**Computer Architecture**

- A computer is essentially a large collection of finite-state machines.
- A common clock is used for all.
- The machines intercommunicate by the output of one machine being the input to another.
- Combinational logic can be interposed between output and input to affect data transformations.

**The ISC**

- ISC is an example of a RISC (Reduced Instruction Set Computer), as opposed to a CISC (Complex Instruction Set Computer)

**ISC Design**

- We will work with a simulated tutorial architecture:

  ISC: Incredibly Simple Computer

**ISC Processor Components (1)**

- General Registers: Hold operands and results, and addresses for jumps in program.
- IR (Instruction Register) holds currently executing instruction.
- IP (Instruction Pointer): Holds address of next instruction.
- MDR (Memory Data Register) holds data en-route to/from memory.
- MAR (Memory Address Register) holds address of memory location for data.
ISC Processor Components (2)

- **ALU (Arithmetic-Logic Unit)**: Combinational unit performing addition, subtraction, logical operations, shifting, etc.
- **ALU registers**: Registers holding operands and results for ALU
- **Control sequencer**: Finite-state machine sequencing register and 3-state strobes, based upon the contents of the IR (Instruction Register)

### Instruction Dichotomies

- Instructions that access memory ("memory access" instructions)
- Instructions that include an operand in the instruction itself ("immediate" instructions)
- Instructions that change instruction location ("jump" instructions)

**Non-memory access instructions**

- **add Ra Rb Rc**: register indices (absolute or symbolic)
  
  \[ \text{reg[Ra]} = \text{reg[Rb]} + \text{reg[Rc]} \]

  Similarly for:
  
  - sub
  - mul
  - div
  - and
  - or
  - comp
  - shr
  - shl

**Memory access instructions**

- **load immediate**:  
  
  \[ \text{lim Ra C} \]

  \[ \text{Reg[Ra]} = C; \]

  \[ \text{aim Ra C} \]

  \[ \text{Reg[Ra]} \leftarrow C; \]

- **load**:  
  
  \[ \text{load Ra Rb} \]

  \[ \text{reg[Ra]} = \text{mem[reg[Rb]]}; \]

- **store**:  
  
  \[ \text{store Ra Rb} \]

  \[ \text{mem[reg[Ra]]} = \text{reg[Rb]}; \]

*The address to be used is in one of the general registers.*

C is a constant

It can be intended for arithmetic, logic, or as an address.
Jump instructions

Jump-if-equal:

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

Jump to address in Ra

if \( \text{reg[Rb]} == \text{reg[Rc]} \)

Similarly for:

\[ \text{jne jlt} \]
\[ \text{jgt jlte} \]
\[ \text{lgte} \]

Jump to address in Ra

The address to be jumped to is in one of the general registers.

Jump instructions

Jump-unconditionally

\[ \text{junc Ra} \]

Jump to address in Ra.

The address to be jumped to is in one of the general registers.

Jump instructions

Jump-to-subroutine

\[ \text{jsub Ra Rb} \]

Jump to address in Ra.

The next location after the current one is put in Rb.

Control Sequerencer FSM

- Inputs are bits from instruction being interpreted
- Outputs are strobes to various registers, 3-state devices, etc.

Control Sequence

- There is a sequence common to all instructions in which the instruction is fetched from memory, followed by
- A sequence particular to the type of instruction being executed.

ISC Instruction Fetch

1. Enable IP onto internal bus.
   Load MAR.

2. Read memory

3. Enable MDR onto internal bus.
   Load IR

4. Increment IP

Branch to control sub-sequence based upon IR contents.
Enable IP onto internal bus.
Load MAR.

Enable MDR onto internal bus.
Load IR.

Control Subsequence Example: Add Ra Rb Rc

1. Enable Rb onto bus
   Load ALU_in[0]

2. Enable Rc onto bus
   Load ALU_in[1]

3. Enable Add function of ALU

4. Enable ALU_out onto bus
   Load Ra.

Actual addition takes place in between.
Exercise

- What would the instruction subsequences be for:
  - aim (add immediate)
  - load
  - store
  - jeq (jump-if-equal)

Machine-Level Programming

- Programming of the bare machine is typically done in "assembly language"
- One line of assembly language is roughly equal to one machine instruction
- A program, the "assembler", allows use of symbolic identifiers for addresses.

