Register Machines

- Connecting evaluators to low level machine code

Plan

- Design a central processing unit (CPU) from:
  - wires
  - logic (networks of AND gates, OR gates, etc)
  - registers
  - control sequencer
- Our CPU will interpret Scheme as its machine language
- Today: Iterative algorithms in hardware
- Recursive algorithms in hardware
- Then: Scheme in hardware (EC-EVAL)
  - EC-EVAL exposes more details of scheme than M-EVAL

The ultimate goal

A universal machine

- Existence of a universal machine has major implications for what "computation" means
- Insight due to Alan Turing (1912-1954)
  - Hilbert's Entscheidungsproblem (decision problem) 1900:
    - Is mathematics decidable? That is, is there a definite method guaranteed to produce a correct decision about all assertions in mathematics?
- Church-Turing thesis:
  - Any procedure that could reasonably be considered to be an effective procedure can be carried out by a universal machine (and thus by any universal machine)

Euclid's algorithm to compute GCD

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

- Given some numbers a and b
- If b is 0, done (the answer is a)
- If b is not 0:
  - the new value of a is the old value of b
  - the new value of b is the remainder of a + b
  - start again

Example register machine: datapaths

```
register
operation
button
constant
wire
```

```
a b rem t
```

```
0
```

test

```
=
```
Example register machine: instructions

```
(controller
test-b
  (test (op =) (reg b) (const 0))
  (branch (label gcd-done))
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t))
  (goto (label test-b))
gcd-done)
```

Complete register machine

Datapath components

- **Button**
  - When pressed, value on input wire flows to output
- **Register**
  - Output the stored value continuously
  - Change value when button on input wire is pressed
- **Operation**
  - Output wire value = some function of input wire values
- **Test**
  - An operation
  - Output is one bit (true or false)
  - Output wire goes to condition register

Euclid’s algorithm to compute GCD

```
(define (gcd a b)
  (if (= b 0)
    a
    (gcd b (remainder a b)))))
```

Datapath for GCD (partial)

- What sequence of button presses will result in:
  - The register `a` containing GCD(a,b)
  - The register `b` containing 0
- The operation `rem` computes the remainder of `a ÷ b`
Example register machine: instructions

(controller
test-b
(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
goto (label test-b))
gcd-done)

Instructions
- Controller: generates a sequence of button presses
  - sequencer
  - instructions
- Sequencer: activates instructions sequentially
  - program counter remembers which one is next
- Each instruction:
  - commands a button press, OR
  - changes the program counter
  - called a branch instruction

Button-press instructions: the sum example

X
Y
sum
+ 0

(assign sum (const 0)) <X>
(assign sum (op +) (reg sum) (const 1)) <Y>
(assign sum (op +) (reg sum) (const 1)))

Unconditional branch

nextPC <- PC + 1
activate instruction at PC
PC <- nextPC
start again

<table>
<thead>
<tr>
<th>PC</th>
<th>nextPC</th>
<th>press</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>A1</td>
<td>--</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>A1</td>
<td>--</td>
</tr>
</tbody>
</table>

(assign sum (const 0))
(assign sum (op +) (reg sum) (const 1))
goto (label increment))

Conditional branch

(test (op =) (reg b) (const 0))
(branch (label gcd-done))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
goto (label test-b))
gcd-done)

Conditional branch details
(test (op =) (reg b) (const 0))
  • push the button which loads the condition register from this operation’s output
(branch (label gcd-done))
  • Overwrite nextPC register with value if condition register is TRUE
  • No effect if condition register is FALSE
Datapaths are redundant
• We can always draw the data path required for an instruction sequence
• Therefore, we can leave out the data path when describing a register machine

Abstract operations
• Every operation shown so far is abstract:
  • abstract = consists of multiple lower-level operations
• Lower-level operations might be:
  • AND gates, OR gates, etc (hardware building-blocks)
  • sequences of register machine instructions
• Example: GCD machine uses
  \[(\text{assign } t \ (\text{op rem}) \ (\text{reg a}) \ (\text{reg b}))\]
  • Rewrite this using lower-level operations

