COS 217: Introduction to Programming Systems

Machine Language
There are many kinds of computer chips out there:

ARM (AARCH64)
Intel x86 series
IBM PowerPC
RISC-V
MIPS
(and, in the old days, dozens more)

Each of these different “machine architectures” understands a different machine language – binary encoding of instructions
Machine Language

The first part of this lecture (today) covers
- A motivating example from Assignment 6: Buffer Overrun
- The AARCH64 machine language

The second part (Wednesday) covers
- The assembly and linking processes
It Gets Much, Much Worse...

Buffer overrun can overwrite return address of a previous stack frame!
- Value can be an invalid address, leading to a segfault, or it can cleverly cause unintended control flow, or even cause arbitrary malicious code to execute!

```c
#include <stdio.h>
int main(void)
{
    char name[12], c;
    int i = 0, magic = 42;
    printf("What is your name?\n");
    while((c = getchar()) != '\n')
        name[i++] = c;
    name[i] = '\0';
    printf("Thank you, %s.\n", name);
    printf("The answer to life, the universe, ",
           "and everything is %d\n", magic);
    return 0;
}
```
Assignment 6: Attack the “Grader” Program

void readString() {
    char buf[BUFSIZE];
    int i = 0;
    int c;
    for (;;) {
        c = fgetc(stdin);
        if (c == EOF || c == 'n')
            break;
        buf[i] = c;
        i++;
    }
    buf[i] = '0';
    for (i = 0; i < BUFSIZE; i++)
        name[i] = buf[i];
}

void getName() {
    printf("What is your name?\n");
    readString();
}
Initially, the name array in BSS is blank (all 0 bits).

Initially, the buf array in readString has garbage.
Memory Map of Stack and BSS Section

Nothing is copied to BSS until the loop filling buf finishes.

You will put your name + \0 into the buf array.
Memory Map of Stack and BSS Section

(readString’s stackframe)

(readString’s stackframe)

(buf[0])

(buf[1])

(...)

(buf[47])

(???)

(getName’s stackframe)

(main’s stackframe)

(SP)

(???)

(???)

(???)

(???)

(???)

(???)

(???)

(???)

(adr x0, grade)

(other instructions go here)

(name[0])

(name[1])

(name[47])

(???)

(‘B’)

(‘o’)

(‘b’)

(‘\0’)

(???)

(x0)

(name[47])

(‘\0’)

(‘\0’)

(‘\0’)

(...)

(...)

(...)

(Nothing is copied to BSS until the loop filling buf finishes.)

You will put the instructions for your attack (to change grade) here
Memory Map of Stack and BSS Section

- **readString's stackframe**
  - SP
  - buf[0]
  - buf[1]
  - ...

- **getName's stackframe**
  - buf[47]
  - ???

- **main's stackframe**

- Now smash the stack like in the 'B' attack!

- (Nothing is copied to BSS until the loop filling buf finishes.)

- You will put the instructions for your attack (to change grade) here

- Replace with BSS address where adr instruction will be put

- Old X30 somewhere
Memory Map of Stack and BSS Section

Now smash the stack like in the ‘B’ attack!

(readString’s stackframe)

…

buf[0]

…

buf[47]

…

getName’s stackframe

…

main’s stackframe

SP

???

(name[0] → ‘B’)

(name[1] → ‘O’)

(… → ‘B’)

(… → ‘O’)

(… → ‘b’)

(name[47] → ‘\0’)

adr x0, grade other instructions go here then enough padding to smash the stack to overwrite:

.getName’s stackframe

…

&name[k]

The address of our adr instruction in BSS (in this example k=4)

(name[0] → ???)

(name[1] → ???)

(name[47] → ???)

(B) attack!

(Nothing is copied to BSS until the loop filling buf finishes.)

How do we write instructions into memory?

Machine language!

_MACHINE language! (Nothing is copied to BSS until the loop filling buf finishes.)
Agenda

A6 “A” Attack
AARCH64 Machine Language
AARCH64 Machine Language after Assembly
AARCH64 Machine Language after Linking

Assembly Language: add x1, x2, x3

Machine Language: 1000 1011 0000 0011 0000 0000 0100 0001
Machine Language: TOY → AARCH64

INSTRUCTION FORMATS  Remember TOY?
ARM is more complex, but the same ideas!

Format RR:  | . . . . . | . . . | . . . | . . | . . | (0–6, A–B)
Format A:   | . . . . . | . . | . . | addr | (7–9, C–F)

AARCH64 machine language

• All instructions are 32 bits long, 4-byte aligned
• Some bits allocated to opcode: what kind of instruction is this?
• Other bits specify register(s)
• Depending on instruction, other bits may be used for an immediate value, a memory offset, an offset to jump to, etc.

Instruction formats

• Variety of ways different instructions are encoded
• We’ll go over quickly in class, to give you a flavor
• Refer to slides as reference for Assignment 6!
  (Every instruction format you’ll need is in the following slides... we think...)
AARCH64 Instruction Format

Operation group
- Encoded in bits 25-28
- x101: Data processing – 3-register
- 100x: Data processing – immediate + register(s)
- 101x: Branch
- x1x0: Load/store
AARCH64 Instruction Format

Op. Group: Data processing – 3-register

- Instruction width in bit 31: 0 = 32-bit, 1 = 64-bit
- Whether to set condition flags (e.g. ADD vs ADDS) in bit 29
- Second source register in bits 16-20
- First source register in bits 5-9
- Destination register in bits 0-4
- Remaining bits encode additional information about instruction
AARCH64 Instruction Format

