Simone Campanoni
simonec@eecs.northwestern.edu
Outline

• L2

• The L2 compiler
Introduction to L2

• Like L1, but we have variables in L2!

Variable is a powerful abstraction
Compilers have to deal with the complexity of implementing that abstraction

Example of L1 programs

L2 programs

To remember a value

(:go
  (:go 0 1
    mem rsp 0 <- 5
    //code
    rdi <- mem rsp 0
    rdi <<= 1
    mem rsp 0 <- rdi
    return
  )
)

(:go
  (:go 0 0
    %myVar1 <- 5
    //code
    %myVar1 <<= 1
    rdi <- %myVar1
    call print 1
    return
  )
)

(:go
  (:go 0
    mem rsp 0 <- 5
    //code
    mem rsp 0 <- rdi
    return
  )
)

(:go
  (:go 0
    %myVar1 <- 5
    //code
    %myVar1 <<= 1
    rdi <- %myVar1
    call print 1
    return
  )
)
Variables in L2

- Variables (on top of registers)
- L2 variables are function local

```assembly
:myF
  1
  %myVar <- 5
  rdi++
  mem rsp -8 <- :RET
  call :myF2 1
  :RET
  rdi <- %myVar
  call print 1
  return
)

:myF2
  1
  %myVar <- rdi
  %myVar++
  return
)```

p ::= (label f+)
f ::= (label N N i+)
i ::= w <- s | w <- mem x M | mem x M <- s |
    w aop t | w sop sx | w sop N | mem x M += t | mem x M -= t | w += mem x M | w -= mem x M |
    w <- t cmp t | cjump t cmp t label | label | goto label |
    return | call u N | call print 1 | call input 0 | call allocate 2 | call tensor-error F |
    w ++ | w -- | w @ w w E
w ::= a | rax | rbx | rbp | r10 | r11 | r12 | r13 | r14 | r15
a ::= rdi | rsi | rdx | sx | r8 | r9
sx ::= rcx
s ::= t | label
t ::= x | N
u ::= w | label
x ::= w | rsp
aop ::= += | -= | *= | &=
sop ::= <<= | >>=
cmp ::= < | <= | =
E ::= 1 | 2 | 4 | 8
F ::= 1 | 3 | 4
M ::= multiplicative of 8 constant (e.g., 0, 8, 16)
N ::= (+|-)? [1-9][0-9]*
label ::= sequence of chars matching :[a-zA-Z_][a-zA-Z_0-9]*
\[
p := (\text{label } f^*)
\]
\[
f := (\text{label } N \ i^*)
\]
\[
i := w <- s \mid w <- \text{mem } x M \mid \text{mem } x M <- s \mid w <- \text{stack-arg } M \mid
\]
\[
\begin{align*}
&w \ \text{aop } t \\
&w \ \text{sop } sx \\
&w \ \text{sop } N \\
&\text{mem } x M += t \\
&\text{mem } x M -= t \\
&w += \text{mem } x M \\
&w -= \text{mem } x M \\
&\text{return} \\
&\text{call } u \ N \\
&\text{call print } 1 \\
&\text{call input } 0 \\
&\text{call allocate } 2 \\
&\text{call tensor-error } F \\
&w ++ \\
&w -- \\
&w \ @ \ w \ w \ E
\end{align*}
\]
\[
w := a \mid \text{rax} \mid \text{rbx} \mid \text{rbp} \mid \text{r10} \mid \text{r11} \mid \text{r12} \mid \text{r13} \mid \text{r14} \mid \text{r15}
\]
\[
a := \text{rdi} \mid \text{rsi} \mid \text{rdx} \mid \text{sx} \mid \text{r8} \mid \text{r9}
\]
\[
sx := \text{rcx} \mid \text{var}
\]
\[
s := t \mid \text{label}
\]
\[
t := x \mid N
\]
\[
u := w \mid \text{label}
\]
\[
x := w \mid \text{rsp}
\]
\[
aop := += \mid -= \mid *= \mid &=
\]
\[
sop := <<= \mid >>=
\]
\[
\text{cmp} := < \mid <= \mid =
\]
\[
E := 1 \mid 2 \mid 4 \mid 8
\]
\[
F := 1 \mid 3 \mid 4
\]
\[
M := \text{multiplicative of 8 constant (e.g., 0, 8, 16)}
\]
\[
N := (+|-)? [1-9][0-9]*
\]
\[
\text{label} := \text{sequence of chars matching } :[a-zA-Z_] [a-zA-Z_0-9]*
\]
\[
\text{var} := %\text{name}
\]
A problem for L1 developers

• We want to print the last argument of :myF plus 1
• Is there any bug in our L1 program?
• We need to save r10
• Is there any bug in our L1 program?
A problem for L1 developers

• We want to print the last argument of :myF plus 1
• Is there any bug in our L1 program?
• We need to save r10
• Is there any bug in our L1 program?
The new L2 instruction

• It accesses stack-based arguments
  \( w \leftarrow \text{stack-arg } M \)

• \((\text{stack-arg } 0)\) is always the last stack argument
• \((\text{stack-arg } 8)\) is always the second to last argument

