Computer Architecture

- We will work with a simulated tutorial architecture:
  - ISC: Incredibly Simple Computer
- ISC is an example of a
  - RISC (Reduced Instruction Set Computer), as opposed to a
  - CISC (Complex Instruction Set Computer)

Instructions

- *Machine language* is the language understood directly by a computer chip.
- A machine language program is a sequence of *instructions* stored in memory.
- Each instruction is a 32-bit word that tells the ISC to perform a single (simple) operation
  - Add the values in two registers
  - Get the value in a memory address
  - Start executing the program at a different memory address
  - etc.

Registers

- The ISC hardware contains 32 *general-purpose registers* that instructions can use for computations.
  - Each register can hold a 32-bit number
  - Registers are numbered 0-31
- Other registers exist, but are used implicitly
  - The most important is the instruction pointer (IP), which always contains the address of the next instruction to execute.
  - In some computers the instruction pointer is called the program counter (PC)
Machine-Language vs. Assembly Language

- Programming of the bare machine is typically done in assembly language
  - Really don't want to write a program as a list of 32-bit numbers...
- Languages correspond very directly
  - One line of assembly language is roughly equal to one machine instruction
  - However, assembly language is (more) human-readable
  - Translation from assembly to machine language is done by a program called an assembler.
  - The assembler may also permit the programmer to use abbreviations (particularly by giving names that represent registers or memory addresses)

Programming in Assembly Language

- Programming in assembly language reminds one of the Japanese saying about climbing Mt. Fuji:

  To never have climbed Mt. Fuji is to be a fool.
  Only a fool would climb Mt. Fuji more than once.

\[
\text{add Instruction} \quad \text{Example:} \quad \text{add 2 3 16} \\
\text{Effect:} \quad \text{reg[Ra] = reg[Rb] + reg[Rc]}
\]
(Bitwise) or Instruction

or Ra Rb Rc

Example: or 9 2 1
Effect: reg[Ra] = reg[Rb] | reg[Rc]

Other Register Instructions

- Many other arithmetic and logical instructions follow a similar pattern
  - sub Ra Rb Rc
  - mul Ra Rb Rc
  - div Ra Rb Rc
  - and Ra Rb Rc
  - shr Ra Rb Rc
  - shl Ra Rb Rc

- shift right
- shift left

Other Register Instructions

- Some register instructions only need two arguments...
  - comp Ra Rb
    - bitwise not
    - Effect: reg[Ra] = ~reg[Rb]

  - copy Ra Rb
    - Effect: reg[Ra] = reg[Rb]

Other Register Instructions

- ...or even just one register argument
  - lim Ra Constant
    - up to 24-bits
    - Example: lim 13 42
    - Effect: reg[Ra] = Constant

  - aim Ra Constant
    - Example: aim 13 42
    - Effect: reg[Ra] = reg[Ra]+Constant
Memory Access Instructions

- Memory acts like a big array (of 32-bit words)
  - Word-addressed, rather than byte-addressed

  \[ \text{load } Ra \ Rb \]

  \textbf{Effect: } reg[Ra] = mem[reg[Rb]]

  \[ \text{store } Ra \ Rb \]

  \textbf{Effect: } mem[reg[Ra]] = reg[Rb]

Jumps

- Recall that each instruction is stored as a 32-bit word in memory.
- Usually, we execute each instruction and then go on to the next word.
  - The IP always contains the address of the next instruction to execute
  - Normally, IP is 1 more than the address of the current instruction.
- Jumps (and, on other machines, branches) let us change the instruction pointer.

  \[ \text{Example: } \text{junc } 2 \]

  \textbf{Effect: } IP = reg[Ra]

Conditional Jumps

- Change the IP if two arguments are in a particular relation (less-than, equal, etc.)

  \[ \text{jeq } Ra \ Rb \ Rc \]

  \textbf{Effect: } if (reg[Rb]==reg[Rc])

  \[ \text{IP} = \text{reg[Ra]} \]

  \[ \text{jeq } Ra \ Rb \ Rc \quad \text{jne } Ra \ Rb \ Rc \quad \text{jgt } Ra \ Rb \ Rc \quad \text{jgte } Ra \ Rb \ Rc \quad \text{jlt } Ra \ Rb \ Rc \quad \text{jlte } Ra \ Rb \ Rc \]
Jump to Subroutine

\[ \text{jsub Ra Rb} \]

Effect:
\[
\begin{align*}
\text{reg}[Rb] &= \text{IP}; \\
\text{IP} &= \text{reg}[Ra]
\end{align*}
\]

Why? The subroutine can get back to the instruction following the jsub by jumping to the address in Rb!