Programming in Assembly Language

- Programming in assembly language reminds me of the Japanese saying about climbing Mt. Fuji:
  
  "To never have climbed Mt. Fuji is to be a fool. Only a (bigger) fool would climb Mt. Fuji more than once."
ISCAL
ISC Assembly Language

- See http://www.cs.hmc.edu/~keller/iscal/
- Free-form input, but generally format line-by-line
- Regular instructions
  - Rs, Rb, Rc are register names
  - C is a constant
    - lim Rs C
    - add Rs Rb Rs
    - copy Rs Rb
    - shi Rs Rb
    - load Rs Rb
    - etc.

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 counter 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
add sum value sum
aim array 1
aim count -1
junc loop
label done_loc
```

Array Summation

- Assembler directives: not instructions to computer, but rather tell assembler what to do:
  - define Identifier Value: Defines Identifier to have Value.
  - use Identifier: Defines Identifier to name an unused register.
  - origin Value: Begin loading instructions at specified memory location. The location counter is incremented as loading progresses.
  - label Identifier: Associates current instruction location with Identifier.

Array Summation

```
In Processor In Memory
General Registers Array to be summed

array
count
sum
zero
value
loop
done

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

Array Summation

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)

1. `jlte done count zero`
2. `load value array`
3. `add sum value sum`
4. `aim array 1`
5. `aim count -1`
6. `junc loop`

#### Showing register state after instruction is executed.

<table>
<thead>
<tr>
<th>IP</th>
<th>array</th>
<th>count</th>
<th>sum</th>
<th>zero</th>
<th>value</th>
<th>loop</th>
<th>done</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>6</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>6</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>6</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

### 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)

1. `jlte done count zero`
2. `load value array`
3. `add sum value sum`
4. `aim array 1`
5. `aim count -1`
6. `junc loop`

#### Showing register state after instruction is executed.

<table>
<thead>
<tr>
<th>IP</th>
<th>array</th>
<th>count</th>
<th>sum</th>
<th>zero</th>
<th>value</th>
<th>loop</th>
<th>done</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>6</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>6</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>6</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Array Summation

In Processor In Memory

General Registers

Array to be summed

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

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be summed

Array to be sum
Array Summation

In Processor | In Memory
--- | ---
**General Registers** | Array to be summed
array | 
count | 
save | 
zero | 
value | 
loop | 
done | 

Showing register state after instruction is executed.

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

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

loop_loc
done_loc

5
3
6
2
9
2
14
0
count
sum
done
zero
loop
value
6

Showing register state after instruction is executed.

IP array
### 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)**
```
jlte done count zero
load value array
add sum value
aim array 1
aim count -1
junc loop
```

**Showing register state after instruction is executed.**

**Register state after instruction is executed:**
- IP: 5
- count: 3
- sum: 6
- zero: 2
- value: 9
- loop: 0
- done: 0

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

**Showing register state after instruction is executed.**

**Register state after instruction is executed:**
- IP: 5
- count: 3
- sum: 6
- zero: 2
- value: 9
- loop: 0
- done: 0

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

**Showing register state after instruction is executed.**

**Register state after instruction is executed:**
- IP: 5
- count: 3
- sum: 6
- zero: 2
- value: 9
- loop: 0
- done: 0

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

**Showing register state after instruction is executed.**

**Register state after instruction is executed:**
- IP: 5
- count: 3
- sum: 6
- zero: 2
- value: 9
- loop: 0
- done: 0

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

**Showing register state after instruction is executed.**

**Register state after instruction is executed:**
- IP: 5
- count: 3
- sum: 6
- zero: 2
- value: 9
- loop: 0
- done: 0
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 ISC Assembler

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

ISC output

```
turing iscc1> lcc array.isc
 25
```
```
turing iscc2> lcc -t array.isc
Begin trace
line:  36 loc:   0 contents: lim   r9 (0)   0
      reg[9] = 0
... lots of trace output ... 
------ output ------
25
```
```
------- output -------
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Example: Recursive Factorial

```plaintext
label fac // recursive factorial routine
lim result 1 // basis is 1
ylte return arg zero // return if count is 0 or less
aim stack_pointer +1 // increment stack pointer
store stack_pointer return // save return address on stack
lim stack_pointer +1 // increment stack pointer
store stack_pointer arg // save argument on stack
aim arg -1 // subtract 1 from argument
push jump_target return // call recursively
load arg stack_pointer // restore original argument
aim stack_pointer +1 // increment stack pointer
load stack_pointer arg // save
push // save argument on stack
aim stack_pointer -1 // restore original argument
load arg stack_pointer // restore original argument
aim stack_pointer -1 // restore original return address
mul result arg // multiply by original arg
junc return // return to caller
```