Compiler Construction

L1 -> x86_64

Simone Campanoni
simone.campanoni@northwestern.edu

Before we start

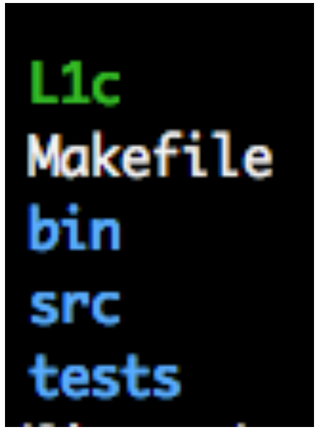
- We use AT&T assembly syntax
  - For compatibility with GNU tools

- rdi += rsi
  - **AT&T:** addq %rsi, %rdi
  - **Intel:** addq %rdi, %rsi

Outline

- Assembler, linker
- From L1 to x86_64
- Calling convention

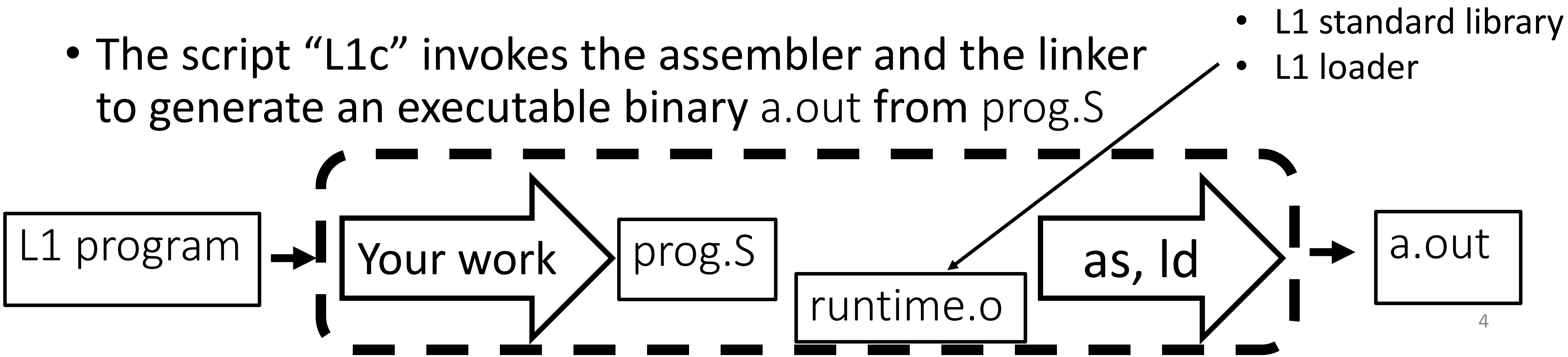
Setup



- You have the structure of a compiler to start from
- Write your assignment in C++ and store the files in “src”

- You work: save x86_64 instructions in prog.S

- The script “L1c” invokes the assembler and the linker to generate an executable binary a.out from prog.S



A simple (incomplete) example

- Write src/compiler.cpp

```
int main(
  int argc,
  char **argv
  ){
  std::ofstream outputFile;
  outputFile.open("prog.S");
  outputFile << "   .text\n" ;
  outputFile.close();
  return 0;
}
```

prog.S structure

```
        .text
        .globl go
go:
        # save callee-saved registers
        pushq   %rbx
        pushq   %rbp
        pushq   %r12
        pushq   %r13
        pushq   %r14
        pushq   %r15

        call  «main label»

        # restore callee-saved registers and return
        popq   %r15
        popq   %r14
        popq   %r13
        popq   %r12
        popq   %rbp
        popq   %rbx
        retq
```

- runtime.c has main
- runtime.c invokes go()
- X86_64 assembly instructions generated from L1 functions are appended here

Example of prog.S

```
(@myGo

  (@myGo
    0 0
    return
  )

)
```

Your work

Substitute @ with _

```
.text
  .globl go
go:
  pushq %rbx
  pushq %rbp
  pushq %r12
  pushq %r13
  pushq %r14
  pushq %r15
  call _myGo
  popq %r15
  popq %r14
  popq %r13
  popq %r12
  popq %rbp
  popq %rbx
  retq
_myGo:
  retq
```

For this L1 program,
We have generated
the equivalent x86_64
version of the code

- Let us assume that all L1 instructions/functions have been translated correctly to x86_64 assembly instructions
- Now what?
  - Assembler
  - Linker