Less-abstract GCD machine
\[
\begin{align*}
\text{controller} \\
\text{test-b} \\
& (\text{assign } t \ (\text{op rem}) \ (\text{reg a}) \ (\text{reg b})) \\
& (\text{assign } t \ (\text{reg a})) \\
\text{rem-loop} \\
& (\text{assign } t \ (\text{op <}) \ (\text{reg t}) \ (\text{reg b})) \\
& (\text{assign } t \ (\text{op -}) \ (\text{reg t}) \ (\text{reg b})) \\
& (\text{goto} \ (\text{label rem-loop})) \\
\text{rem-done} \\
& (\text{assign } a \ (\text{reg b})) \\
& (\text{assign } b \ (\text{reg t})) \\
& (\text{goto} \ (\text{label test-b})) \\
\text{gcd-done}
\end{align*}
\]

Importance of register machine abstraction
• A CPU is a very complicated device
• We will study only the core of the CPU
  • eval, apply, etc.
• We will use abstract register-machine operations for all the other instruction sequences and circuits:
  \[(\text{test} \ (\text{op self-evaluating?}) \ (\text{reg exp}))\]
  • remember, (op +) is abstract, (op <) is abstract, etc.
  • no magic in (op self-evaluating?)

Review of register machines
• Registers hold data values
• Controller specifies sequence of instructions, order of execution controlled by program counter
  • Assign puts value into register
    • Constants
    • Contents of register
    • Result of primitive operation
  • Goto changes value of program counter, and jumps to label
  • Test examines value of a condition, setting a flag
  • Branch resets program counter to new value, if flag is true
• Data paths are redundant

Machines for recursive algorithms
• GCD, odd?, increment
  • iterative, constant space
• factorial, EC-EVAL
  • recursive, non-constant space
• Extend register machines with subroutines and stack
• Main points
  • Every subroutine has a contract
  • Stacks are THE implementation mechanism for recursive algorithms
Part 1: Subroutines

• **Subroutine**: a sequence of instructions that
  • starts with a label and ends with an indirect branch
  • can be called from multiple places

• New register machine instructions
  • `(assign continue (label after-call-1))`
    – store the instruction number corresponding to label
      `after-call-1` in register `continue`
    – this instruction number is called the *return point*
  • `(goto (reg continue))`
    – an indirect branch
    – change the PC to the value stored in register `continue`

Example subroutine: increment

```scheme
(assign (reg sum) (const 0))
(assign continue (label after-call-1))
goto (label increment)
(assign continue (label after-call-2))
goto (label done)
```

• `increment`
  • input: `sum, continue`
  • output: `sum`
  • writes: none

Subroutines have contracts

• Follow the contract or register machine will fail:
  • registers containing input values and return point
  • registers in which output is produced
  • registers that will be overwritten
    – in addition to the output registers

• subroutine `increment`
  • input: `sum, continue`
  • output: `sum`
  • writes: none

End of part 1

• Why subroutines?
  • reuse instructions
  • reuse data path components
  • make instruction sequence more readable
    – just like using helper functions in scheme
  • support recursion

• Contracts
  • specify inputs, outputs, and registers used by subroutine

Part 2: Stacks

• **Stack**: a memory device
  • `save` a register:
    • send its value to the stack
  • `restore` a register:
    • get a value from the stack

When this machine halts, `b` contains 0:

```scheme
(assign a (const 0))
(assign b (const 5))
save a
save b
restore a
restore b
```

Stacks: hold many values, last-in first-out

• This machine halts with 5 in `a` and 0 in `b`
  ```scheme
  (controller
    (assign a (const 0))
    (assign b (const 5))
    (save a)
    (save b)
    (restore a)
    (restore b)
  )
  ```

  contents of stack
  after step

<table>
<thead>
<tr>
<th>step</th>
<th>a</th>
<th>b</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>5</td>
<td>empty</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

• 5 is the top of stack after step 3
  •`save`: put a new value on top of the stack
  •`restore`: remove the value at top of stack
Check your understanding

- Draw the stack after step 5. What is the top of stack value?
- Add restores so final state is a: 3, b: 5, c: 8, and stack is empty

{(controller
  0 (assign a (const 8))
  1 (assign b (const 3))
  2 (assign c (const 5))
  3 (save b)
  4 (save c)
  5 (save a)
)

