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 places, and get the result from another register.
- The jsub instruction can be used to make sure we can get back, by storing the return address in a pre-agreed register.

Review: ISCAL array summation

```
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 registers ...

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
```

First Try

```
use inc    // to contain address of inc routine
use ret    // to contain return address
use a    // register to be incremented

li inc inc_loc
li ret ret_loc

li a 1
junc inc

label ret_loc
    // continue on with the register a containing 2
    // ...

label inc
    aim a 1
    junc inc
```
Better

use inc  // to contain address of inc routine
use ret  // to contain return address
use a   // register to be incremented

li inc inc_loc
li a 1
jsub inc ret
// continue on with the register a containing 2
// ...

label inc
aim a 1
junc ret

Does This Work?

use inc  // to contain address of inc routine
use inc2 // to contain address of inc2 routine
use ret  // to contain return address
use a   // register to be incremented

li inc2 inc2_loc
li inc inc_loc
li a 1
jsub inc2 ret
// continue on with the register a containing 3
// ...

label inc_loc
aim a 1
junc ret
label inc2_loc
jsub inc ret
jsub inc ret
junc ret

Implementing Recursion

• Recursion presents a problem: The fixed code would tend to clobber ("over-write") the return address when the procedure calls itself.

• Also, the arguments in registers 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.
  – That is, we put important register values in memory while we call other functions

• 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

```plaintext
label fac // recursive factorial routine
lin result 1 // basis is 1
jite return arg zero // return if count is 0 or less
push
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
jsub jump_target return // call recursively
load arg stack_pointer // restore original argument
pop
load return stack_pointer // restore original return address
aim stack_pointer -1
mul result result arg // multiply by original arg
junc return // return to caller
```

Stack trace of \texttt{fac(4)}

- **Fourth Call:**
  - ret addr. 4
  - \texttt{fac r.a.} 3
  - \texttt{fac r.a.} 2

- **Fifth Call:**
  - ret addr. 4
  - \texttt{fac r.a.} 3
  - \texttt{fac r.a.} 2

Stack trace of \texttt{factorial(4)}

- **Second Call:**
  - ret addr. 4
  - arg contains 3

- **Third Call:**
  - ret addr. 4
  - \texttt{fac r.a.} 3
  - stack_pointer 2

Stack trace of \texttt{fac(4)}

- **Second Call:**
  - ret addr. 4
  - arg contains 3
  - return contains address after jsub

- **Third Call:**
  - ret addr. 4
  - arg contains 2
  - return contains address after jsub

- **Fourth Call:**
  - ret addr. 4
  - \texttt{fac r.a.} 3
  - \texttt{fac r.a.} 2

- **Fifth Call:**
  - ret addr. 4
  - arg contains 0
  - return contains address after jsub
Machine Language

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lim</td>
<td>000</td>
<td>register constant</td>
</tr>
<tr>
<td>aim</td>
<td>001</td>
<td>register constant</td>
</tr>
<tr>
<td>load</td>
<td>10000000</td>
<td>register register unused</td>
</tr>
<tr>
<td>store</td>
<td>10000001</td>
<td>register register unused</td>
</tr>
<tr>
<td>copy</td>
<td>10000010</td>
<td>register register unused</td>
</tr>
<tr>
<td>add</td>
<td>10000100</td>
<td>register register register</td>
</tr>
<tr>
<td>sub</td>
<td>10000101</td>
<td>register register register</td>
</tr>
</tbody>
</table>

Machine Language

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jeq</td>
<td>10010010</td>
<td>register register register</td>
</tr>
<tr>
<td>jne</td>
<td>10011101</td>
<td>register register register</td>
</tr>
<tr>
<td>junc</td>
<td>10010111</td>
<td>register unused</td>
</tr>
<tr>
<td>jsop</td>
<td>10011101</td>
<td>register register unused</td>
</tr>
</tbody>
</table>

what instruction is 2239303603 ?
10000101011110010000111110110011

ISC Design

A big array

- **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)

Controlling the Components

- Each register has a strobe controlling it (not shown in the picture):
  - Write current value to the bus
  - Read current value from the bus
  - Unconnected to the bus

- Memory also has control bits:
  - Get the address from the MAR and put the value in the MDR at that address
  - Get the address from the MAR and put its current value into the MDR
  - Do nothing

- The ALU unit also has control bits saying what operation it should be performing (add, or, ...)

A Finite-State Controller

- We get a (programmable) computer by having a finite state machine that turns the strobes/control bits on and off as needed!

- For each state, we say which control bits should be on and which should be off
  - Technically, therefore, this FSM is a classifier with many bits of output determined by the state.
  - Move from one state to the next every clock cycle
  - Next state determined by the contents of some of the registers.
General Control Sequence

- From the start state, begin with a common sequence of states
  - Effect: fetch an instruction from memory and put it into the Instruction Register (IR)
- Then, a transition which depends on the type of instruction being executed.
  - Instruction decoding
- The next sequence of states actually executes the particular instruction.
  - Do the add, or the jsub, etc.
- Then we go back to the start state.
Enable MDR onto internal bus.
Load IR

3

Increment IP

3'

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.
Enable Rc onto bus
Load ALU_in[1]

Enable ALU_out onto bus
Load Ra

Enable Add function of ALU

Exercise

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