Before we start

- **Host to use for your assignments:**
  hanlon.wot.eecs.northwestern.edu
  - Login to it (ssh YOUR_NET_ID@hanlon.wot.eecs.northwestern.edu)
  - Run “bash”
  - Run “source /opt/rh/devtoolset-6/enable”
  - Suggestion: use .bashrc to do it automatically

- **Pair programming**
  - If you want to do it, then send me an email to tell me the members
Homework

• Compiler assignment
  • Day X: you have the assignment
  • Deadline: before the same day of the next week
  • Your compiler has to pass all tests included in the framework

• Program assignment (when I’ll mention in the class)
  • You need to write Y programs
    in the source language of that assignment
    (Y programs per person)
  • Deadline: 2 days

• Late assignment: you cannot be selected as a panelist
Last time

• Compilers translate a source language (e.g., C++) to a destination language (e.g., x86_64)
  • We use them every day
  • If you understand their internals, you better understand (and take advantage of) the tools you rely on
  • Are you interested in computer architectures? Their inputs is the outputs of a compiler
Outline

• Assembler, linker
• L1 language
• Value encoding
• Calling convention
• Heap
Assembler

• Translate assembly instructions (e.g., \texttt{movq $5, \%rdi}) to their machine code representation (e.g., \texttt{48 c7 c7 05 00 00 00})

• Replace labels (e.g., \texttt{jmp \_cool}) to actual offsets (e.g., \texttt{jmp 42})

• Embed machine code instructions in an object file (e.g., ELF)

\begin{verbatim}
$ gcc -c yourfile.c -o yourfile.o
$ file yourfile.o
yourfile.o: ELF 64-bit LSB relocatable, x86-64 ...
\end{verbatim}

\textbf{Linux Standard Base}
Assembler in action

```asm
.go:
    movq $2, %rdi
    inc %rdi
    call print
    ret
```

```
mov $0x2,%rdi
inc %rdi
callq 29 <0xf>
retq
```

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

```
prog.S
```

```
prog.o
```
Linker

• Link object files together and link them with existing libraries (e.g., libc)
  
  callq 29 <0xf>

Relative offsets can now become absolute
• Undefined symbols (e.g., print) are now resolved
Putting it all together

- Let’s say we have assembly instructions in prog.S
- Let’s say we have a runtime in runtime.o
- To invoke the assembler:
  
  \$ as –o prog.o prog.S

- To invoke the linker

  \$ gcc –o a.out prog.o runtime.o

- How to actually invoke the linker (ld) requires platform-specific information
  - gcc invokes the linker for your platform (hanlon)
  - You want to know how? Run “gcc –v –o a.out prog.o runtime.o”
Outline

• Assembler, linker

• L1 language

• Value encoding

• Calling convention

• Heap
The L1 source language

• Similar to a subset of x86_64, but with some abstractions
  \texttt{movq} $1, \%rax \quad \text{rax} \leftarrow 1

• L1 is going to be the input of your first compiler L1c

• The output of L1c is an executable ELF binary
  • You will rely on the GNU assembler and linker
L1 program

\[ p \ ::= (\text{label } f^+) \]

\[ \text{label} \ ::= \text{sequence of chars matching :}[a-zA-Z_][a-zA-Z0-9]^* \]

\[ (:\text{go} \quad \text{The entry point of this L1 program is the function } :\text{go} \]
\[ f1 \quad \text{One of these functions must be } :\text{go} \]
\[ f2 \]

\]
L1 function

\[ p ::= (\text{label } f^+) \]

\[ \text{label ::= sequence of chars matching :\{a-zA-Z_\}\{a-zA-Z_0-9\}*} \]

\[ f ::= (\text{label } N N i^+) \]

\[ (:\text{go} \]
\[ \quad 4 2 \]
\[ \quad i1 \]
\[ \quad i2 \]
\[ ) \]
L1 instruction: assignment

\[ p ::= (\text{label } f^+) \]

\[ \text{label ::= sequence of chars matching :[a-zA-Z_]a-zA-Z_0-9]* } \]

\[ f ::= (\text{label } N N i^+) \]

\[ i ::= w <- s \]

\[ w ::= a | \text{rax} | \text{rbx} | \text{rbp} | r10 | r11 | r12 | r13 | r14 | r15 \]

\[ a ::= \text{rdi} | \text{rsi} | \text{rdx} | \text{rcx} | r8 | r9 \]

\[ s ::= x | N | \text{label} \]

\[ x ::= w | \text{rsp} \]
L1 example

(:go
  (:go
    4 2
    rdi <- 5
    rax <- rdi
  )
)
)
L1 instruction: load

\[ p ::= \text{(label } f^+) \]