Example: `add x1, x2, x3`

- opcode = `add`
- Instruction width in bit 31: `1` = 64-bit
- Whether to set condition flags in bit 29: no
- Second source register in bits 16-20: `3`
- First source register in bits 5-9: `2`
- Destination register in bits 0-4: `1`
- Additional information about instruction: none
AARCH64 Instruction Format

Op. Group: Data processing – immediate + register(s)

- Instruction width in bit 31: 0 = 32-bit, 1 = 64-bit
- Whether to set condition flags (e.g. ADD vs ADDS) in bit 29
- Immediate value in bits 10-21 for 2-register instructions, bits 5-20 for 1-register instructions
- Source register in bits 5-9
- Destination register in bits 0-4
- Remaining bits encode additional information about instruction
AARCH64 Instruction Format

Example: `subs w1, w2, 42`
- **opcode**: subtract immediate
- **Instruction width in bit 31**: 0 = 32-bit
- **Whether to set condition flags in bit 29**: yes
- **Immediate value in bits 10-21**: $101010_2 = 42$
- **First source register in bits 5-9**: 2
- **Destination register in bits 0-4**: 1
- **Additional information about instruction**: none
AARCH64 Instruction Format

Example: mov  x1, 42

- opcode: move immediate
- Instruction width in bit 31: 1 = 64-bit
- Immediate value in bits 5-20: 101010b = 42
- Destination register in bits 0-4: 1
AARCH64 Instruction Format


- Relative address of branch target in bits 0-25 for unconditional branch (b) and function call (bl)
- Relative address of branch target in bits 5-23 for conditional branch
- Because all instructions are 32 bits long and are 4-byte aligned, relative addresses end in 00. So, the values in the instruction must be shifted left by 2 bits. This provides more range with fewer bits!
- Type of conditional branch encoded in bits 0-3
AARCH64 Instruction Format

Example: `b someLabel`

- This depends on where `someLabel` is relative to this instruction!
  - For this example, `someLabel` is 3 instructions (12 bytes) earlier
- opcode: unconditional branch
AARCH64 Instruction Format

Example: `bl someLabel`

- This depends on where `someLabel` is relative to this instruction!
  - For this example, `someLabel` is 3 instructions (12 bytes) earlier
- opcode: branch and link (function call)
- Relative address in bits 0-25: two’s complement of \(11_b\).
  - Shift left by 2: \(1100_b = 12\). So, offset is \(-12\).
AARCH64 Instruction Format

Example: `ble someLabel`

- This depends on where `someLabel` is relative to this instruction!
  - For this example, `someLabel` is 3 instructions (12 bytes) later
- opcode: conditional branch
- *Relative* address in bits 5-23: 11\textsubscript{b}. Shift left by 2: 1100\textsubscript{b} = 12
- Conditional branch type in bits 0-3: LE
AARCH64 Instruction Format


• Instruction width in bits 30-31: 00 = 8-bit, 01 = 16-bit, 10 = 32-bit, 11 = 64-bit
• For [Xn,Xm] addressing mode: second source register in bits 16-20
• For [Xn,offset] addressing mode: offset in bits 10-21, shifted left by 3 bits for 64-bit, 2 bits for 32-bit, 1 bit for 16-bit
• First source register in bits 5-9
• Destination register in bits 0-4
• Remaining bits encode additional information about instruction
AARCH64 Instruction Format

Example: `ldr x0, [x1, x2]`

- opcode: load, register+register
- Instruction width in bits 30-31: 11 = 64-bit
- Second source register in bits 16-20: 2
- First source register in bits 5-9: 1
- Destination register in bits 0-4: 0
- Additional information about instruction: no LSL
AARCH64 Instruction Format

Example: `str x0, [sp,24]`

- opcode: store, register+offset
- Instruction width in bits 30-31: $11 = 64$-bit
- Offset value in bits 12-20: $11_b$, shifted left by 3 = $11000_b = 24$
- “Source” (really destination!) register in bits 5-9: $31 = sp$
- “Destination” (really source!) register in bits 0-4: $0$
- Remember that store instructions use the opposite convention from others: “source” and “destination” are flipped!
AARCH64 Instruction Format

Example: `strb x0, [sp,24]`

- opcode: store, register+offset
- Instruction width in bits 30-31: 00 = 8-bit
- Offset value in bits 12-20: 11000\text{b} (don’t shift left!) = 24
- “Source” (really destination!) register in bits 5-9: 31 = sp
- “Destination” (really source!) register in bits 0-4: 0
- Remember that store instructions use the opposite convention from others: “source” and “destination” are flipped!
AARCH64 Instruction Format

ADR instruction
(Distinct from others w/ Op Group bits 100x)

- Specifies \textit{relative} position of label (data location)
- 19 High-order bits of offset in bits 5-23
- 2 Low-order bits of offset in bits 29-30
- Destination register in bits 0-4
AARCH64 Instruction Format

Example: `adr x19, someLabel`

- This depends on where `someLabel` is relative to this instruction!
  - For this example, `someLabel` is 50 bytes later
- Opcode: generate address
- 19 High-order bits of offset in bits 5-23: 1100
- 2 Low-order bits of offset in bits 29-30: 10
- Relative data location is $110010_2 = 50$ bytes after this instruction
- Destination register in bits 0-4: 0001 1001 0011

msb: bit 31

\[0101 \ 0000 \ 0000 \ 0000 \ 0000 \ 0001 \ 1001 \ 0011\]

lsb: bit 0
Agenda

A6 “A” Attack
AARCH64 Machine Language
AARCH64 Machine Language after Assembly
AARCH64 Machine Language after Linking
The Build Process

