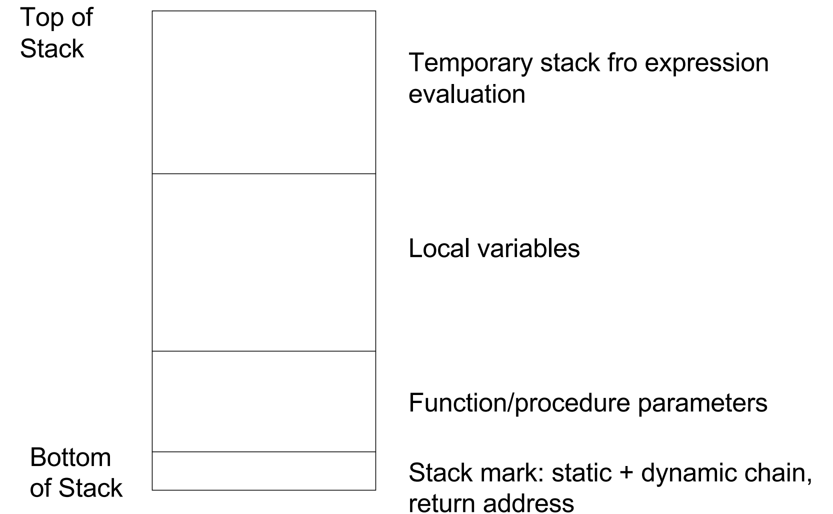
RPAL – Robot machine language

- Previous lecture has considered a general model of the store
- This lecture considers the specifics for a virtual machine called ***RPAL***
- A stack machine architecture
  - operators take operands from the top of stack
- Support for managing stack frames
  - variables are accessed relative to these frames
- A typed architecture
  - all data types occupy ***one*** storage location – even strings!!

A stack architecture – the stack frame



Stack frame management

- **MST L 0**
  - mark the stack preparatory to function/procedure call
  - L = level difference between call and declaration
- **CAL M A**
  - procedure call
  - M = number of parameters
  - A = target address
- **INC 0 I**
  - increment top-of-stack pointer (to allocate space for local variables)
- **OPR 0 0**
  - procedure return (i.e. no result)
- **OPR 0 1**
  - function return (i.e. result at TOS left on TOS)

Loading constants

- **LCB 0 B**
  - load byte-length integer constant onto the stack
- **LCI 0 I**
  - load integer constant onto the stack
- **LCR 0 R**
  - load real constant onto the stack (not used)
- **LCS 0 S**
  - load string literal onto the stack
  - only used for debugging
- **LDU 0 0**
  - load an undefined or void value (not used)
- **OPR 0 17**
  - load constant ***true***
- **OPR 0 18**
  - load constant ***false***

Accessing variables

- **LDA L D**
  - load the absolute address of the variable onto the stack
- **LDI 0 0**
  - load the value stored at the address indicated by the value at the top of stack
- **LDV L D**
  - load the value of a variable onto the stack
- **STI 0 0**
  - load top of stack -1 into variable at address top of stack
- **STO L D**
  - store into a variable
- Where:
  - L = level difference to definition stack frame
  - D = offset within the stack frame

Expression evaluation

- Load values onto stack and perform operations on top of stack
- Example: a (1,2) = a (1,2) + b(0,1) * 5

```
LDV   1   2     load a
LDV   0   1     load b
LCI   0   5     load constant 5
OPR   0   5     multiply
OPR   0   3     add
STO   1   2     store result
```


A typed architecture

- A typed architecture
  - don't have to worry about how many bytes an integer occupies
  - same even applies to strings!

Type conversion

- Type conversion is required
- **OPR 0 25**
  - integer-to-real conversion (not used)
- **OPR 0 26**
  - real-to-integer conversion (not used)
- **OPR 0 27**
  - integer-to-string conversion
- **OPR 0 28**
  - real-to-string conversion
- **OPR 0 32**
  - short-to-integer conversion
- **OPR 0 33**
  - integer-to-short conversion

Flow of control

- Jumps
- **JIF 0 A**
  — jump if false to address A
- **JMP 0 A**
  — jump to address A

- Exceptions
- **SIG 0 I**
  — raise signal I
- **REH 0 A**
  — register exception handler at address A (not used)
- **OPR 0 31**
  — raises exception matching integer at TOS (not used)

Flow of control

- Example: while loop

```
while (a (1,2) < b (0,1)) do
    b (0,1) = b (0,1) – 1;
```

- Code:

```
L1:  LDV  1  2    load a
     LDV  0  1    load b
     OPR  0  12   compare for less than
     JIF  0  L2   exit loop if false
     LDV  0  1    load b
     LCI  0  1    load 1
     OPR  0  4    subtract
     STO  0  1    store
     JMP  0  L1   continue
L2:  end of loop
```


Exceptions

- We use exceptions to flag run-time errors:

- SIG 0 1
  — run time errors, including range checks

- SIG 0 2
  — falling through a function without a return

- You do not need to handle exceptions
- Let the machine terminate with an unhandled exception

Target code for simple program

- Consider sample program in RCL handout:

```
program circle;
   short lengthSide, rotateSteps, side;
   void edge(short len);
   {
     step (len, len);
     step (rotateSteps, -rotateSteps);
     return;
   } edge;
   {
     lengthSide = 127;
     rotateSteps = 75;
     side = 1;
     while ( side <= 16 ) do
     {
       side = side + 1;
       edge (lengthSide);
     };
   } circle;
```

Variable initialisation

- Initialise local variables of main program:

```
program circle;
   short lengthSide, rotateSteps, side;
   ...
   {
     lengthSide = 127;
     rotateSteps = 75;
     ...
   } circle;

   INC   0   3      make space on stack
   LCB   0   127    initial value for first variable
   STO   0   0      store it
   LCB   0   75     initial value for second variable
   STO   0   1      store it
```


While loop

```
program circle;
   short lengthSide, rotateSteps, side;
   ...
     while ( side <= 16 ) do
     {
        ...
     };
   } circle;

   loop:  LDV   0   2      evaluate loop condition
          LCB   0   16
          OPR   0   15
          JIF   0   done   skip if no longer true
          ...
          JMP   0   loop   back to start of loop
   done:
```


Function call

```
program circle;
   short lengthSide, rotateSteps, side;
   void edge(short len);
   {
     ...
   } edge;
   {
     ...
       edge (lengthSide);
     ...
   } circle;

   MST   1   0        mark the stack - 1 level
   LDV   0   0        load parameters
   CAL   1   edge     call function
                      void function - no result
```


Function body

```
program circle;
   ...
   void edge(short len);
   {
     step (len, len);
     ...
   } edge;

   edge:     LDV   0   0    load parameter
             OPR   0   23   duplicate - efficient !?
             OPR   0   33   call step function
             ...
             OPR   0   0    return
             SIG   0   2    trap function fall-through
```

Symbolic representation

- We choose to use XML as a symbolic representation of RCL programs
- Allows us to use symbolic labels
- Allows the possibility of linking modules (but this is not required)

© Charles Lakos