Assembler

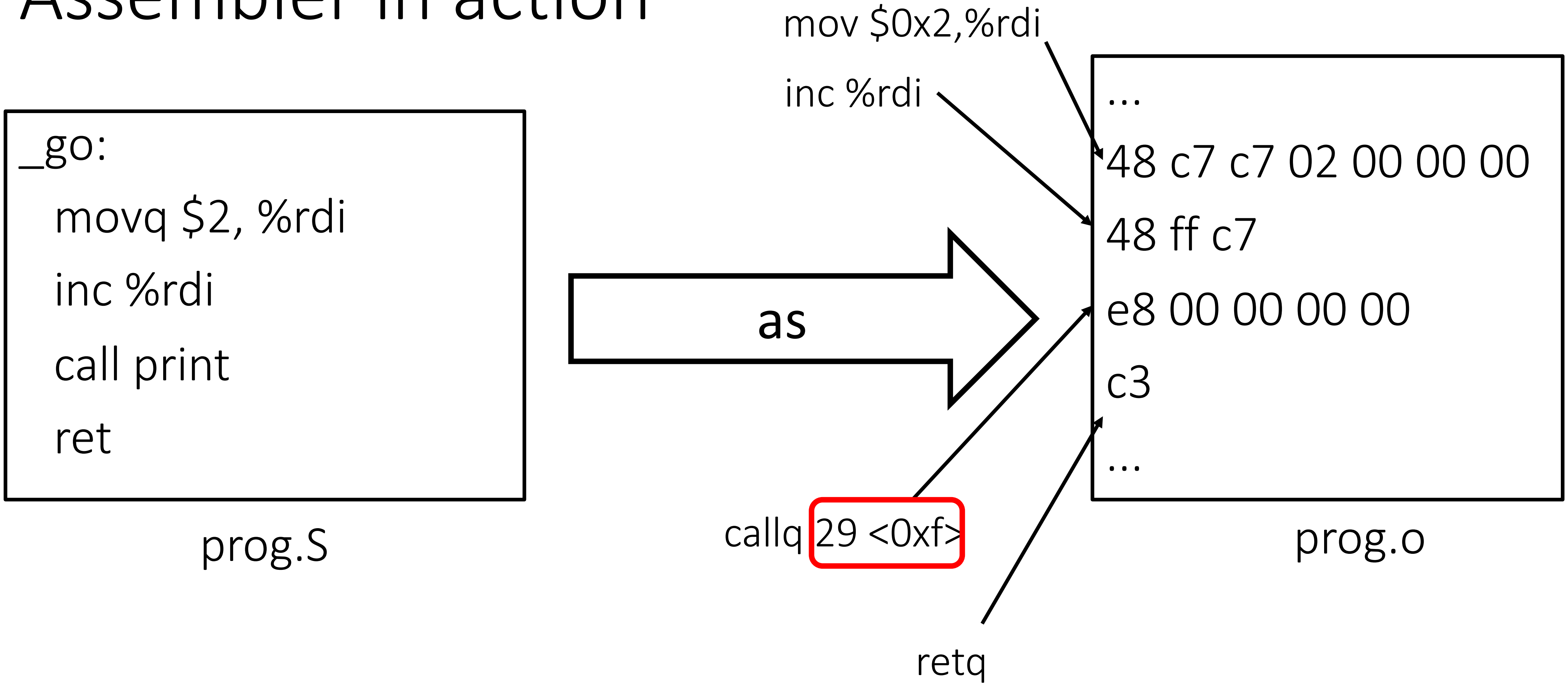
- Translate assembly instructions (e.g., movq $5, %rdi) to their machine code representation (e.g., 48 c7 c7 05 00 00 00)
- Replace labels (e.g., jmp _cool
  ...
  _cool:
  OTHERS
)
  to actual offsets (e.g., jmp 42
  ...
  OTHERS
)
- Embed machine code instructions in an object file (e.g., ELF)

```
$ gcc –c yourfile.c -o yourfile.o
$ file yourfile.o
   yourfile.o: ELF 64-bit LSB relocatable, x86-64 …
```

E xecutable and
L inkable
F ormat

Linux Standard Base

Assembler in action



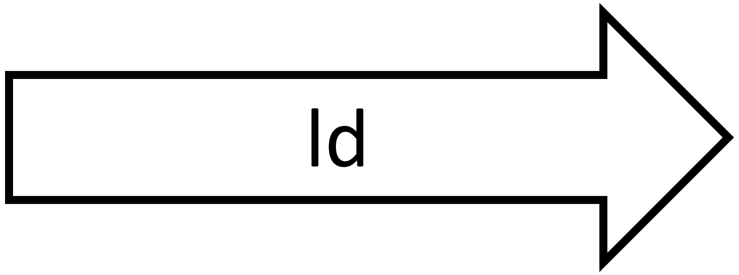
Linker

- Link object files together
  and link them with existing libraries (e.g., libc)

  callq 29 <0xf>

```
...
48 c7 c7 02 00 00 00
48 ff c7
e8 00 00 00 00
c3
...
```

```
...
```




```
...
48 c7 c7 02 00 00 00
48 ff c7
e8 47 ff ff ff
f4   <- hlt
...
```

- Relative offsets can now become absolute
- Undefined symbols (e.g., print) are now resolved

Outline

- Assembler, linker
- From L1 to x86_64
- Calling convention

L1

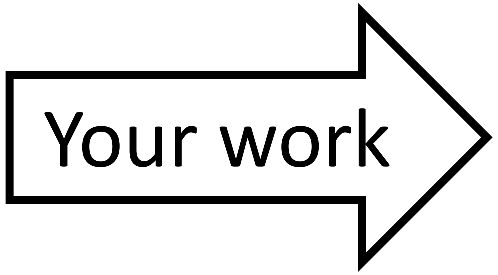
```
p      ::= (l f+)
f      ::= (l N N i+)
i      ::= w <- s | w <- mem x M | mem x M <- s |
           w aop t | w sop sx | w sop N | mem x M += t | mem x M -= t | w += mem x M | w -= mem x M |
           w <- t cmp t | cjump t cmp t label | label | goto label |
           return | call u N | call 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]*
l      ::= @name
label  ::= :name
name   ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*
```

L1 returns

To compile return instructions:

- Add **q** to specify 8 bytes values are returned

return



… # see later

retq

Example of prog.S

```
(@myGo
 (@myGo
  0 0
  return
 )
)
```

Your work

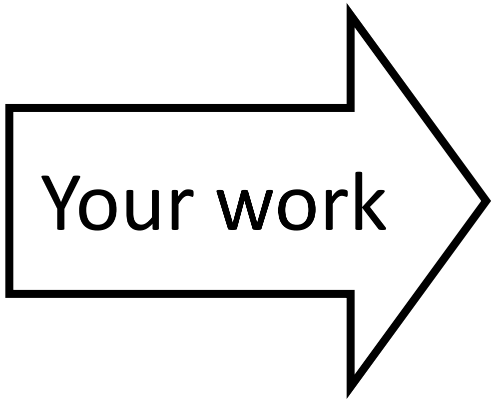
```
.text
  .globl go
go:
  pushq %rbx
  pushq %rbp
  pushq %r12
  pushq %r13
  pushq %r14
  pushq %r15
  call _myGo
  popq %r15
  popq %r14
  popq %r13
  popq %r12
  popq %rbp
  popq %rbx
  retq
_myGo:
  retq
```

L1 assignments

To compile simple assignments:

- prefix registers with **%** and constants and labels with **$**
- Substitute **@** of function names with **_**

rax <- 1 → movq $1, %rax



rax <- rbx → movq %rbx, %rax

rax <- @f → movq $_f, %rax

L1 assignment example

```
(@myGo
  (@myGo
    0 0
    rdi <- 5
    return
  )
)
```

Your work

```
.text
  .globl go
go:
  pushq %rbx
  …
  call _myGo
  popq %r15
  …
  retq
_myGo:
  movq $5, %rdi
  retq
```

L1 assignments to/from memory

To compile memory references:

- put parents around the register and prefix it with the offset

mem rsp 0 <- rdi → movq %rdi, 0(%rsp)

Your work

rdi <- mem rsp 8 → movq 8(%rsp), %rdi

L1 arithmetic operations

Each of the **aop=** operations correspond to their own assembly instruction

```
rdi += rax    ⇒   addq %rax, %rdi

rdi -= rax    ⇒   subq %rax, %rdi

r10 *= r12    ⇒   imulq %r12, %r10

r14 &= r15    ⇒   andq %r15, %r14
```

L1 arithmetic operations (2)

- rdi-- => dec %rdi

- rdi++ => inc %rdi

L1 arithmetic operations in memory

- rdi -= mem rsp 8 => subq 8(%rsp), %rdi

- rdi += mem rsp 8 => addq 8(%rsp), %rdi

- mem rsp 8 -= rdi => subq %rdi, 8(%rsp)

- mem rsp 8 += rdi => addq %rdi, 8(%rsp)

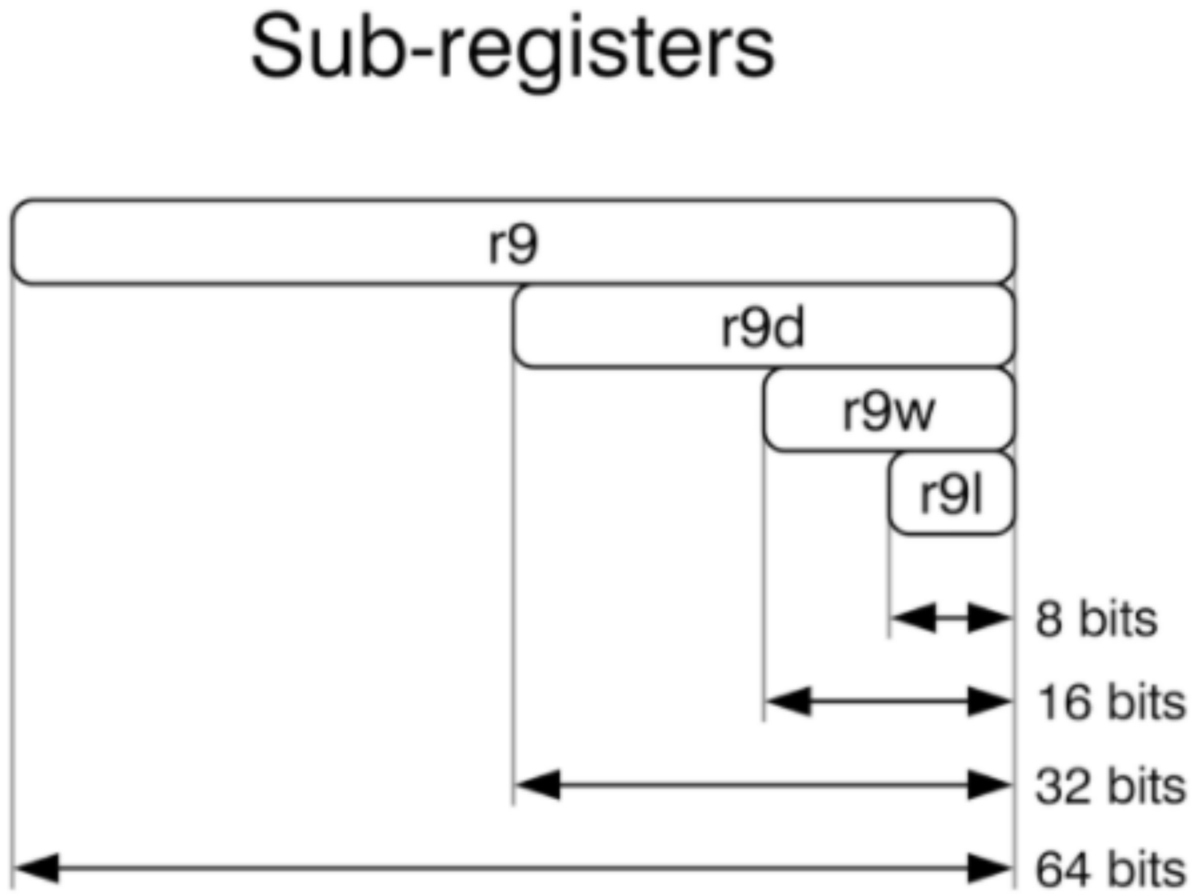
L1 comparisons

- Saving the result of a comparison requires a few extra instructions

```
                                   cmpq %rbx, %rax
rdi <- rax <= rbx        ⇒         setle %dil
                                   movzbq  %dil, %rdi
```

- cmpq updates a condition code in some hidden place (flags register)
- Then, we need to use setle to extract the condition code from this hidden place
- setle, however, needs an 8 bit register as its destination

Intel sub-registers



L1 comparisons

- Saving the result of a comparison requires a few extra instructions

```
rdi <- rax <= rbx    ⇒   cmpq %rbx, %rax
                         setle %dil
                         movzbq  %dil, %rdi
```

- cmpq updates a condition code in some hidden place (flags register)
- Then, we need to use setle to extract the condition code from this hidden place
- setle, however, needs an 8 bit register as its destination
- So we use %dil here because that's the name of the least significant 8 bits of %rdi
- setle updates only those 8 bits; therefore we need movzbq to zero out the rest