Covered here

Preprocess

Compile

Assemble

Link

mypgm.c
mypgm.i
mypgm.s
mypgm.o
libc.a
mypgm
An Example Program

A simple (nonsensical) program, in C and assembly:

```c
#include <stdio.h>
int main(void)
{
    printf("Type a char: ");
    if (getchar() == 'A')
        printf("Hi\n");
    return 0;
}
```

Let's consider the machine language equivalent...

```
.section .rodata
msg1: .string "Type a char: 
msg2: .string "Hi\n .section .text
.global main
main:
    sub    sp, sp, 16
    str    x30, [sp]
    adr    x0, msg1
    bl     printf
    bl     getchar
    cmp    w0, 'A'
    bne    skip
    adr    x0, msg2
    bl     printf
    skip:
    mov    w0, 0
    ldr    x30, [sp]
    add    sp, sp, 16
    ret
```
Examining Machine Lang: RODATA

Assemble program; run objdump

```bash
$ gcc217 -c detecta.s
$ objdump --full-contents --section .rodata detecta.o
```

detecta.o: file format elf64-littleaarch64

Contents of section .rodata:

<table>
<thead>
<tr>
<th>Offsets</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>54797065 20612063 6861723a 20004869</td>
</tr>
<tr>
<td>0010</td>
<td>0a00</td>
</tr>
</tbody>
</table>

Type a char: .Hi

• Assembler does not know addresses
• Assembler knows only offsets
• "Type a char: " starts at offset 0x0
• "Hi\n" starts at offset 0xe
Examining Machine Lang: TEXT

```bash
$ objdump --disassemble --reloc detecta.o

detecta.o: file format elf64-littleaarch64

Disassembly of section .text:

0000000000000000 <main>:
0: d10043ff    sub   sp, sp, #0x10
  4: f90003fe    str   x30, [sp]
  8: 10000000    adr   x0, 0 <main>
     R_AARCH64_ADR_PREL_LO21 .rodata
 c: 94000000    bl    0 <printf>
    R_AARCH64_CALL26 printf
10: 94000000    bl    0 <getchar>
   R_AARCH64_CALL26 getchar
14: 7101041f    cmp   w0, #0x41
18: 54000061    b.ne  24 <skip>
1c: 10000000    adr   x0, 0 <main>
   R_AARCH64_ADR_PREL_LO21 .rodata+0xe
20: 94000000    bl    0 <printf>
   R_AARCH64_CALL26 printf
0000000000000024 <skip>:
24: 52800000    mov   w0, #0x0
28: f94003fe    ldr   x30, [sp]
2c: 910043ff    add   sp, sp, #0x10
30: d65f03c0    ret
```

Run objdump to see instructions

Assembly language
$ objdump --disassemble --reloc detecta.o$

detecta.o:  file format elf64-littleaarch64

Disassembly of section .text:

0000000000000000 <main>:

0: d10043ff    sub   sp, sp, #0x10
4: f90003fe    str   x30, [sp]
8: 10000000    adr   x0, 0 <main>
     R_AARCH64_ADR_PREL_LO21    .rodata
0c: 94000000    bl    0 <printf>
     R_AARCH64_CALL26    printf
10: 94000000    bl    0 <getchar>
     R_AARCH64_CALL26    getchar
14: 7101041f    cmp   w0, #0x41
18: 54000061    b.ne  24 <skip>
1c: 10000000    adr   x0, 0 <main>
     R_AARCH64_ADR_PREL_LO21    .rodata+0xe
20: 94000000    bl    0 <printf>
     R_AARCH64_CALL26    printf
0000000000000024 <skip>:

24: 52800000    mov   w0, #0x0
28: f94003fe    ldr   x30, [sp]
2c: 910043ff    add   sp, sp, #0x10
30: d65f03c0    ret

Examining Machine Lang: TEXT

Run objdump to see instructions

Machine language
$ objdump --disassemble --reloc detecta.o$

detecta.o:  file format elf64-littleaarch64

Disassembly of section .text:

0000000000000000 <main>:
  0: d10043ff    sub   sp, sp, #0x10
  4: f90003fe    str   x30, [sp]
  8: 10000000    adr   x0, 0 <main>
         R_AARCH64_ADR_PREL_LO21  .rodata
 c: 94000000    bl    0 <printf>
         R_AARCH64_CALL26  printf
 10: 94000000    bl    0 <getchar>
         R_AARCH64_CALL26  getchar
 14: 7101041f    cmp   w0, #0x41
 18: 54000061    b.ne  24 <skip>
 1c: 10000000    adr   x0, 0 <main>
         R_AARCH64_ADR_PREL_LO21  .rodata+0xe
 20: 94000000    bl    0 <printf>
         R_AARCH64_CALL26  printf

0000000000000024 <skip>:
 24: 52800000    mov   w0, #0x0
 28: f94003fe    ldr   x30, [sp]
 2c: 910043ff    add   sp, sp, #0x10
 30: d65f03c0    ret

Run objdump to see instructions

Offsets

Let’s examine one line at a time...
Disassembly of section .text:

0000000000000000 <main>:
0: d10043ff    sub   sp, sp, #0x10
4: f90003fe    str   x30, [sp]
8: 10000000    adr   x0, 0 <main>
   R_AARCH64_ADR_PREL_LO21 .rodata
 c: 94000000    bl    0 <printf>
   c: R_AARCH64_CALL26 printf
10: 94000000    bl    0 <getchar>
   10: R_AARCH64_CALL26 getchar
14: 7101041f    cmp   w0, #0x41
18: 54000061    b.ne  24 <skip>
1c: 10000000    adr   x0, 0 <main>
   R_AARCH64_ADR_PREL_LO21 .rodata+0xe
20: 94000000    bl    0 <printf>
   20: R_AARCH64_CALL26 printf

