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.

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).

**ISC Design**

- 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)

**ISC Processor Structure**

**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):\n    - \( \text{reg[Ra]} = \text{reg[Rb]} + \text{reg[Rc]} \)
  - Similarly for: sub, mul, div, and, or, comp, shr, shl

**"Immediate" instructions**

- **load immediate**: \( \text{lim Ra C} \)
  - \( \text{Reg[Ra]} = C \)
- **add immediate**: \( \text{aim Ra C} \)
  - \( \text{Reg[Ra]} += C \)

**Memory access instructions**

- **load**: \( \text{load Ra Rb} \)
  - \( \text{reg[Ra]} = \text{mem[reg[Rb]]} \)
- **store**: \( \text{store Ra Rb} \)
  - \( \text{mem[reg[Ra]]} = \text{reg[Rb]} \)

**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:
\[ \text{jne} \quad \text{jlt} \quad \text{jgt} \quad \text{jite} \quad \text{jgte} \]

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.

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

Control Sequencer 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 MMR.
2. Read memory
3. Enable MDR onto internal bus.
   Load IR
   Increment IP
   Branch to control sub-sequence based upon IR contents.
Enable IP onto internal bus.
Load MAR.

Read memory

Enable MDR onto internal bus.
Load IR

Increment IP

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.
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.

Exercise

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

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 fool would climb Mt. Fuji more than once."
ISCAL
ISC Assembly Language

- See http://www.cs.hmc.edu/~keller/isc/
- Free-form input, but generally format line-by-line
- Regular instructions
  - Rs, Rb, Rc are register names
  - C is a constant
    - add Rs Rb Rc
    - copy Rs Rb
    - shl 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 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 Processor**

<table>
<thead>
<tr>
<th>General Registers</th>
<th>In Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td></td>
</tr>
<tr>
<td>zero</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td></td>
</tr>
</tbody>
</table>

**In Memory**

```
array          //          
count          //          
sum            //          
zero           //          
value          //          
loop           //          
done           //          

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

 Showing register state after instruction is executed.
Array Summation

In Processor | In Memory
---|---
**General Registers** | **Array to be summed**
array | 
count | 
sum | 
para | 
value | 
lap | 
da| 

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
```

loop_loc
done_loc

count
sum
done
zero
loop
value

Showing register state after instruction is executed.

IP
Array Summation

In Memory
General Registers
Array to be summed

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

loop_loc
done_loc

In Processor
General Registers
Array to be summed

count
sum
c
value
loop
done

5
3
6
2
9

4
8
0
count
sum
done
zero
loop
value

0x0

Showing register state after instruction is executed.

IP
array

Array Summation

In Memory
General Registers
Array to be summed

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

loop_loc
done_loc

In Processor
General Registers
Array to be summed

count
sum
c
value
loop
done

5
3
6
2
9

4
8
0
count
sum
done
zero
loop
value

0x0

Showing register state after instruction is executed.

IP
array

Array Summation

In Memory
General Registers
Array to be summed

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

loop_loc
done_loc

In Processor
General Registers
Array to be summed

count
sum
c
value
loop
done

5
3
6
2
9

4
8
0
count
sum
done
zero
loop
value

0x0

Showing register state after instruction is executed.

IP
array

Array Summation

In Memory
General Registers
Array to be summed

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

loop_loc
done_loc

In Processor
General Registers
Array to be summed

count
sum
c
value
loop
done

5
3
6
2
9

4
8
0
count
sum
done
zero
loop
value

0x0

Showing register state after instruction is executed.

IP
array

Array Summation

In Memory
General Registers
Array to be summed

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

loop_loc
done_loc

In Processor
General Registers
Array to be summed

count
sum
c
value
loop
done

5
3
6
2
9

4
8
0
count
sum
done
zero
loop
value

0x0

Showing register state after instruction is executed.

IP
array

Array Summation

In Memory
General Registers
Array to be summed

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

loop_loc
done_loc

In Processor
General Registers
Array to be summed

count
sum
c
value
loop
done

5
3
6
2
9

4
8
0
count
sum
done
zero
loop
value

0x0

Showing register state after instruction is executed.

IP
array

Array Summation

In Memory
General Registers
Array to be summed

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

loop_loc
done_loc

In Processor
General Registers
Array to be summed

count
sum
c
value
loop
done

5
3
6
2
9

4
8
0
count
sum
done
zero
loop
value

0x0

Showing register state after instruction is executed.