L1 comparisons

- Mapping register names to their 8-bit variants

```
r10 → r10b     r11 → r11b     r12 → r12b
r13 → r13b     r14 → r14b     r15 → r15b
r8  → r8b      r9  → r9b      rax → al
rbp → bpl      rbx → bl       rcx → cl
rdi → dil      rdx → dl       rsi → sil
```

L1 comparisons

- Saving the result of a comparison requires a few extra instructions

```
rdi <- rax <= rbx    ⇒   cmpq %rbx, %rax
                         setle %dil
                         movzbq  %dil, %rdi
```

- if we had < we'd need to use
  setg or setl (for less than or greater than)

- If we had = then we would use sete

L1 comparisons with a constant

```
rdi <- rax <= 10
```

⇨

```
cmpq $10, %rax
setle %dil
movzbq %dil, %rdi
```

L1 comparisons with a constant

Must be a register

rdi <- 10 <= rax ⇨

~~cmpq %rax, $10~~
~~setle %dil~~
~~movzbq %dil, %rdi~~

Your compiler must handle this x86_64-specific constraint

L1 comparisons with a constant

```
rdi <- 10 <= rax
```

⇨

```
cmpq $10, %rax
setge %dil
movzbq %dil, %rdi
```

L1 comparisons

So when we don’t have any registers at all, we need to compute the answer at compile time and just use that

```
rdi <- 10 <= 11    ⇒   movq $1, %rdi

rax <- 12 <= 11    ⇒   movq $0, %rax
```

L1 shifting operations

The shifting, **sop=**, operations also use the 8-bit registers, this time for their sources

```
rdi <<= rcx    ⇒   salq %cl, %rdi

rdi >>= 3      ⇒   sarq $3, %rdi
```

The **l** is for "left shift" and the **r** stands for "right shift".

Labels and direct jumps

Labels and gotos are what you might guess; just replace the leading colon with an underscore and add a colon suffix when you define the label

```
:a_label            ⇒   _a_label:

goto :a_label       ⇒   jmp _a_label
```

Labels (2)

- When a label is stored in a memory location, you need to add “$” before the label

mem rsp -8 <- :myLabel

movq $_myLabel, -8(%rsp)

Conditional jumps

- We have the three same cases as for comparisons
- Here, however, we use a jump
  instead of storing the result in a register

```
cjump rax <= rdi :yes
```

⇒

```
cmpq %rdi, %rax
jle _yes
```

- For <=, use jge (jump greater than or equal) or jle
- For <, use jg (jump greater than) or jl (jump less than)
- For =, use je

Conditional jumps with constants

cjump 1 <= 3 :true ⇒ jmp _true

cjump 3 <= 1 :true ⇒

The missing L1 CISC instruction

- The next instruction computes rdi + rsi*4

rax @ rdi rsi 4 ⇨ lea (%rdi, %rsi, 4), %rax

L1 instructions that modify rsp

- call and return instructions

- Function prologue (entry to a function)

```
_myF:
  ...   } prologue
  ...   ┐
        } body
  ...   ┘
  ...   } epilogue
  retq
```

L1 function prologue

- The function prologue allocates locals
- For each local: move the stack pointer by 8 bytes

```
(@myF
   0 3
   …
)
```

⇒

```
_myF:
   subq $24, %rsp   #Allocate locals
   …
```

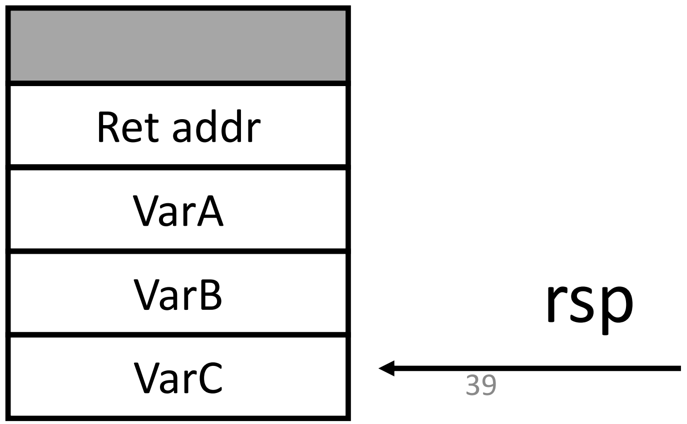
L1 return instructions

The return instruction

- frees locals and … (next slide)
- pops the return address from the stack and jumps to it

```
(@myF
   0 3
   …
   return
)
```

```
…
addq $24, %rsp
retq
```



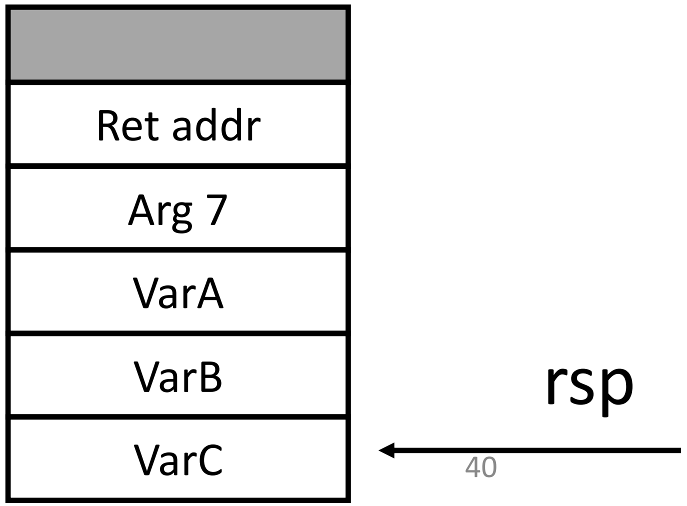
L1 return instructions

The return instruction
- frees locals and stack arguments
- pops the return address from the stack and jumps to it

```
(@myF
   7 3
   ...
   return
)
```

⇨

```
...
addq $32, %rsp
retq
```



L1 call instructions

Calls are translated differently
depending on whether or not they invoke another L1 function

These calls are already considered differently in L1

- Calls to L1 functions: we have to store the return address

```
mem rsp -8 <- :f_ret
call @myCallee
:f_ret
```

- Calls to the L1 runtime: we don't

```
call print 1
```

L1 call instructions to L1 functions

The L1 call instructions to L1 functions

1. moves rsp based on the number of arguments and the return address
2. and then jumps to the callee

```
call @theCallee 11   ⇨   subq $48, %rsp
                         jmp _theCallee

call @aCallee 6      ⇨   subq $8, %rsp
                         jmp _aCallee
```

Why?