0000000000000024 <skip>:
24: 52800000    mov   w0, #0x0
28: f94003fe    ldr   x30, [sp]
2c: 910043ff    add   sp, sp, #0x10
30: d65f03c0    ret
sub sp, sp, #0x10

- **opcode**: subtract immediate
- **Instruction width in bit 31**: 1 = 64-bit
- **Whether to set condition flags in bit 29**: no
- **Immediate value in bits 10-21**: 10000\(_b\) = 0x10 = 16
- **First source register in bits 5-9**: 31 = sp
- **Destination register in bits 0-4**: 31 = sp
- **Additional information about instruction**: none
$ objdump --disassemble --reloc detecta.o

detecta.o:   file format elf64-littleaarch64

Disassembly of section .text:

0000000000000000 <main>:
  0: d10043ff    sub   sp, sp, #0x10
  4: f90003fe    str   x30, [sp]
  8: 10000000    adr   x0, 0 <main>
     : R_AARCH64_ADR_PREL_LO21 .rodata
  c: 94000000    bl    0 <printf>
     : R_AARCH64_CALL26 printf
  10: 94000000   bl    0 <getchar>
     : R_AARCH64_CALL26 getchar
  14: 7101041f    cmp   w0, #0x41
  18: 54000061    b.ne 24 <skip>
  1c: 10000000    adr   x0, 0 <main>
     : R_AARCH64_ADR_PREL_LO21 .rodata+0xe
  20: 94000000   bl    0 <printf>
     : R_AARCH64_CALL26 printf

0000000000000024 <skip>:
  24: 52800000    mov   w0, #0x0
  28: f94003fe    ldr   x30, [sp]
  2c: 910043ff    add   sp, sp, #0x10
  30: d65f03c0    ret
str  x30, [sp]

- opcode: store, register + offset
- Instruction width in bits 30-31: 11 = 64-bit
- Offset value in bits 12-20: 0
- “Source” (really destination) register in bits 5-9: 31 = sp
- “Destination” (really source) register in bits 0-4: 30
- Additional information about instruction: none
adr  x0, 0  <main>

$ objdump --disassemble --reloc detecta.o

detecta.o:  file format elf64-littleaarch64

Disassembly of section .text:

0000000000000000 <main>:
0: d10043ff    sub   sp, sp, #0x10
4: f90003fe    str   x30, [sp]
 8: 10000000  adr   x0, 0 <main>
     R_AARCH64_ADR_PREL_LO21  .rodata
c: 94000000    bl    0 <printf>
   c: R_AARCH64_CALL26  printf
c: 94000000  bl    0 <getchar>
   c: R_AARCH64_CALL26  getchar
10: 94000000  bl    0 <printf>
   R_AARCH64_CALL26  printf
    R_AARCH64_ADR_PREL_LO21    .rodata+0xe
14: 7101041f    cmp   w0, #0x41
18: 54000061  b.ne  24 <skip>
1c: 10000000   adr   x0, 0 <main>
   R_AARCH64_ADR_PREL_LO21    .rodata+0xe
20: 94000000  bl    0 <printf>
   R_AARCH64_CALL26  printf

0000000000000024 <skip>:
24: 52800000  mov   w0, #0x0
28: f94003fe  ldr   x30, [sp]
 2c: 910043ff  add   sp, sp, #0x10
30: d65f03c0  ret
adr  x0, 0  <main>

- opcode: generate address
- 19 High-order bits of relative address in bits 5-23: 0
- 2 Low-order bits of relative address in bits 29-30: 0
- Relative data location is 0 bytes after this instruction
- Destination register in bits 0-4:0

Huh? That's not where msg1 lives!
  - Assembler knew that msg1 is a label within the RODATA section
  - But assembler didn’t know address of RODATA section!
  - So, assembler couldn’t generate this instruction completely, left a placeholder, and will request help from the linker
Examining Machine Lang: TEXT

Run objdump to see instructions

Relocation records
$ objdump --disassemble --reloc detecta.o

detecta.o:  file format elf64-littleaarch64

Disassembly of section .text:

0000000000000000 <main>:
  0: d10043ff    sub   sp, sp, #0x10
  4: f90003fe    str   x30, [sp]
  8: 10000000    adr   x0, 0 <main>
  8: R_AARCH64_ADR_PREL_LO21    .rodata
  c: 94000000    bl    0 <printf>
  c: R_AARCH64_CALL26    printf
  10: 94000000    bl    0 <getchar>
  10: R_AARCH64_CALL26    getchar
  14: 7101041f    cmp   w0, #0x41
  18: 54000061    b.ne  24 <skip>
  1c: 10000000    adr   x0, 0 <main>
  1c: R_AARCH64_ADR_PREL_LO21    .rodata+0xe
  20: 94000000    bl    0 <printf>
  20: R_AARCH64_CALL26    printf

0000000000000024 <skip>:
  24: 52800000    mov   w0, #0x0
  28: f94003fe    ldr   x30, [sp]
  2c: 910043ff    add   sp, sp, #0x10
  30: d65f03c0    ret
Dear Linker,

Please patch the TEXT section at offset 0x8. Patch in a 21-bit* signed offset of an address, relative to the PC, as appropriate for the adr instruction format. When you determine the address of .rodata, use that to compute the offset you need to do the patch.