label ::= sequence of chars matching :[a-zA-Z_][a-zA-Z_0-9]*

\[ f ::= \text{(label } N N i^+) \]

\[ i ::= w \leftarrow s \mid w \leftarrow \text{mem } x \ M \]

\[ w ::= a \mid \text{rax} \mid \text{rbx} \mid \text{rbp} \mid \text{r10} \mid \text{r11} \mid \text{r12} \mid \text{r13} \mid \text{r14} \mid \text{r15} \]

\[ a ::= \text{rdi} \mid \text{rsi} \mid \text{rdx} \mid \text{rcx} \mid \text{r8} \mid \text{r9} \]

\[ s ::= x \mid N \mid \text{label} \]

\[ x ::= w \mid \text{rsp} \]

\[ M ::= N \text{ times } 8 \text{ (e.g., 0, 8, 16)} \]
L1 example

(:go
  (:go
    4 2
    rdi <- 5
    rax <- rdi
    rbx <- mem rdi 8
  )
)
)
L1 instruction: store

\[ p ::= (\text{label } f^+) \]

\[ \text{label} ::= \text{sequence of chars matching } :[a-zA-Z_][a-zA-Z_0-9]^* \]

\[ f ::= (\text{label } N N i^+) \]

\[ i ::= w \leftarrow s \mid w \leftarrow \text{mem } x M \mid \text{mem } x M \leftarrow s \]

\[ w ::= a \mid \text{rax} \mid \text{rbx} \mid \text{rbp} \mid \text{r10} \mid \text{r11} \mid \text{r12} \mid \text{r13} \mid \text{r14} \mid \text{r15} \]

\[ a ::= \text{rdi} \mid \text{rsi} \mid \text{rdx} \mid \text{rcx} \mid \text{r8} \mid \text{r9} \]

\[ s ::= x \mid N \mid \text{label} \]

\[ x ::= w \mid \text{rsp} \]

\[ M ::= N \text{ times } 8 \text{ (e.g., 0, 8, 16)} \]
L1 instruction: arithmetic operations

\[ p ::= (\text{label } f^+) \]
\[ \text{label ::= sequence of chars matching } [a-zA-Z_][a-zA-Z_0-9]^* \]
\[ f ::= (\text{label } N N \ i^+) \]
\[ i ::= w <- s \mid w <- \text{mem } x \ M \mid \text{mem } x \ M <- s \]
\[ \quad \quad \mid w \ \text{aop } t \]
\[ \text{aop ::= } += \mid -= \mid *= \mid &= \]
\[ t ::= x \mid N \]
L1 example

(:go
 (:go
  4 2
  rdi <- 5
  rdi += 2
 )
 )
)
L1 instruction: shifting

\[ p ::= (\text{label } f^+) \]
\[ \text{label} ::= \text{sequence of chars matching} :[a-zA-Z_][a-zA-Z_0-9]^* \]
\[ f ::= (\text{label } N N i^+) \]
\[ i ::= w \leftarrow s \mid w \leftarrow \text{mem } x \ M \mid \text{mem } x \ M \leftarrow s \]
\[ \quad \mid w \ aop \ t \mid w \ sop \ rcx \]
\[ aop ::= += \mid -= \mid *= \mid &= \]
\[ t ::= x \mid N \]
\[ sop ::= <=\mid >>= \]
L1 instruction: shifting

p ::= (label f+)
labeled ::= sequence of chars matching :[a-zA-Z_][a-zA-Z0-9]*
f ::= (label N N i+)
i ::= w <- s | w <- mem x M | mem x M <- s
    | w aop t | w sop rcx | w sop N
aop ::= += | -= | *= | &=
t ::= x | N
sop ::= <<= | >>=
L1 instruction: arithmetic operations (2)

\[ p ::= (\text{label } f^+) \]

label ::= sequence of chars matching :[\text{a-zA-Z}][\text{a-zA-Z_0-9}]^*

\[ f ::= (\text{label } N \text{ N } i^+) \]

\[ i ::= w <- s | w <- \text{mem x M} | \text{mem x M} <- s \]

| w aop t | w sop rcx | w sop N
| mem x M += t
| mem x M -= t
| w += mem x M
| w -= mem x M

Notice you cannot have both operands in memory
L1 instruction: comparison

\[
p ::= (\text{label } f^+)\]

\[
\text{label} ::= \text{sequence of chars matching } [:a-zA-Z_][a-zA-Z_0-9]^*\]

\[
f ::= (\text{label } N \text{ } N \text{ } i^+)\]

\[
i ::= \ldots\]

\[
| \text{ w } <- \text{ t cmp t}\]

\[
\text{cmp} ::= < | <= | =\]