We need to allocate space for both arguments passed via the stack and the return address

$(11 - 6)*8 + 8$

Return address

Arguments passed via stack

L1 indirect call instructions

- If call gets a register instead of a function name, then the generated assembly code needs an extra asterisk

```
call rdi 0
```

⇨

```
subq $8, %rsp
jmp *%rdi
```

L1 call instructions to runtime.c functions

The translation of these L1 call instructions

1. Does not need to change rsp
2. Relies on the Intel x86_64 call instruction

call print 1	⇒	call print
call allocate 2	⇒	call allocate
call input 0	⇒	call input

It takes care of

1. identifying the return address
2. storing the return address on the stack
3. jumping to the callee

Function overloading in L1: call instructions to tensor-error

Problem:
all functions/symbols in x86_64 must have unique names

Solution: different functions in runtime.c for different #parameters

call tensor-error 1	call array_tensor_error_null
call tensor-error 3	call array_error
call tensor-error 4	call tensor_error

Outline

- Assembler, linker
- From L1 to x86_64
- Calling convention

x86_64 calling convention

- It is different than L1 calling convention

- Why does it matter for L1 programs?

  call print 1 call allocate 2 call array_error 2

  - runtime.c includes the body of these functions
  - runtime.c is compiled with gcc,
    which follows x86_64 calling convention

Why does it work then?

Registers (same for L1)