Sincerely,
Assembler

* 19 High-order bits of relative address in bits 5-23
2 Low-order bits of relative address in bits 29-30
$ objdump --disassemble --reloc detecta.o

detecta.o: file format elf64-littleaarch64

Disassembly of section .text:

0000000000000000 <main>:
   0: d10043ff    sub   sp, sp, #0x10
   4: f90003fe    str   x30, [sp]
   8: 10000000    adr   x0, 0 <main>
      8: R_AARCH64_ADR_PREL_LO21 .rodata
   c: 94000000    bl    0 <printf>
      c: R_AARCH64_CALL26   printf
  10: 94000000    bl    0 <getchar>
     10: R_AARCH64_CALL26   getchar
  14: 7101041f    cmp   w0, #0x41
  18: 54000061    b.ne  24 <skip>
  1c: 10000000    adr   x0, 0 <main>
     1c: R_AARCH64_ADR_PREL_LO21 .rodata+0xe
  20: 94000000    bl    0 <printf>
     20: R_AARCH64_CALL26   printf

0000000000000000 <skip>:
   24: 52800000    mov   w0, #0x0
   28: f94003fe    ldr   x30, [sp]
   2c: 910043ff    add   sp, sp, #0x10
   30: d6503c0    ret
• opcode: branch and link
• Relative address in bits 0-25: 0

• Huh? That’s not where `printf` lives!
  • Assembler had to calculate `[addr of printf] - [addr of this instr]`
  • But assembler didn’t know address of `printf` – it’s off in some library (`libc`) and isn’t present (yet)!
  • So, assembler couldn’t generate this instruction completely, left a placeholder, and will request help from the linker
$ objdump --disassemble --reloc detecta.o

detecta.o:  file format elf64-littleaarch64

Disassembly of section .text:

0000000000000000 <main>:
  0: d10043ff    sub   sp, sp, #0x10
  4: f90003fe    str   x30, [sp]
  8: 10000000    adr   x0, 0 <main>
     8: R_AARCH64_ADR_PREL_LO21    .rodata
   c: 94000000    bl    0 <printf>
     c: R_AARCH64_CALL26    printf
 10: 94000000    bl    0 <getchar>
 10: R_AARCH64_CALL26    getchar
 14: 7101041f    cmp   w0, #0x41
 18: 54000061    b.ne  24 <skip>
 1c: 10000000    adr   x0, 0 <main>
     1c: R_AARCH64_ADR_PREL_LO21 .rodata+0xe
 20: 94000000    bl    0 <printf>
     20: R_AARCH64_CALL26    printf

0000000000000024 <skip>:
 24: 52800000    mov   w0, #0x0
 28: f94003fe    ldr   x30, [sp]
 2c: 910043ff    add   sp, sp, #0x10
 30: d65f03c0    ret
Dear Linker,

Please patch the TEXT section at offset 0xc. Patch in a 26-bit signed offset relative to the PC, appropriate for the function call (bl) instruction format. When you determine the address of printf, use that to compute the offset you need to do the patch.

Sincerely,
Assembler
$ objdump --disassemble --reloc detecta.o

detecta.o: file format elf64-littleaarch64

Disassembly of section .text:

0000000000000000 <main>:
  0: d10043ff    sub   sp, sp, #0x10
  4: f90003fe    str   x30, [sp]
  8: 10000000   adr   x0, 0 <main>
    c: 94000000   bl    0 <printf>
    c: R_AARCH64_CALL26  printf
  10: 94000000  bl    0 <getchar>
      10: R_AARCH64_CALL26  getchar
  14: 7101041f  cmp   w0, #0x41
  18: 54000061  b.ne  24 <skip>
  1c: 10000000  adr   x0, 0 <main>
      1c: R_AARCH64ADR_PREL_LO21 .rodata+0xe
  20: 94000000  bl    0 <printf>
      20: R_AARCH64_CALL26  printf

0000000000000024 <skip>:
  24: 52800000  mov   w0, #0x0
  28: f94003fe  ldr   x30, [sp]
  2c: 910043ff  add   sp, sp, #0x10
  30: d65f03c0  ret
bl 0 <getchar>

- opcode: branch and link
- Relative address in bits 0-25: 0

- Same situation as before – relocation record coming up!
Relocation Record 3

10: R_AARCH64_CALL26  getchar

Dear Linker,

Please patch the TEXT section at offset 0x10. Patch in a 26-bit signed offset relative to the PC, appropriate for the function call (bl) instruction format. When you determine the address of getchar, use that to compute the offset you need to do the patch.

Sincerely,
Assembler
$ objdump --disassemble --reloc detecta.o

detecta.o:   file format elf64-littleaarch64

Disassembly of section .text:

0000000000000000 <main>:
  0: d10043ff  sub   sp, sp, #0x10
  4: f90003fe  str   x30, [sp]
  8: 10000000  adr   x0, 0 <main>
  8: R_AARCH64_ADR_PREL_LO21 .rodata
   c: 94000000  bl    0 <printf>
   c: R_AARCH64_CALL26  printf
 10: 94000000  bl    0 <getchar>
 10: R_AARCH64_CALL26  getchar
 14: 7101041f  cmp   w0, #0x41
 18: 54000061  b.ne  24 <skip>
 1c: 10000000  adr   x0, 0 <main>
 1c: R_AARCH64_ADR_PREL_LO21 .rodata+0xe
 20: 94000000  bl    0 <printf>
 20: R_AARCH64_CALL26  printf

0000000000000024 <skip>:
 24: 52800000  mov   w0, #0x0
 28: f94003fe  ldr   x30, [sp]
 2c: 910043ff  add   sp, sp, #0x10
 30: d65f03c0  ret
Recall that `cmp` is really an assembler alias: this is the same instruction as `subs wzr, w0, 0x41`