IP
array

Array Summation

In Memory
General Registers
Array to be summed

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

loop_loc
done_loc

In Processor
General Registers
Array to be summed

count
sum
c
value
loop
done

5
3
6
2
9

4
8
0
count
sum
done
zero
loop
value

0x0

Showing register state after instruction is executed.

IP
array
Array Summation

In Memory

Array to be summed

General Registers

count

cum

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

loop_loc

done_loc

count

sum

done

zero

loop

value

0x0

Showing register state after instruction is executed.

IP

array

showing register state after instruction is executed.

5

3

6

2

9

3

14

0

count

sum

done

zero

loop

value

0x0

IP

array
Array Summation

In Processor

General Registers

Array to be summed

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

loop_loc
done_loc

5
3
6
2
9
16
0
count
sum
done
zero
loop
count

In Memory

Showing register state after instruction is executed.

value

IP

array

done

value

IP

array

done

value

IP

array

done

value

IP

array

done

value

IP

array

done

value

IP

array

done

value

IP

array

done

value

2
2
0x0

9
1

Array Summation

In Processor

<table>
<thead>
<tr>
<th>General Registers</th>
<th>Array to be summed</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td></td>
</tr>
<tr>
<td>para</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td>done_pc</td>
</tr>
</tbody>
</table>

In Memory

<table>
<thead>
<tr>
<th>Program code (symbolic, the actual code is a bit vector)</th>
</tr>
</thead>
<tbody>
<tr>
<td>jlte done count zero</td>
</tr>
<tr>
<td>load value array</td>
</tr>
<tr>
<td>add sum value sum</td>
</tr>
<tr>
<td>aim array 1</td>
</tr>
<tr>
<td>aim count -1</td>
</tr>
<tr>
<td>junc loop</td>
</tr>
</tbody>
</table>

loop_loc

done_loc

count

sum

done

value

loop

done_pc

Array Summation

In Processor

<table>
<thead>
<tr>
<th>General Registers</th>
<th>Array to be summed</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td></td>
</tr>
<tr>
<td>para</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td>done_pc</td>
</tr>
</tbody>
</table>

In Memory

<table>
<thead>
<tr>
<th>Program code (symbolic, the actual code is a bit vector)</th>
</tr>
</thead>
<tbody>
<tr>
<td>jlte done count zero</td>
</tr>
<tr>
<td>load value array</td>
</tr>
<tr>
<td>add sum value sum</td>
</tr>
<tr>
<td>aim array 1</td>
</tr>
<tr>
<td>aim count -1</td>
</tr>
<tr>
<td>junc loop</td>
</tr>
</tbody>
</table>

loop_loc

done_loc

count

sum

done

value

loop

done_pc

Array Summation

In Processor

<table>
<thead>
<tr>
<th>General Registers</th>
<th>Array to be summed</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td></td>
</tr>
<tr>
<td>para</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td>done_pc</td>
</tr>
</tbody>
</table>

In Memory

<table>
<thead>
<tr>
<th>Program code (symbolic, the actual code is a bit vector)</th>
</tr>
</thead>
<tbody>
<tr>
<td>jlte done count zero</td>
</tr>
<tr>
<td>load value array</td>
</tr>
<tr>
<td>add sum value sum</td>
</tr>
<tr>
<td>aim array 1</td>
</tr>
<tr>
<td>aim count -1</td>
</tr>
<tr>
<td>junc loop</td>
</tr>
</tbody>
</table>

loop_loc

done_loc

count

sum

done

value

loop

done_pc

Array Summation

In Processor

<table>
<thead>
<tr>
<th>General Registers</th>
<th>Array to be summed</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td></td>
</tr>
<tr>
<td>para</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td>done_pc</td>
</tr>
</tbody>
</table>

In Memory

<table>
<thead>
<tr>
<th>Program code (symbolic, the actual code is a bit vector)</th>
</tr>
</thead>
<tbody>
<tr>
<td>jlte done count zero</td>
</tr>
<tr>
<td>load value array</td>
</tr>
<tr>
<td>add sum value sum</td>
</tr>
<tr>
<td>aim array 1</td>
</tr>
<tr>
<td>aim count -1</td>
</tr>
<tr>
<td>junc loop</td>
</tr>
</tbody>
</table>

loop_loc

done_loc

count

sum

done

value

loop

done_pc

Array Summation

In Processor

<table>
<thead>
<tr>
<th>General Registers</th>
<th>Array to be summed</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td></td>
</tr>
<tr>
<td>para</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td>done_pc</td>
</tr>
</tbody>
</table>