Arguments

rdi
rsi
rdx
rcx
r8
r9

First argument

Result

rax

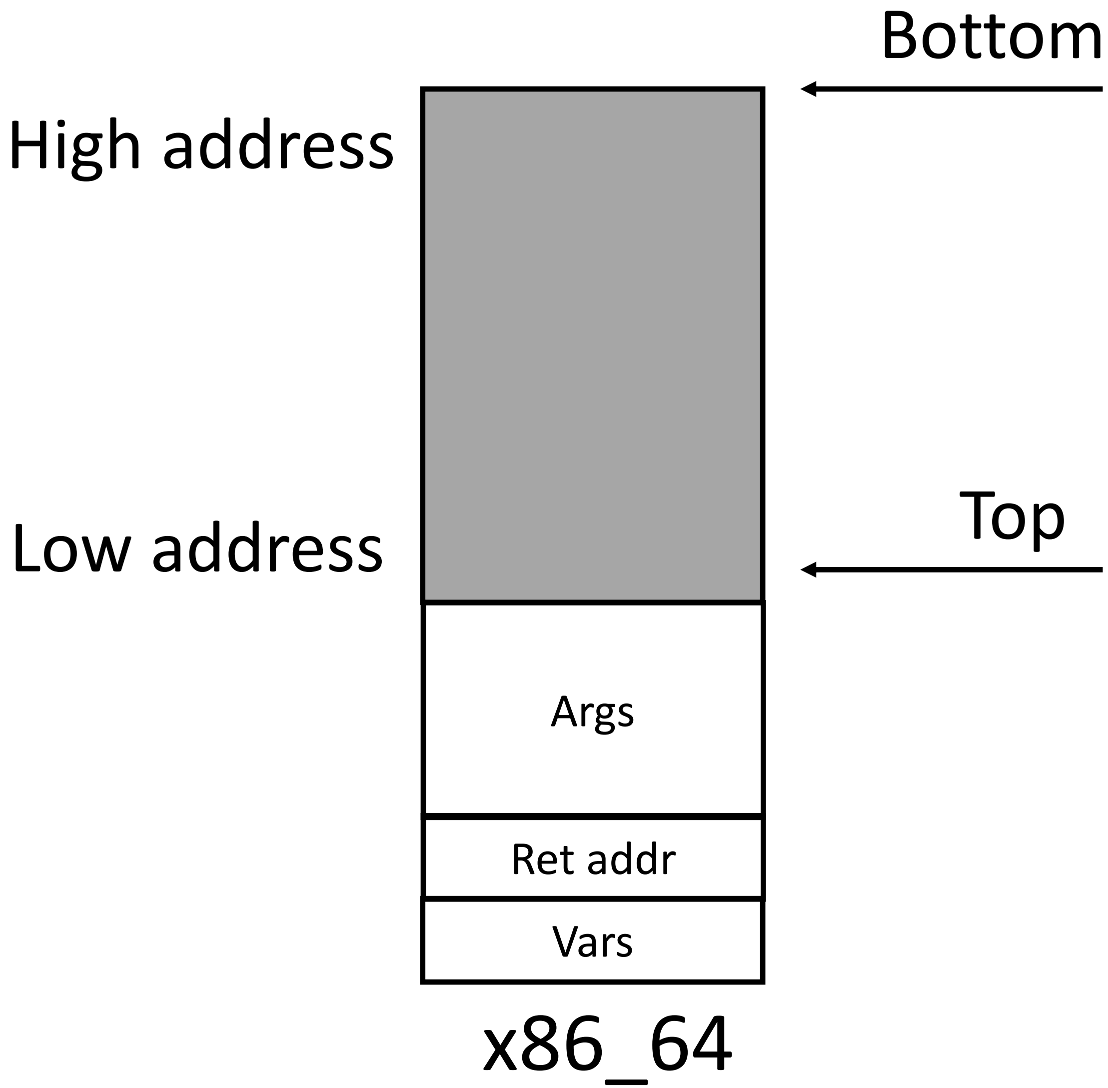
Caller save

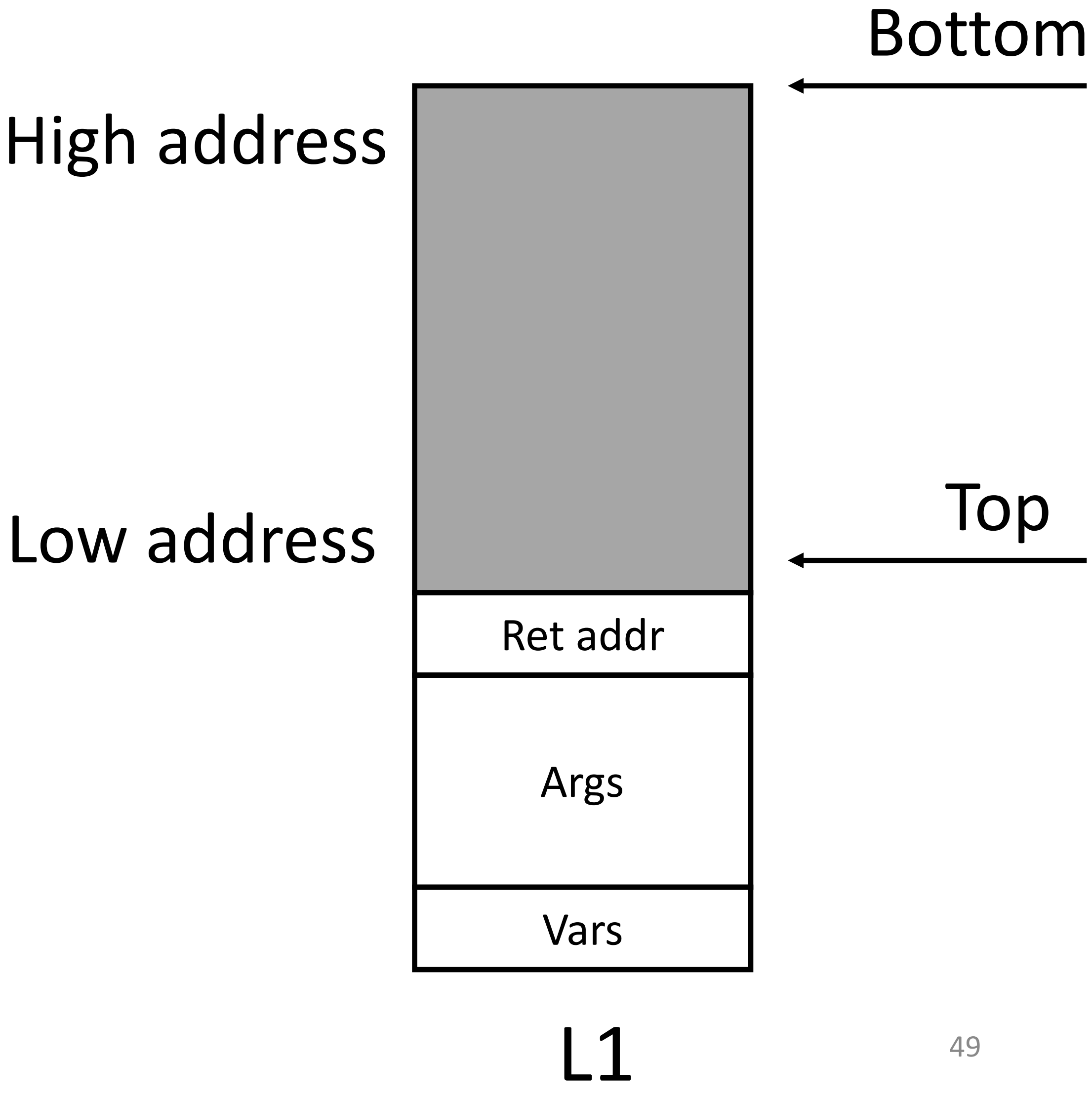
r10
r11
r8
r9
rax