Things to know about stacks

- stack depth
- stacks and subroutine contracts
- tail-call optimization

Stack depth

- depth of the stack = number of values it contains
- At any point while the machine is executing
  - stack depth = (total # of saves) - (total # of restores)
- stack depth limits:
  - low: 0 (machine fails if restore when stack empty)
  - high: amount of memory available
- max stack depth:
  - measures the space required by an algorithm

Stacks and subroutine contracts

- Standard contract: subroutine increment
  - input: sum, continue
  - output: sum
  - writes: none
  - stack: unchanged
- Rare contract: strange
  (assign val (op *) (reg val) (const 2))
  (restore continue)
  (goto (reg continue))
  - input: val, return point on top of stack
  - output: val
  - writes: continue
  - stack: top element removed

Optimizing tail calls

no work after call except (goto (reg continue))

<table>
<thead>
<tr>
<th>Setup</th>
<th>Unoptimized version</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup (assign sum (const 15)) (save continue) (assign continue (label after-call)) (goto (label increment)) after-call (restore continue) (goto (reg continue))</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Setup</th>
<th>Optimized version</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup (assign sum (const 15)) (goto (label increment))</td>
<td></td>
</tr>
</tbody>
</table>

This optimization is important in EC-EVAL
- iterative algorithms expressed as recursive procedures would use non-constant space without it

End of part 2

- stack
  - a LIFO memory device
  - save: put data on top of the stack
  - restore: remove data from top of the stack
- things to know
  - concept of stack depth
  - expectations and effect on stack is part of the contract
  - tail call optimization
Part 3: recursion

(define (fact n)
  (if (= n 1) 1
    (* n (fact (- n 1)))))

(fact 3)
(* 3 (* 2 (fact 1)))
(* 3 2)
6

• The stack is the key mechanism for recursion
  • remembers return point of each recursive call
  • remembers intermediate values (e.g., n)

Code: base case

(define (fact n)
  (if (= n 1) 1
    ...))

fact    (test (op =) (reg n) (const 1))
(branch (label b-case))
...

b-case    (assign val (const 1))
(goto (reg continue))

• fact expects its input in which register? n
• fact expects its return point in which register? continue
• fact produces its output in which register? val

Code: recursive call

(define (fact n)
  ...
    (fact (- n 1))
    ...)

(assign n (op -) (reg n) (const 1))
(assign continue (label r-done))
(goto (label fact))

r-done
(restore n)
(restore continue)
(assign val (op *) (reg n) (reg val))
(goto (reg continue))

b-case
(assign val (const 1))
(goto (reg continue))
halt

• A t r-done, which register will contain the return value of the recursive call?
  val

Code: after recursive call

(define (fact n)
  ...
    (* n <return-value> )
    ...)

(assign val (op *) (reg n) (reg val))
(goto (reg continue))

• Problem!
  • Overwrote register n as part of recursive call
  • Also overwrote continue

Code: complete recursive case

(save continue)
(save n)
(assign n (op -) (reg n) (const 1))
(assign continue (label r-done))
(goto (label fact))

r-done
(restore n)
(restore continue)
(assign val (op *) (reg n) (reg val))
(goto (reg continue))

• Save a register if:
  • value is used after call AND
  • register is not output of subroutine AND
  • (register written as part of call OR
    register written by subroutine)
Check your understanding
• Write down the contract for subroutine fact
  • input:
  • output:
  • writes:
  • stack:

Execution trace
• Contents of registers and stack at each label
• Top of stack at left
  label  continue n  val  stack
  fact  halt 3  ???  empty
  fact  r-done 2  ???  3 halt
  fact  r-done 1  ???  2 r-done 3 halt
  b-case  r-done 1  ???  2 r-done 3 halt
  r-done  r-done 1  1  ??  2 r-done 3 halt
  r-done  r-done 2  2  3 halt
  halt  halt 3  6  empty

• Contents of stack represents pending operations
  (* 3 (* 2 (fact 1))) at base case

End of part 3
• To implement recursion, use a stack
  • stack records pending work and return points
  • max stack depth = space required
  – (for most algorithms)

Where we are headed
• Next time will use register machine idea to implement an evaluator
  • This will allow us to capture high level abstractions of Scheme while connecting to low level machine architecture