduled procedure calls.
The following sequence of MIPS instructions can detect overflow in signed addition:

```
addu $t0, $t1, $t2     # add but don’t trap
xor  $t3, $t1, $t2     # check if signs differ
slt  $t3, $t3, $zero   # $t3=1 if signs differ
bne  $t3 $zero, no_overflow
xor  $t3, $t0, $t1     # sum sign = operand sign?
slt  $t3, $t3, $zero   # if not, then $t3=1
bne  $t3 $zero, overflow

no_overflow:

overflow:
```

More Programming Examples

Copying a string

Each char is represented by an ASCII byte. In C, a string is terminated by a 0 (Null in ASCII). $s0 will hold the array index.

```
add $s0, $zero, $zero  # i = 0
L1: add $t1, $a1, $s0    # address of y[i] in t1
    lb $t2, 0($t1)     # t2 = y[i]
    add $t3, $a0, $s0  # address of x[i] in t3
    sb $t2, 0($t3)     # x[i] = y[i]
    addi $s0, $s0, 1   # i = i+1
    bne $t2,$zero, L1  # if y[i]=0 then goto L1
```
Loading a 32-bit constant into a register

lui $t0, 255
(load upper-byte immediate)

$\text{t}0$

\begin{verbatim}
0000 0000 1111 1111
\end{verbatim}

addi $\text{t}0$, $\text{t}0$, 15, (add immediate), or
ori $\text{t}0$, $\text{t}0$, 15 (or immediate)

$\text{t}0$

\begin{verbatim}
0000 0000 1111 1111 0000 0000 0000 1111
\end{verbatim}

This is how the “la” (pseudo-instruction) is implemented.
Example of shift instructions

sll $t2, $s0, 8   # $t2 = $s0 << 8 bits

(shift left logical) What is it used for?
(shift right logical) What is it used for?

Multiplication (or division) by $2^k$
Character manipulation etc.
Logic Design (Go to Appendix B)

When you write add $t0, $t1, $t2, you imagine something like this:

What kind of hardware can ADD two binary integers?

We need to understand about GATES and BOOLEAN ALGEBRA, that are foundations of logic design.
**AND gate**

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X.Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**OR gate**

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X+Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**NOT gate**

<table>
<thead>
<tr>
<th>X</th>
<th>\overline{X}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Typically, logical 1 = +3.5 volt, and logical 0 = 0 volt. Other representations are possible.
Analysis of logical circuits

What is the value of $F$ when $X=0$ and $Y=1$?

Draw a truth table.

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

This is the exclusive or (XOR) function. In algebraic form $F = \bar{X}Y + X\bar{Y}$
More practice

1. Let \( \bar{A}.B + A.C = 0 \). What are the values of \( A, B, C \)?
2. Let \( (A + B + C) \cdot (A + \bar{B} + C) = 0 \). What are the possible values of \( A, B, C \)?

• Draw truth tables.
• Draw the logic circuits for the above two functions.
Elementary Boolean Algebra

\[
\begin{align*}
A + \overline{A} &= 1 \\
A \cdot \overline{A} &= 0
\end{align*}
\]

\[
\begin{align*}
1 + A &= 1 \\
1 \cdot A &= A
\end{align*}
\]

\[
\begin{align*}
0 \cdot A &= 0 \\
0 + A &= A
\end{align*}
\]

\[
\begin{align*}
A + A &= A \\
A \cdot A &= A
\end{align*}
\]

\[
A \cdot (B + C) = A.B + A.C \quad \text{Distributive Law}
\]

\[
A + B.C = (A+B). (A+C)
\]

\[
\begin{align*}
\overline{A \cdot B} &= \overline{A} + \overline{B} \\
\overline{A + B} &= A \cdot B
\end{align*}
\]

\text{De Morgan's theorem}
**Synthesis of logic circuits**

Many problems of logic design can be specified using a truth table. Give such a table, can you design the logic circuit?

Design a logic circuit with three inputs $A$, $B$, $C$ and one output $F$ such that $F=1$ only when a majority of the inputs is equal to 1.

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Sum of product form**

$$F = \overline{A}.B.C + A\overline{B}.C + A.B.\overline{C} + A.B.C$$

Draw a logic circuit to generate $F$. 