rcx
rdi
rdx
rsi

Callee save

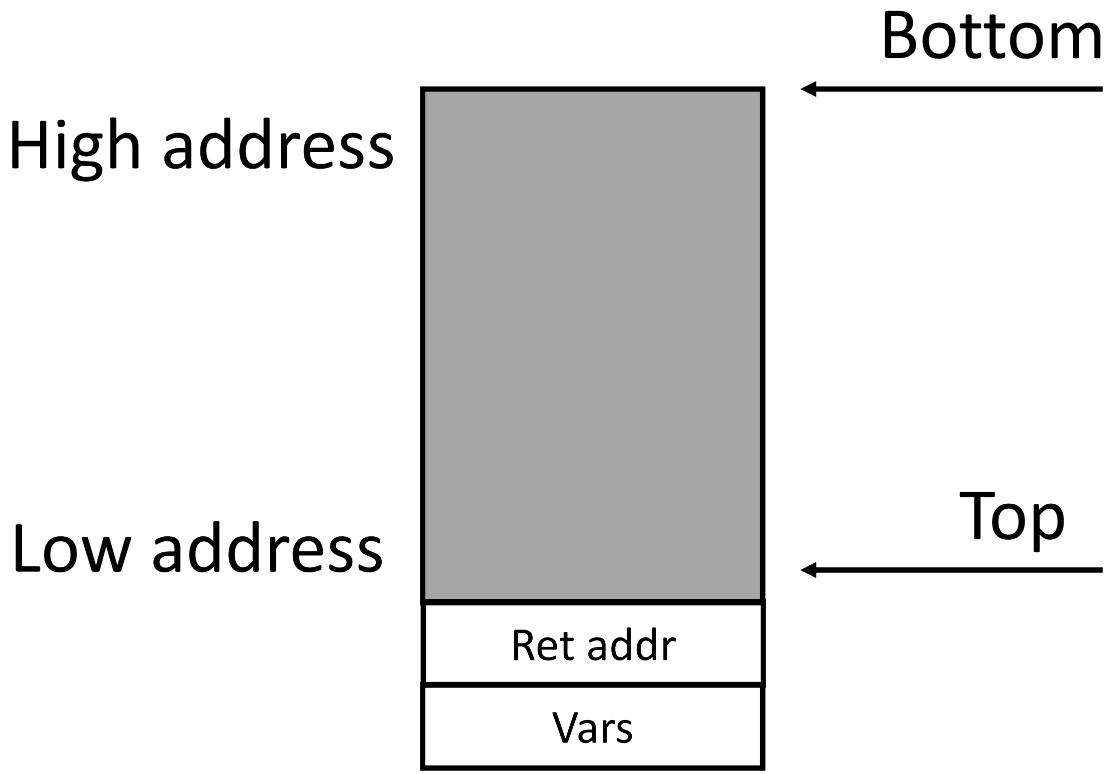
r12
r13
r14
r15
rbp
rbx

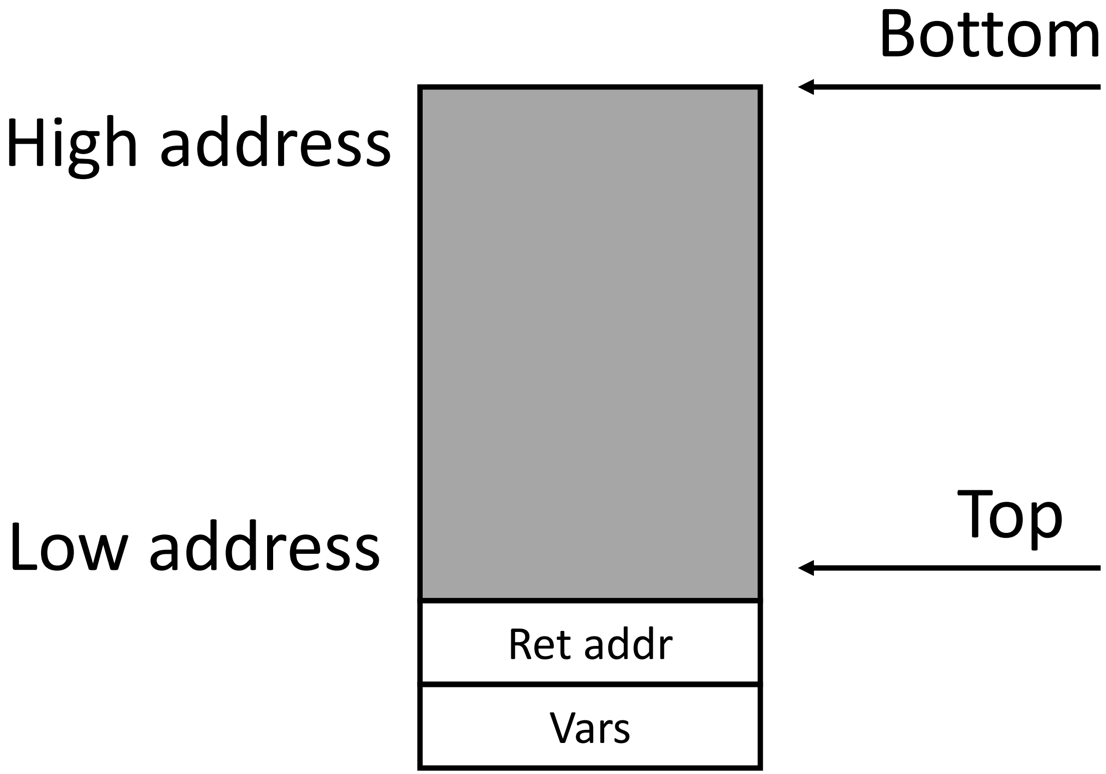
The stack (different compared to L1)