In Memory

<table>
<thead>
<tr>
<th>Program code (symbolic, the actual code is a bit vector)</th>
</tr>
</thead>
<tbody>
<tr>
<td>jlte done count zero</td>
</tr>
<tr>
<td>load value array</td>
</tr>
<tr>
<td>add sum value sum</td>
</tr>
<tr>
<td>aim array 1</td>
</tr>
<tr>
<td>aim count -1</td>
</tr>
<tr>
<td>junc loop</td>
</tr>
</tbody>
</table>

loop_loc

done_loc

count

sum

done

value

loop

done_pc

Array Summation

In Processor

<table>
<thead>
<tr>
<th>General Registers</th>
<th>Array to be summed</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td></td>
</tr>
<tr>
<td>para</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td>done_pc</td>
</tr>
</tbody>
</table>

In Memory

<table>
<thead>
<tr>
<th>Program code (symbolic, the actual code is a bit vector)</th>
</tr>
</thead>
<tbody>
<tr>
<td>jlte done count zero</td>
</tr>
<tr>
<td>load value array</td>
</tr>
<tr>
<td>add sum value sum</td>
</tr>
<tr>
<td>aim array 1</td>
</tr>
<tr>
<td>aim count -1</td>
</tr>
<tr>
<td>junc loop</td>
</tr>
</tbody>
</table>

loop_loc

done_loc

count

sum

done

value

loop

done_pc

Array Summation

In Processor

<table>
<thead>
<tr>
<th>General Registers</th>
<th>Array to be summed</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td></td>
</tr>
<tr>
<td>para</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td>done_pc</td>
</tr>
</tbody>
</table>

In Memory

<table>
<thead>
<tr>
<th>Program code (symbolic, the actual code is a bit vector)</th>
</tr>
</thead>
<tbody>
<tr>
<td>jlte done count zero</td>
</tr>
<tr>
<td>load value array</td>
</tr>
<tr>
<td>add sum value sum</td>
</tr>
<tr>
<td>aim array 1</td>
</tr>
<tr>
<td>aim count -1</td>
</tr>
<tr>
<td>junc loop</td>
</tr>
</tbody>
</table>

loop_loc

done_loc

count

sum

done

value

loop

done_pc

Array Summation

In Processor

<table>
<thead>
<tr>
<th>General Registers</th>
<th>Array to be summed</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td></td>
</tr>
<tr>
<td>para</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td>done_pc</td>
</tr>
</tbody>
</table>

In Memory

<table>
<thead>
<tr>
<th>Program code (symbolic, the actual code is a bit vector)</th>
</tr>
</thead>
<tbody>
<tr>
<td>jlte done count zero</td>
</tr>
<tr>
<td>load value array</td>
</tr>
<tr>
<td>add sum value sum</td>
</tr>
<tr>
<td>aim array 1</td>
</tr>
<tr>
<td>aim count -1</td>
</tr>
<tr>
<td>junc loop</td>
</tr>
</tbody>
</table>

loop_loc

done_loc

count

sum

done

value

loop

done_pc
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/```

isc output

turing isc:1> isc array.isc
25

turing isc:2> isc -t array.isc
Begin trace
line: 36 loc: 0 contents: lim r9 (0) 0
reg[9] = 0
... lots of trace output ...
------- output --------
25
------------ --------
line: 88 loc: 32 contents: junc r9 (0)
jumping unconditionally to 0

Implementing Procedures

- A procedure is just some fixed code we jump to to perform some computation.
- We put the arguments to the procedure into pre-agreed standard registers, and get the result from another register.
- The jsib instruction can be used to get the return address.

Implementing Recursion

- Recursion presents a problem: The fixed code would tend to clobber (an inelegant way of saying “over-write”) the return address when the procedure calls itself.
- Also, the arguments in registered would get clobbered.

Stacks to the Rescue

- To deal with the clobbering problem, we can save the return address and any arguments on a stack allocated at a standard place in memory.
- The stack is typically an array, with a pointer to the top element.

```
lim stack_pointer save_area_loc // initialize stack pointer
aim stack_pointer -1 // always point to top of stack
```
Example: Recursive Factorial

```
label fac                      // recursive factorial routine
  lde result 1               // basis is 1
  jne 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
  push
  aim stack_pointer +1      // increment stack pointer
  store stack_pointer arg   // save argument on stack
  aim arg -1                // subtract 1 from argument
  sub jump_target return    // call recursively
  pop
  load arg stack_pointer    // restore original argument
  aim stack_pointer -1      // return
  pop
  load return stack_pointer // restore original return address
  aim stack_pointer -1      // return
  mul result result arg     // multiply by original arg
  jun return                // return to caller
```

Stack trace of factorial(4)