- **opcode**: subtract immediate
- **Instruction width in bit 31**: 0 = 32-bit
- **Whether to set condition flags in bit 29**: yes
- **Immediate value in bits 10-21**: `1000001b = 0x41 = ‘A’`
- **First source register in bits 5-9**: 0
- **Destination register in bits 0-4**: 31 = wzr
  - Note that register #31 (11111b) is used to mean either sp or xzr/wzr, depending on the instruction
$ objdump --disassemble --reloc detecta.o

detecta.o: file format elf64-littleaarch64

Disassembly of section .text:

0000000000000000 <main>:
  0: d10043ff    sub   sp, sp, #0x10
  4: f90003fe    str   x30, [sp]
  8: 10000000    adr   x0, 0 <main>
   8: R_AARCH64_ADR_PREL_LO21 .rodata
  c: 94000000    bl    0 <printf>
   c: R_AARCH64_CALL26   printf
  10: 94000000    bl    0 <getchar>
   10: R_AARCH64_CALL26   getchar
  14: 7101041f    cmp   w0, #0x41
  18: 54000061    b.ne  24 <skip>
  1c: 10000000    adr   x0, 0 <main>
   1c: R_AARCH64_ADR_PREL_LO21 .rodata+0xe
  20: 94000000    bl    0 <printf>
   20: R_AARCH64_CALL26   printf

0000000000000024 <skip>:
  24: 52800000    mov   w0, #0x0
  28: f94003fe    ldr   x30, [sp]
  2c: 910043ff    add   sp, sp, #0x10
  30: d65f03c0    ret
This instruction is at offset 0x18, and `skip` is at offset 0x24, which is 0x24 - 0x18 = 0xc = 12 bytes later

- opcode: conditional branch
- Relative address in bits 5-23: $11_b$. Shift left by 2: $1100_b = 12$
- Conditional branch type in bits 0-4: NE

No need for relocation record!
- Assembler had to calculate [addr of skip] - [addr of this instr]
- Assembler did know offsets of skip and this instruction
- So, assembler could generate this instruction completely, and does not need to request help from the linker
$ objdump --disassemble --reloc detecta.o

detecta.o: file format elf64-littleaarch64

Disassembly of section .text:

0000000000000000 <main>:
  0: d10043ff    sub   sp, sp, #0x10
  4: f90003fe    str   x30, [sp]
  8: 10000000    adr   x0, 0 <main>

  8: R_AARCH64_ADR_PREL_LO21 .rodata
    c: 94000000    bl    0 <printf>
    c: R_AARCH64_CALL26   printf
 10: 94000000    bl    0 <getchar>
 10: R_AARCH64_CALL26   getchar
 14: 7101041f    cmp   w0, #0x41
 18: 54000061    b.ne  24 <skip>
 1c: 10000000    adr   x0, 0 <main>
 1c: R_AARCH64_ADR_PREL_LO21 .rodata+0xe

 20: 94000000    bl    0 <printf>
 20: R_AARCH64_CALL26   printf

0000000000000024 <skip>:
  24: 52800000    mov   w0, #0x0
  28: f94003fe    ldr   x30, [sp]
  2c: 910043ff    add   sp, sp, #0x10
  30: d65f03c0    ret
Dear Linker,

Please patch the TEXT section at offset 0x1c. Patch in a 21-bit signed offset of an address, relative to the PC, as appropriate for the adr instruction format. When you determine the address of .rodata, add 0xe and use that to compute the offset you need to do the patch.

Sincerely,
Assembler
$ objdump --disassemble --reloc detecta.o

detecta.o: file format elf64-littleaarch64

Disassembly of section .text:

0000000000000000 <main>:
 0: d10043ff    sub  sp, sp, #0x10
 4: f90003fe    str   x30, [sp]
 8: 10000000   adr  x0, 0 <main>
 8: R_AARCH64_ADR_PREL_LO21   .rodata
 c: 94000000    bl    0 <printf>
 c: R_AARCH64_CALL26   printf
10: 94000000    bl    0 <getchar>
10: R_AARCH64_CALL26   getchar
14: 7101041f    cmp  w0, #0x41
18: 54000061    b.ne  24 <skip>
1c: 10000000   adr  x0, 0 <main>
1c: R_AARCH64_ADR_PREL_LO21   .rodata+0xe
20: 94000000    bl    0 <printf>
20: R_AARCH64_CALL26   printf

0000000000000024 <skip>:
 24: 52800000    mov  w0, #0x0
 28: f94003fe    ldr  x30, [sp]
 2c: 910043ff    add  sp, sp, #0x10
 30: d65f03c0    ret

Another printf, with relocation record…
What does this relocation record mean?

```
20: 94000000    bl    0 <printf>
```

```
20: R_AARCH64_CALL26    printf
```

See context on previous slides with parallel records:
- `bl printf` (#50)
- `bl getchar` (#53)

Dear Linker,

Please patch the TEXT section at offset `0x20`. Patch in a 26-bit signed offset relative to the PC, appropriate for the function call (bl) instruction format. When you determine the address of `printf`, use that to compute the offset you need to do the patch.

Sincerely,
Assembler
Everything Else is Similar...