Notice there is no > or >=
L1 example

(:go
 (:go
   4 2
   rax <- 5
   rdi <- rax <= 3
 )
)
)
L1 instruction: conditional jump

\[
p ::= (\text{label } f^+) \\
\text{label} ::= \text{sequence of chars matching } :[a-zA-Z_][a-zA-Z0-9]^* \\
f ::= (\text{label } N N i^+) \\
i ::= \ldots \begin{align*}
| & \text{w }<- \text{ t cmp t} \\
| & \text{cjump t cmp t label label}
\end{align*}
\]
\[
\text{cmp} ::= < | \leq | =
\]
L1 example

(:go
  (:go
    4 2
    rax <- 5
    rdi <- rax <= 3
    cjump rdi = 1 :true :false
    cjump rax <= 3 :true :false
  )
)
)
L1 instruction: label and jump

\[ p ::= (\text{label } f^+) \]

\[ \text{label ::= sequence of chars matching :}[a-zA-Z_][a-zA-Z_0-9]^* \]

\[ f ::= (\text{label } N N i^+) \]

\[ i ::= \ldots \]

\[ \quad | w \leftarrow t \text{ cmp } t \]

\[ \quad | \text{cj} \text{jump } t \text{ cmp } t \text{ label label} \]

\[ \quad | \text{cj} \text{jump } t \text{ cmp } t \text{ label} \quad \text{Fall-through semantic} \]
L1 example

(:go
  (:go
    4 2
    rax <- 5
    rdi <- rax <= 3
    cjump rdi = 1 :true
  )
)
)
L1 instruction: label and jump

\( p ::= \text{label f}^+ \)

\( \text{label} ::= \text{sequence of chars matching :}[a-zA-Z_][a-zA-Z_0-9]^* \)

\( f ::= \text{label N N i}^+ \)

\( i ::= \ldots \)

\( | w \gets t \ cmp t \)

\( | \text{cj} \ \text{jump} t \ cmp t \ label \ label \)

\( | \text{cj} \ \text{jump} t \ cmp t \ label \)

\( | \text{label} \)

\( | \text{goto} \ label \)
L1 example

(:go
 (:go
  4 2
  rax <- 5
  rax += 2
  cjump rax <= 3 :LOW :HIGH
  :LOW
  rax += 4
  goto :HIGH
  :HIGH
 ))
L1 another example

(:go
  (:go
    4 2
    rax <- 5
    rax += 2
    cjump 3 < rax :HIGH
    rax += 4
    goto :HIGH
    :HIGH
  ))
L1 instruction: label and jump

\[ p ::= (\text{label } f^+) \]

\[ \text{label} ::= \text{sequence of chars matching } :[a-zA-Z_][a-zA-Z_0-9]^* \]

\[ f ::= (\text{label } N \ N \ i^+) \]

\[ i ::= \ldots \]

\[ \mid w \leftarrow t \ \text{cmp} \ t \]

\[ \mid \text{cjump } t \ \text{cmp} \ t \ \text{label} \ \text{label} \]

\[ \mid \text{cjump } t \ \text{cmp} \ t \ \text{label} \]

\[ \mid \text{label} \]

\[ \mid \text{goto } \text{label} \]

The scope of labels is the program
L1 another example

(:F1
 (:F1
   0 0
   :L1
 )
 (:F2
   0 0
   :L1
 )
)
L1 instruction: return

\[ p ::= (\text{label } f^+) \]

\[ \text{label ::= sequence of chars matching } [a-zA-Z_][a-zA-Z_0-9]^* \]

\[ f ::= (\text{label } N \ N \ i^+) \]

\[ i ::= \ldots \]

\[ \text{ } | \text{ return} \]
L1 example

(:go
 (:go
  4 2
  rax <- 5
  cjump rax <= 3 :true :false
  :true
  return
  :false
  rax += 1
  return
))
L1 instruction: call

p ::= (label f+)

label ::= sequence of chars matching :[a-zA-Z_][a-zA-Z_0-9]*

f ::= (label N N i+)

i ::= ...
    | call u N
    Number of arguments of the called function (a.k.a. callee)

u ::= w | label
L1 example

Why do we have redundant information in L1?
To simplify the L1 compiler (your work)

They must match

Number of parameters of the function
L1 instruction: call

p ::= (label f+)

label ::= sequence of chars matching :[a-zA-Z_][a-zA-Z_0-9]*

f ::= (label N N i+)

i ::= ...
    | call u N
    | call print 1
L1 example

(:go
 (:go
  4 2
  rdi <- 5
  call print 1
  return
 )
)