The stack for runtime.c



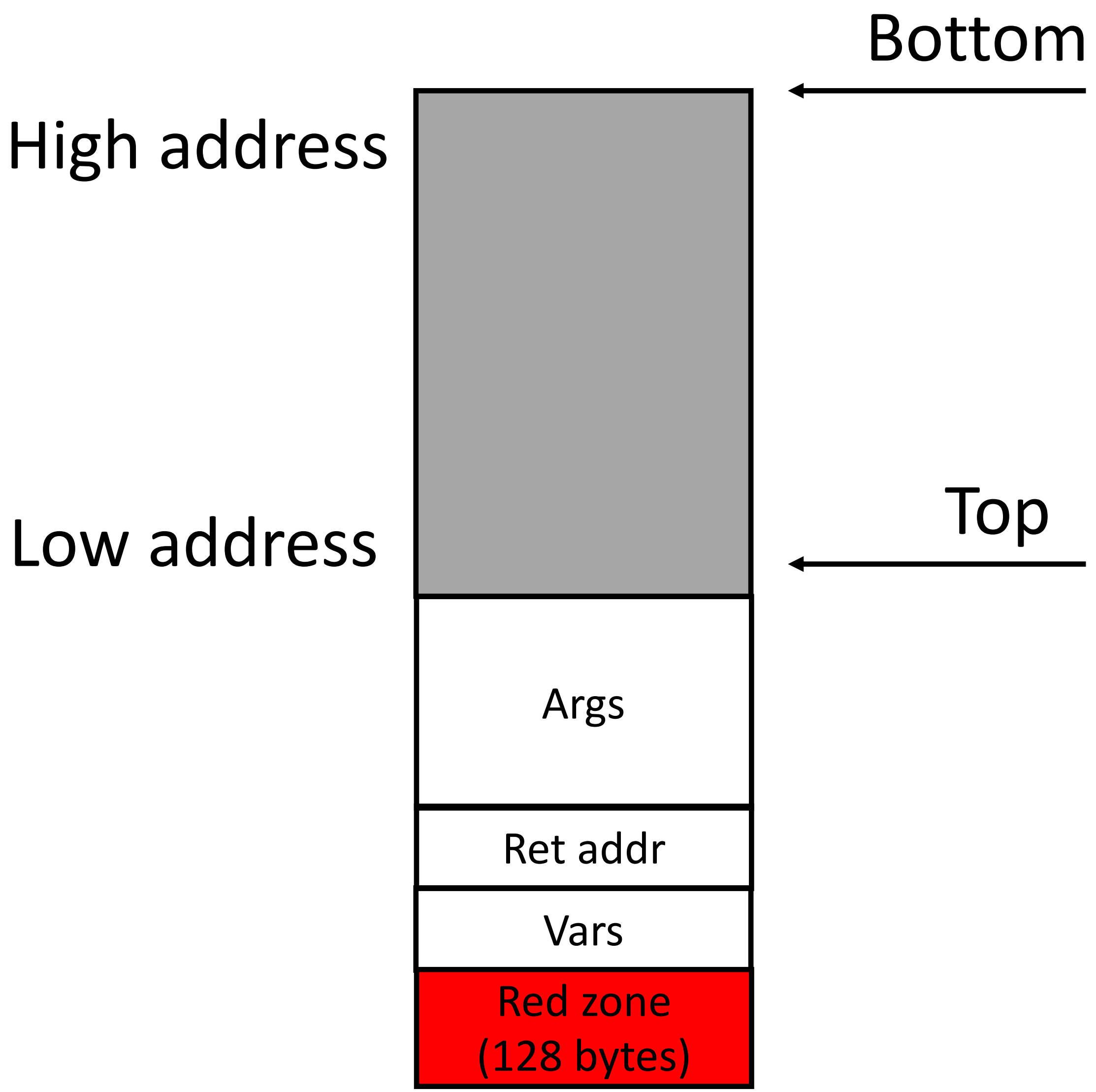


The callee is responsible for allocating and deallocating Vars

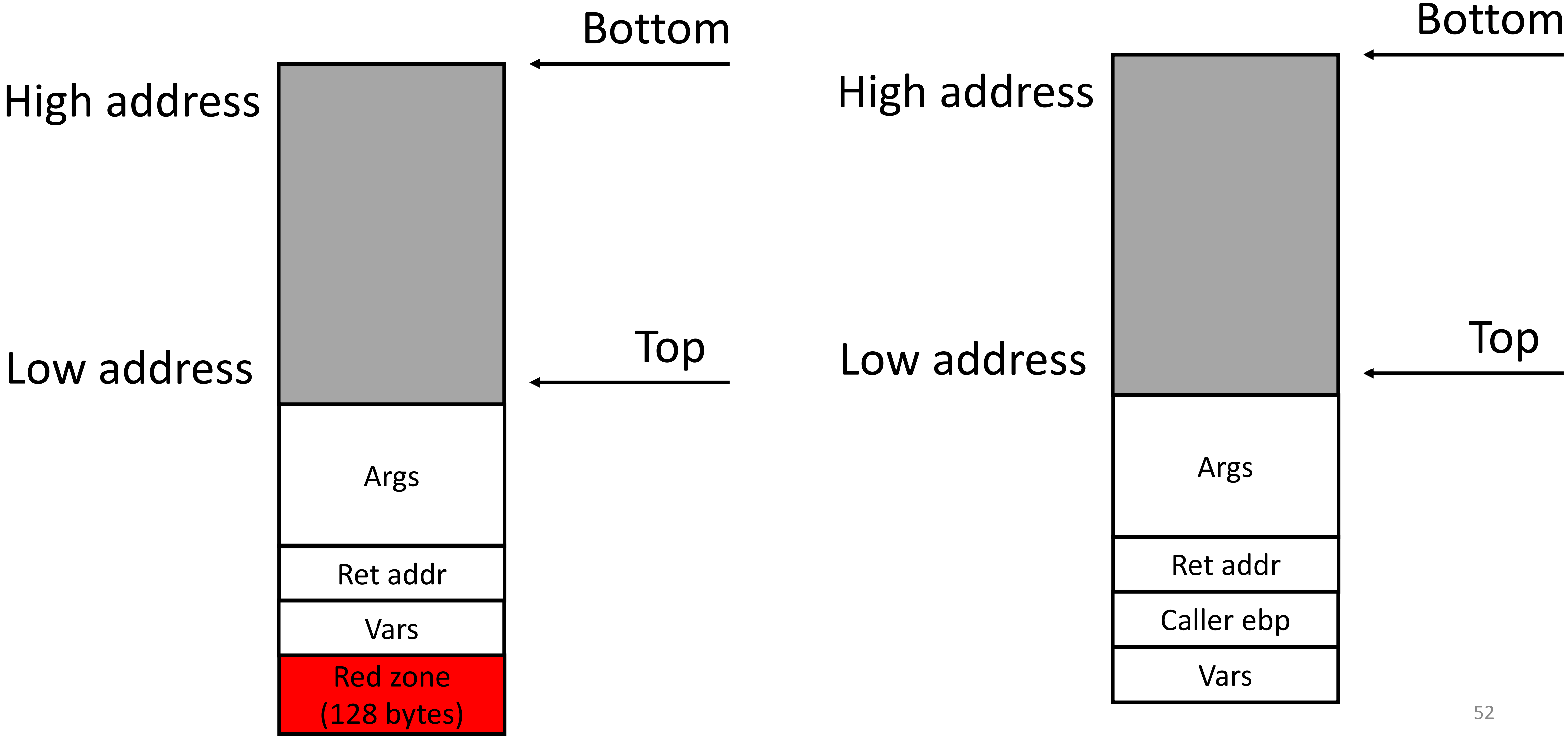
x86_64

L1

More about x86_64 calling convention



x86_64 vs. x86 calling convention



Homework 0

- Develop the L1 compiler
  to translate L1 programs to x86_64 binaries
  - You must follow the translation specified by these slides
  - You must be able to pass all tests

    cd L1 ; make test

- Deadline: see Canvas

Always have faith in your ability

Success will come your way eventually

Best of luck!