$ objdump --disassemble --reloc detecta.o

detecta.o:  file format elf64-littleaarch64

Disassembly of section .text:

Disassembly of section .text:

0000000000000000 <main>:
  0: d10043ff    sub    sp, sp, #0x10
  4: f90003fe    str    x30, [sp]
  8: 10000000    adr    x0, 0 <main>
     8: R_AARCH64_ADR_PREL_LO21    .rodata
 c: 94000000    bl    0 <printf>
     c: R_AARCH64_CALL26    printf
 10: 94000000    bl    0 <getchar>
     10: R_AARCH64_CALL26    getchar
 14: 7101041f    cmp    w0, #0x41
 18: 54000061    b.ne   24 <skip>
 1c: 10000000    adr    x0, 0 <main>
     1c: R_AARCH64_ADR_PREL_LO21    .rodata+0xe
 20: 94000000    bl    0 <printf>
     20: R_AARCH64_CALL26    printf

0000000000000024 <skip>:
  24: 52800000    mov    w0, #0x0
  28: f94003fe    ldr    x30, [sp]
  2c: 910043ff    add    sp, sp, #0x10
  30: d65f03c0    ret

Exercise for you:
using information from these slides, create a bitwise breakdown of these instructions, and convince yourself that the hex values are correct!
Agenda

A6 “A” Attack
AARCH64 Machine Language
AARCH64 Machine Language after Assembly
AARCH64 Machine Language after Linking
Assembler writes its data structures to .o file

Linker:
- Reads .o file
- Writes executable binary file
- Works in two phases: **resolution** and **relocation**
Resolution

- Linker resolves references

For our sample program, linker:

- Notes that labels getchar and printf are unresolved
- Fetches machine language code defining getchar and printf from libc.a
- Adds that code to TEXT section
- Adds more code (e.g. definition of _start) to TEXT section too
- Adds code to other sections too
Relocation

- Linker patches ("relocates") code
- Linker traverses relocation records, patching code as specified
Examining Machine Language: RODATA

Link program; run objdump on final executable

$ gcc217 detecta.o -o detecta
$ objdump --full-contents --section .rodata detecta

detecta: file format elf64-littleaarch64

Contents of section .rodata:

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400710</td>
<td>01000200 00000000 00000000 00000000 ...............</td>
</tr>
<tr>
<td>0x400720</td>
<td>54797065 20612063 6861723a 20004869 Type a char: .Hi</td>
</tr>
<tr>
<td>0x400730</td>
<td>0a00</td>
</tr>
</tbody>
</table>

Addresses, not offsets

RODATA is at 0x400710
Starts with some header info
Real start of RODATA is at 0x400720
"Type a char: " starts at 0x400720
"Hi\n" starts at 0x40072e
$ objdump --disassemble --reloc detecta

detecta: file format elf64-littleaarch64
...

00000000000400650 <main>:
  400650: d10043ff    sub   sp, sp, #0x10
  400654: f90003fe    str   x30, [sp]
  400658: 10000640    adr   x0, 400720 <msg1>
  40065c: 97ffffa1    bl    4004e0 <printf@plt>
  400660: 97ffff9c    bl    4004d0 <getchar@plt>
  400664: 7101041f    cmp   w0, #0x41
  400668: 54000061    b.ne  400674 <skip>
  40066c: 50000600    adr   x0, 40072e <msg2>
  400670: 97ffff9c    bl    4004e0 <printf@plt>

00000000000400674 <skip>:
  400674: 52800000    mov   w0, #0x0
  400678: f94003fe    ldr   x30, [sp]
  40067c: 910043ff    add   sp, sp, #0x10
  400680: d65f03c0    ret

Examining Machine Language: TEXT

Addresses, not offsets

Run objdump to see instructions
Examining Machine Language: TEXT

$ objdump --disassemble --reloc detecta

detecta:   file format elf64-littleaarch64

...  
0000000000400650 <main>:
  400650:   d10043ff    sub   sp, sp, #0x10
  400654:   f90003fe    str   x30, [sp]
  400658:   10000640   adr   x0, 400720 <msg1>
  40065c:   97ffffa1    bl    4004e0 <printf@plt>
  400660:   97ffff9c    bl    4004d0 <getchar@plt>
  400664:   7101041f   cmp   w0, #0x41
  400668:   54000061   b.ne   400674 <skip>
  40066c:   50000600   adr   x0, 40072e <msg2>
  400670:   97ffff9c    bl    4004e0 <printf@plt>

0000000000400674 <skip>:
  400674:   52800000   mov   w0, #0x0
  400678:   f94003fe   ldr   x30, [sp]
  40067c:   910043ff   add   sp, sp, #0x10
  400680:   d65f03c0   ret
Examining Machine Language: TEXT

$ objdump --disassemble --reloc detecta

detecta:   file format elf64-littleaarch64

...  
00000000000400650 <main>:
  400650:   d10043ff    sub  sp, sp, #0x10
  400654:   f90003fe    str   x30, [sp]
  400658:   10000640   adr   x0, 400720 <msg1>
  40065c:   97ffffa1    bl   4004e0 <printf@plt>
  400660:   97ffff9c    bl   4004d0 <getchar@plt>
  400664:   7101041f    cmp   w0, #0x41
  400668:   54000061    b.ne  400674 <skip>
  40066c:   50000600   adr   x0, 40072e <msg2>
  400670:   97ffff9c    bl   4004e0 <printf@plt>

00000000000400674 <skip>:
  400674:   52800000    mov   w0, #0x0
  400678:   f94003fe    ldr   x30, [sp]
  40067c:   910043ff    add   sp, sp, #0x10
  400680:   d65f03c0    ret