The calling convention will be explained soon
L1 instruction: call

\[ p ::= (\text{label } f^+) \]

\[ \text{label ::= sequence of chars matching :}[a-zA-Z_][a-zA-Z_0-9]^* \]

\[ f ::= (\text{label } N \ N \ i^+) \]

\[ i ::= \ldots \]

\[ \begin{align*}
| \ & \text{call } u \ N \\
| \ & \text{call print } 1 \\
| \ & \text{call allocate } 2 \\
| \ & \text{call array-error } 2
\end{align*} \]
L1 instruction: misc

\[ p ::= \text{(label } f^+\text{)} \]

\[ \text{label ::= sequence of chars matching }[:a-zA-Z_][a-zA-Z_0-9]^* \]

\[ f ::= \text{(label } N N \text{ i}^+\text{)} \]

\[ i ::= \ldots \]

\[ \begin{align*}
| & w++ \\
| & w-- \\
| & w \@ w w E
\end{align*} \]

\[ E ::= 1 \mid 2 \mid 4 \mid 8 \]

\[ \text{Set rax to rdi + (rsi \times 4)} \]

\[ \text{rax @ rdi rsi 4} \]
L1

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 | cjump t cmp t label | label | goto label |
    return | call u N | call print 1 | call allocate 2 | call array-error 2 |
    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
M ::= N times 8
label ::= sequence of chars matching :[a-zA-Z_][a-zA-Z0-9]*
Outline

• Assembler, linker

• L1 language

• Value encoding

• Calling convention

• Heap
High level vs. low level languages

C language

```c
printf("5");
```
You expect the output

5

Back-end languages

```c
rdi <- 5
call print 1
```
You expect the output

? It depends on the encoding scheme designed for correctness
Value encoding in L1

• A value is either an 8 byte integer value or a memory address
• We would like to differentiate between the two
  • Safer programming environment
  • Problem: how to do it?
    • For example:
      mem rdi 8 <- rax
      is the value in rdi a memory address?
• Our solution: using the least significant bit to specify it
  0: it is a memory address
  1: it is an integer value
• Values in L1 are all encoded

Two’s complement integer

00000011
High level vs. low level language L1

C language
printf(“5”);
You expect the output → 5

L1 language
rdi <- 5
call print 1
You expect the output → 2

00000101 → 00000010
Decoding an encoded value

• $x \& 1 = 0$
  $x$ is a memory address

• $x \& 1 = 1$
  $x >> 1$ is a 63 bit two’s complement integer

• Values (integer or addresses) must be encoded for runtime APIs
  • print
  • allocate
  • array-error
L1 example

(:go
 (:go
  0 0
  rdi <- 5
  call print 1
  return
 )
)

- print writes to the terminal the integer value encoded in rdi
- What is going to be the output? 2
Outline

• Assembler, linker

• L1 language

• Value encoding

• Calling convention

• Heap
Calling convention

• How many arguments a given function has?
  call :myF 2

• Where are the arguments stored?

• Who (caller vs. callee) is responsible for what?

• Where is the return value stored?
  rax
First argument

(:go
  0 1
  r10 < - 5
  ...
)

- What about the previous value of r10?
- We want to write our function without knowing the registers used/needed by every possible caller
  - Is it possible to know them all?
- Who is responsible to save the previous value?
  - Are we (the callee)?
  - Are the callers?
  - We need to establish a convention
Registers

- Arguments: rdi, rsi, rdx, rcx, r8, r9
- Result: rax
- Caller save: r10, r11, r8, r9, rax, rcx, rdi, rdx, rsi
- Callee save: r12, r13, r14, r15, rbp, rbx

First argument
Caller save registers (e.g., r10)

```
(:myF
  0 1
  r10 <- 5
  mem rsp 0 <- r10
  mem rsp -8 <- :myF2_ret
  call :myF2 0
  :myF2_ret
  r10 <- mem rsp 0
  rdi <- r10
  call print 1
  return
)
```

```
(:myF2
  0 0
  r10 <- 3
  return
)
```

What is the output?

Whoever generates L1 code (developer, compiler that targets L1) is **responsible** to properly save caller-save registers.
Caller save registers (e.g., r10)

Whoever generates L1 code (developer, compiler that targets L1) is responsible to properly save caller-save registers.