Example (fragment)

```
... 
lim 2 0 
lim 3 0 
lim 6 20 
lim 5 14 
jlte 6 1 3 
load 4 0 
add 2 4 2 
aim 0 1 
aim 1 -1 
junc 5 
...
```

ISCAL
ISC Assembly Language

- Free-form input, but generally format line-by-line
- Regular instructions
  - \(Ra, Rb, Rc\) are register names
  - \(C\) s a constant
    - \(\text{lim } Ra \ C\)
    - \(\text{add } Ra \ Rb \ Rc\)
    - \(\text{copy } Ra \ Rb\)
    - \(\text{shl } Ra \ Rb\)
    - \(\text{load } Ra \ Rb\)
    - etc.

ISC Assembly Language

- **Assembler directives**: not instructions to computer, but rather tell assembler what to do:
  - `define Ident Value`: Defines `Ident` to abbreviate `Value`.
  - `register Ident Reg`: `Ident` can be used for register `Reg`.
  - `use Identifier`: Defines `Identifier` to name an unused register.
  - `origin Value`: Put next instruction at specified memory location. The location counter is incremented as loading progresses.
  - `label Identifier`: Associates current instruction location with `Identifier`.

ISCAL code for summing an array

```
use array use count // array base and count
use sum use zero use value // local registers
use loop use done // address registers

... insert code to load array and count values ...

lim sum 0 // initialize sum
lim zero 0 // comparison value
lim done done_loc // address of instruction following
lim loop loop_loc // address of next instruction

label loop_loc
jlte done count zero // jump if <= 0
load value array // load register next array value
add sum value sum // add the next number to the sum
aim array 1 // add 1 to the array address
aim count -1 // add -1 to the count
junc loop // go back and compare

label done_loc
```
Array Summation

In Memory

<table>
<thead>
<tr>
<th>General Registers</th>
<th>Array to be summed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>array</code></td>
<td>5 3 6 2 9</td>
</tr>
<tr>
<td><code>count</code></td>
<td>5</td>
</tr>
<tr>
<td><code>sum</code></td>
<td>5</td>
</tr>
<tr>
<td><code>zero</code></td>
<td>0</td>
</tr>
<tr>
<td><code>value</code></td>
<td>5</td>
</tr>
<tr>
<td><code>loop</code></td>
<td><code>IP</code></td>
</tr>
<tr>
<td><code>done</code></td>
<td><code>done_loc</code></td>
</tr>
</tbody>
</table>

Program code (symbolic, the actual code is a bit vector)

```
jlte done count zero
load value array
add sum value sum
aim array 1
aim count -1
junc loop
```

Showing register state after instruction is executed.

Array Summation

In Processor

<table>
<thead>
<tr>
<th>General Registers</th>
<th>Array to be summed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>array</code></td>
<td>5 3 6 2 9</td>
</tr>
<tr>
<td><code>count</code></td>
<td>5</td>
</tr>
<tr>
<td><code>sum</code></td>
<td>5</td>
</tr>
<tr>
<td><code>zero</code></td>
<td>0</td>
</tr>
<tr>
<td><code>value</code></td>
<td>5</td>
</tr>
<tr>
<td><code>loop</code></td>
<td><code>IP</code></td>
</tr>
<tr>
<td><code>done</code></td>
<td><code>done_loc</code></td>
</tr>
</tbody>
</table>

Showing register state after instruction is executed.
Array Summation

In Processor
- General Registers
  - array
  - count
  - sum
  - zero
  - value
  - loop
  - done

In Memory
- Array to be summed
- Program code (symbolic, the actual code is a bit vector)

Showing register state after instruction is executed.