Let's see what the linker did with them...
$ objdump --disassemble --reloc detecta

detecta: file format elf64-littleaarch64

... 0000000000400650 <main>:  
0000000000400650: d10043ff sub sp, sp, #0x10  
0000000000400654: f90003fe str x30, [sp]  
 0000000000400658: 10000640 adr x0, 400720 <msg1>  
000000000040065c: 97ffffa1 bl 4004e0 <printf@plt>  
0000000000400660: 97ffff9c bl 4004d0 <getchar@plt>  
0000000000400664: 7101041f cmp w0, #0x41  
0000000000400668: 54000061 b.ne 400674 <skip>  
000000000040066c: 97ffff9c bl 4004e0 <printf@plt>  
0000000000400670: 52800000 mov w0, #0x0  
0000000000400674: 50000600 adr x0, 40072e <msg2>  
0000000000400678: f94003fe ldr x30, [sp]  
000000000040067c: 910043ff add sp, sp, #0x10  
0000000000400680: d65f03c0 ret  
0000000000400684 <skip>:  
0000000000400684: 52800000 mov w0, #0x0  
0000000000400688: f94003fe ldr x30, [sp]  
000000000040068c: 910043ff add sp, sp, #0x10  
0000000000400690: d65f03c0 ret  

adr x0, 400720 <msg1>

• opcode: generate address
• 19 High-order bits of offset in bits 5-23: 110010
• 2 Low-order bits of offset in bits 29-30: 00
• Relative data location is $11001000_b = 0xc8$ bytes after this instruction
• Destination register in bits 0-4:0

• msg1 is at 0x400720; this instruction is at 0x400658
• $0x400720 - 0x400658 = 0xc8$ ✓
$ objdump --disassemble --reloc detecta

detecta: file format elf64-littleaarch64

... 00000000000400650 <main>:
400650:   d10043ff    sub   sp, sp, #0x10
400654:   f90003fe    str   x30, [sp]
400658:   10000640    adr   x0, 400720 <msg1>
40065c:   97ffffa1    bl    4004e0 <printf@plt>
400660:   97ffff9c    bl    4004d0 <getchar@plt>
400664:   7101041f    cmp   w0, #0x41
400668:   54000061    b.ne  400674 <skip>
40066c:   50000600    adr   x0, 40072e <msg2>
400670:   97ffff9c    bl    4004e0 <printf@plt>

00000000000400674 <skip>:
400674:   52800000    mov   w0, #0x0
400678:   f94003fe    ldr   x30, [sp]
40067c:   910043ff    add   sp, sp, #0x10
400680:   d65f03c0    ret
• opcode: branch and link
• *Relative* address in bits 0-25: 26-bit two's complement of $1011111_b$. But remember to shift left by two bits (see earlier slides)! This gives $-10111100_b = -0x17c$

• *printf* is at 0x4004e0; this instruction is at 0x40065c
• $0x4004e0 - 0x40065c = -0x17c$ ✔
Everything Else is Similar...

$ objdump --disassemble --reloc detecta

detecta: file format elf64-littleaarch64

...  

0000000000400650 <main>:
  400650: d10043ff    sub   sp, sp, #0x10    
  400654: f90003fe    str   x30, [sp]    
  400658: 10000640   adr   x0, 400720 <msg1>   
  40065c: 97ffffa1    bl   4004e0 <printf@plt>   
  400660: 97ffff9c    bl   4004d0 <getchar@plt>   
  400664: 7101041f   cmp   w0, #0x41   
  400668: 54000060    b.ne 400674 <skip>   
  40066c: 50000600   adr   x0, 40072e <msg2>   
  400670: 97ffff9c    bl   4004e0 <printf@plt>   

0000000000400674 <skip>:
  400674: 52800000    mov   w0, #0x0   
  400678: f94003fe   ldr   x30, [sp]   
  40067c: 910043ff    add   sp, sp, #0x10   
  400680: d65f03c0    ret
Summary

AARCH64 Machine Language

• 32-bit instructions
• Formats have conventional locations for opcodes, registers, etc.

Assembler

• Reads assembly language file
• Generates TEXT, RODATA, DATA, BSS sections
  • Containing machine language code
• Generates relocation records
• Writes object (.o) file

Linker

• Reads object (.o) file(s)
• Does resolution: resolves references to make code complete
• Does relocation: traverses relocation records to patch code
• Writes executable binary file
Wrapping Up the Course

Precepts all done!

Assignment 5 due on Thursday (4/21) at 9:00 PM

Assignment 6

- Partnered assignment
- Due on Dean’s Date (Tuesday, May 3) at 5 PM ET
- Extensions past 11:59 PM require permission of the Dean

Extensive office hours during reading period and exam period

- Exact schedule will be announced on Ed

Final exam: Tuesday, 5/10, 9:00 AM – 12:00 PM ET, in Friend 101

- Closed-book, closed-notes, 1-page study sheet, no electronics
- Details and old exams are available at https://www.cs.princeton.edu/courses/archive/spr22/cos217/exam2.html
- Review and Q&A sessions scheduled after Dean's date – announced on Ed
We Have Covered:

<table>
<thead>
<tr>
<th>Programming in the large</th>
<th>Programming at several levels</th>
<th>Core systems and organization ideas</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Program design</td>
<td>• The C programming language</td>
<td>• Storage hierarchy</td>
</tr>
<tr>
<td>• Programming style</td>
<td>• ARM Assembly Language</td>
<td>• Compile, Assemble, Link</td>
</tr>
<tr>
<td>• Building</td>
<td>• ARM Machine Language</td>
<td>• (just a taste of) Processes and VM</td>
</tr>
<tr>
<td>• Testing</td>
<td>• (just a taste of) the bash shell</td>
<td></td>
</tr>
<tr>
<td>• Debugging</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Data structures</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Modularity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Performance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Version control</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The end.

return EXIT_SUCCESS;