R10 is not used after the call. Hence, we don’t need to save it.
Callee save registers (e.g., r12)

Whoever generates L1 code (developer, compiler that targets L1) is responsible to properly save caller-save registers as well as callee-save registers.
Callee save registers (e.g., r12)

Whoever generates L1 code (developer, compiler that targets L1) is responsible to properly save caller-save registers as well as callee-save registers.
The stack (of plates)
The stack (of plates)
The stack in L1

- High address
- Low address

Stack grows
The stack in L1

- High address
- Low address
  - Ret addr
  - Args
  - Vars

Bottom  Top
The stack in L1

- High address
- Low address
  - Ret addr
  - Arg 7
  - Arg N
  - Vars

Bottom ➔ Top
Stack frame: <= 6 arguments, no locals

(:go
  ...
  mem rsp -8 <- :f_ret
  call :f 1
  :f_ret
    ...
  )
(:f
  1 0
  return
  )
Stack frame: > 6 arguments, no locals

(:go
  rdi<-1
  rsi<-3
  rdx<-5
  rcx<-7
  r8<-9
  r9<-11
  ...
  call :f 7
  ...
)

Arguments
  rdi
  rsi
  rdx
  rcx
  r8
  r9

First argument
(:go

... //passing the first 6 arguments
mem rsp -8 <- :f_ret
mem rsp -16 <- 11
call :f 7

:f _ret
)
(:f
  7 0
rdi <- mem rsp 0
call print 1
return)
Stack frame: <= 6 arguments, 1 local

(:go

...  
mem rsp -8 <- :f_ret  
call :f 1  

:f_ret  
... 
)  
(:f  
1[1]  
return  
)  

Bottom

mem rsp -8 <- :f_ret

Ret addr

?????
L1 program example

Is there a bug? Where?

What is the output?
Stack pointer

- \texttt{rsp} (the stack pointer) is never modified directly

- \texttt{call} and \texttt{return} instructions modify \texttt{rsp} to do their jobs (see the grammar)
Outline

• Assembler, linker

• L1 language

• Value encoding

• Calling convention

• Heap
Heap memory in L1

- Arrays are allocated in the heap
- No explicit deallocation
  - A garbage collector is assumed
- APIs
  - allocate:
    allocate an array of a given number of 64-bit integer elements
  - array-error:
    write to stdout an error message and abort the execution
Heap memory in L1

• allocate
  • Argument 1: number of array elements to allocate
  • Argument 2: 64-bit integer value used to initialize all array elements
  • Return: pointer to the array allocated and initialized

• array-error
  • Argument 1: pointer to a previously allocated array
  • Argument 2: number of the element that has been tried to access that does not exist
Example of L1 program using heap memory

(:go
 (:go
  0 0
  rdi <- 5
  rsi <- 7
  call allocate 2
  return
 )
)

\[
\begin{array}{c}
\text{rax} \\
2 \\
7 \\
7
\end{array}
\]
Example of L1 program using heap memory

(:go
  0 0
  rdi <- 5
  rsi <- 7
  call allocate 2
  rdi <- mem rax 8
  call print 1
  return
)

What is the output?

3

rax + 8

2
7
7
Example of L1 program using heap memory

(:go
  0 0
  rdi <- 5
  rsi <- 7
  call allocate 2
  rdi <- mem rax 16
  call print 1
  return
)

What is the output?

3

rax + 16

2

7

7
Example of L1 program using heap memory

(:go
  0 0
  rdi <- 5
  rsi <- 7
  call allocate 2
  rdi <- mem rax 0
  call print 1
  return
)

What is the output?
Segmentation fault

rdi <<= 1
rdi++

How can we fix this L1 program?
Printing an array

- The API `print` writes to stdout the whole array if its pointer is passed as argument

```plaintext
di <- 5
rsi <- 7
call allocate 2
di <- rax
call print 1
{s:2, 3, 3}
```
Array of arrays

(:go 0 0
 rdi <- 5
 rsi <- 7
 call allocate 2
 rdi <- 7
 rsi <- rax
 call allocate 2
 rdi <- rax
 call print 1
 return)

Allocate an array of 2 integer values
Allocate an array of 3 pointers and initialize them to point to the previously allocated array

What is the output?
{s:3, {s:2, 3, 3}, {s:2, 3, 3}, {s:2, 3, 3}}
Final notes

• Every L1 function must return to its caller with a (return) instruction

```plaintext
(:wrongF
  0 0
  rax <- 1
)
```

• The calling convention must be preserved always

• You can write comments in L1
  • A comment starts with “//” and it comments until the end of the line
  • Example
    ```plaintext
    // This is a comment
    rdi <- 5
    
    // this is another comment
    ```
Tests

• Write 3 non trivial (e.g., more than 30 instructions) L1 programs
  • The name of the files must end with .L1
  • For example: myTest1.L1

• Deadline: 2 days from today at midnight