Array Summation

In Processor
- General Registers
  - array
  - count
  - sum
  - zero
  - value
  - loop
  - done

In Memory
- Array to be summed
- Program code (symbolic, the actual code is a bit vector)

Showing register state after instruction is executed.
Array Summation

In Processor
- General Registers
  - array
  - count
  - sum
  - zero
  - value
  - loop
  - done

In Memory
- Array to be summed

Showing register state after instruction is executed.

Program code (symbolic, the actual code is a bit vector)

Array Summation

In Processor
- General Registers
  - array
  - count
  - sum
  - zero
  - value
  - loop
  - done

In Memory
- Array to be summed

Showing register state after instruction is executed.

Program code (symbolic, the actual code is a bit vector)
Array Summation

In Processor
- General Registers
  - array
  - count
  - sum
  - zero
  - value
  - loop
  - done

In Memory
- Array to be summed

Showing register state after instruction is executed.

Program code (symbolic, the actual code is a bit vector)

- loop_LOC: 5
- done_LOC: 3
- 5
- 6
- 2
- 9
- 0
- 14
- 3
- 14
- 0
- count
- sum
- done
- zero
- loop
- value
- done
- array
- load value array
- add sum value sum
- aim array 1
- aim count -1
- junc loop
Array Summation

In Processor

- General Registers
  - array
  - count
  - sum
  - zero
  - value
  - loop
  - done

In Memory

- Array to be summed
- Program code (symbolic, the actual code is a bit vector)

Showing register state after instruction is executed.

1. \( \text{array} \leftarrow \)
2. \( \text{count} \leftarrow 2 \)
3. \( \text{sum} \leftarrow 14 \)
4. \( \text{zero} \leftarrow 0 \)
5. \( \text{value} \leftarrow 2 \)
6. \( \text{loop} \leftarrow \text{IP} \)
7. \( \text{done} \leftarrow \text{done_loc} \)

Program code:

- \( \text{lile done count zero} \)
- \( \text{load value array} \)
- \( \text{add sum value sum} \)
- \( \text{add array value sum} \)
- \( \text{add count -1} \)
- \( \text{junc loop} \)

Showing register state after instruction is executed.

1. \( \text{array} \leftarrow \)
2. \( \text{count} \leftarrow 3 \)
3. \( \text{sum} \leftarrow 14 \)
4. \( \text{zero} \leftarrow 0 \)
5. \( \text{value} \leftarrow 2 \)
6. \( \text{loop} \leftarrow \text{IP} \)
7. \( \text{done} \leftarrow \text{done_loc} \)
Array Summation

In Processor

General Registers
- array
- count
- sum
- zero
- value
- loop
- done

In Memory

Array to be summed

Showing register state after instruction is executed.

Program code (symbolic, the actual code is a bit vector)

- jlte done count zero
- load value array
- add sum value sum
- aim array 1
- aim count -1
- junc loop

Showing register state after instruction is executed.
Array Summation

In Memory

Array to be summed

In Processor

General Registers

count
5
6
2
9

sum
5

zero
0

value
9

loop

done

loop_loc

done_loc

Program code (symbolic, the actual code is a bit vector)

jkte done count zero
load value array
add sum value sum
sin array 1
sin count -1
junc loop

Showing register state after instruction is executed.

Array Summation

In Memory

Array to be summed

In Processor

General Registers

count
5
6
2
9

sum
5

zero
0

value
9

loop

done

loop_loc

done_loc

Program code (symbolic, the actual code is a bit vector)

jkte done count zero
load value array
add sum value sum
sin array 1
sin count -1
junc loop

Showing register state after instruction is executed.
Array Summation

In Processor

General Registers

count
sum
zero
value
loop
done

In Memory

Array to be summed

Program code (symbolic, the actual code is a bit vector)

loop_loc
done_loc

The ISC Assembler

• The ISC assembler is actually a combined assembler, loader, and tracer (for debugging).
  - The executable is /cs/cs60/bin/isc
  - Sample programs are in /cs/cs60/isc/

• The array summation program is in /cs/cs60/isc/array.isc. It includes additional code to create the array and output the result.