```plaintext
(:myF 8
  r10 <- stack-arg 0
  r10 += 2
  rdi <- r10
  call print 1
  return
)
```

![Stack diagram](image)
Final notes on L2

As for L1:
• The scope of labels is the program
• Values are encoded following the same rules of L1
• Same calling convention of L1
• Same rules for memory heap allocation
• Same undefined behaviors
Tests for homework 1

• Rewrite your L1 programs using the L2 language
  • To compile an L2 program:
    • Use the original framework, which is still available on Canvas
      • Not the one you have modified to implement your L1 compiler
    • cd L2
    • ./L2c L2_PROGRAM.L2
    • ./a.out

• Deadline: check Canvas
Outline

• L2

• The L2 compiler
A compiler

Character stream --- Source code (e.g., C)

Front-end

Middle-end

Back-end

Machine code (e.g., x86_64)
The L2 compiler (L2c)

- To build L2c: translate an L2 program to an equivalent L1

- We need to map L2 variables to registers
  - Register allocation

- We need to translate the L2 instruction
  \( w \leftarrow \text{stack-arg M} \)
Debugging Suggestion: testing your L1 compiler with my L2 compiler

• If your L2c does not pass a test, then the bug can be either
  • in your L1 compiler or
  • In your L2 compiler

• To understand where is the bug, you can mix your and mine compilers

• To compile an L2 program:
  • Compile your own L1 compiler: L1/bin/L1
  • Make sure you have my L2 compiler: L2/bin/L2
  • cd L2
  • ./L2c L2_PROGRAM.L2
  • ./a.out
Register allocation task

• Intra-procedural approach (most used)
  For each function $f$
  Map each variable of $f$ to either a register or to a stack location
  (within locals in the L1 stack)

• Inter-procedural approach
  Map variables of functions in registers
  exploiting their caller-callee links
Task: From Variables to Registers

(:MyVeryImportantFunction

%MyVar1 <- 2
%MyVar2 <- 40
%MyVar3 <- %MyVar1
%MyVar3 += %MyVar2
print %MyVar3
)

Software

MyVar1   MyVar2   MyVar3

Hardware

r8       r9       r10

No overlapping
To register allocators: what are you doing?

(:MyVeryImportantFunction
   %MyVar1 <- 2
   %MyVar2 <- 40
   %MyVar3 <- 0
   %MyVar3 += %MyVar1
   %MyVar3 += %MyVar2
   print %MyVar3
)

- **MyVar1** -> stack (spilled)
- **MyVar2** -> r8
- **MyVar3** -> r9

Two naïve solutions for register allocation:
1. Spill all variables
2. Increase the #registers
A register allocator structure

Register allocator

Map heuristic

f without varX

Current f, varX

Spill

f without variables

f
Register Allocation

A. Spill all variables
B. Puzzle solving
C. Linear scan
D. Graph coloring
E. Integer linear programming