Machine Language
Instruction Set Architecture (ISA)

There are many kinds of computer chips out there:

- ARM
- 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*
This lecture is about

- machine language (in general)
- AARCH64 machine language (in particular)
- The assembly and linking processes
- Amusing and important applications to computer security
  (and therefore, Programming Assignment 5, Buffer Overrun)
The Build Process

Preprocess
mympgm.c → mympgm.i

Compile
mympgm.i → mympgm.s

Assemble
mympgm.s → mympgm.o

Link
mympgm.o → mympgm
libc.a

Covered in COS 320: Compiling Techniques

Covered here
Agenda

AARCH64 Machine Language
AARCH64 Machine Language after Assembly
AARCH64 Machine Language after Linking
Buffer overrun vulnerabilities

Assembly Language: \texttt{add x1, x2, x3}

Machine Language: \texttt{1000 1011 0000 0011 0000 0000 0100 0001}
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 5!
  (Every instruction format you’ll need is in the following slides… we think…)
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

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

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
Example: \texttt{subs w1, w2, 42}

- \textit{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: \texttt{mov x1, 42}

- opcode: move immediate
- Instruction width in bit 31: 1 = 64-bit
- Immediate value in bits 5-20: \(101010_b = 42\)
- Destination register in bits 0-4: 1
Branch

- *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: `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 0-4: LE
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
- Relative address in bits 0-25: two’s complement of 11\textsubscript{b}. Shift left by 2: 1100\textsubscript{b} = 12. So, offset is -12.
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 $11_2$. Shift left by 2: $1100_2 = 12$. So, offset is $-12$. 

1001 0111 1111 1111 1111 1111 1111 1101

msb: bit 31

lsb: bit 0
AARCH64 Instruction Format

Load / store

- 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
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 every other instruction: “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\(_b\) (not shifted 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 every other instruction: “source” and “destination” are flipped!
AARCH64 Instruction Format

ADR instruction

- Specifies *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 0-4:19
Agenda

AARCH64 Machine Language

AARCH64 Machine Language after Assembly

AARCH64 Machine Language after Linking

Buffer overrun vulnerabilities
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…

```assembly
.section .rodata
msg1: .string "Type a char: 
msg2: .string "Hi

.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

```
$ 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>

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

Run objdump to see instructions

Offsets
Run objdump to see instructions

<table>
<thead>
<tr>
<th>Byte Address</th>
<th>Machine Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000000000</td>
<td>d10043ff</td>
<td>sub           sp, sp, #0x10</td>
</tr>
<tr>
<td>0000000000000004</td>
<td>f90003fe</td>
<td>str           x30, [sp]</td>
</tr>
<tr>
<td>0000000000000008</td>
<td>10000000</td>
<td>adr           x0, 0 &lt;main&gt;</td>
</tr>
<tr>
<td>000000000000000c</td>
<td>94000000</td>
<td>bl            0 &lt;printf&gt;</td>
</tr>
<tr>
<td>0000000000000010</td>
<td>94000000</td>
<td>bl            0 &lt;getchar&gt;</td>
</tr>
<tr>
<td>0000000000000014</td>
<td>7101041f</td>
<td>cmp           w0, #0x41</td>
</tr>
<tr>
<td>0000000000000018</td>
<td>540000061</td>
<td>b.ne          24 &lt;skip&gt;</td>
</tr>
<tr>
<td>000000000000001c</td>
<td>10000000</td>
<td>adr           x0, 0 &lt;main&gt;</td>
</tr>
<tr>
<td>0000000000000020</td>
<td>94000000</td>
<td>bl            0 &lt;printf&gt;</td>
</tr>
<tr>
<td>0000000000000024</td>
<td>52800000</td>
<td>mov           w0, #0x0</td>
</tr>
<tr>
<td>0000000000000028</td>
<td>f940003fe</td>
<td>ldr           x30, [sp]</td>
</tr>
<tr>
<td>000000000000002c</td>
<td>910043ff</td>
<td>add           sp, sp, #0x10</td>
</tr>
<tr>
<td>0000000000000030</td>
<td>d65f03c0</td>
<td>ret</td>
</tr>
</tbody>
</table>
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>
 c:   94000000    bl    0 <printf>
10:  94000000    bl    0 <getchar>
14:  7101041f    cmp   w0, #0x41
18:  54000061    b.ne  24 <skip>
1c:  10000000    adr   x0, 0 <main>
20:  94000000    bl    0 <printf>
24:  52800000    mov   w0, #0x0
28:  f94003fe    ldr   x30, [sp]
2c:  910043ff    add   sp, sp, #0x10
30:  d65f03c0    ret
```

Assembly language

Run objdump to see instructions
Examining Machine Lang: TEXT

$ 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

Run objdump to see instructions
Relocation records
Let’s examine one line at a time…
$ 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
sub sp, sp, #0x10

msb: bit 31

0: d10043ff sub sp, sp, #0x10

lsb: bit 0

1101 0001 0000 0000 0100 0011 1111 1111
The instruction `sub sp, sp, #0x10` is an example of a subtract immediate instruction. Here are the details:

- **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

The hexadecimal representation of the instruction is `d10043ff`, which can be broken down as follows:

- **msb:** bit 31
- **lsb:** bit 0

It's important to note that the instruction subtracts the immediate value from the value of `sp` and places the result back into `sp`. This is a common operation in many programming languages and assembly languages to manipulate the stack pointer.
str x30, [sp]

```
$ 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
```
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


```plaintext
$ 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
```
**adr x0, 0 <main>**

- **Opcode:** generate address
- **19 High-order bits of offset in bits 5-23:** 0
- **2 Low-order bits of offset 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
$ 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 instruction format. When you determine the address of .rodata, 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
```
bl 0 <printf>

- 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.a) 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>
     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
bl 0 <getchar>

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

- Same situation as before – relocation record coming up!
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

cmp     w0, #0x41
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 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 <getline>
  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
b.ne  24 <skip>

This instruction is at address 0x18, and skip is at address 0x24, which is 0x24 – 0x18 = 0xc = 12 bytes later.

- Opcode: conditional branch
- Relative address in bits 5-23: 11b. Shift left by 2: 1100b = 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 address of skip
- 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 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
Another printf, with relocation record…

```
$ 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
```
mov w0, #0x0

$ 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

53
mov   w0, #0x0

- opcode: move immediate
- Instruction width in bit 31: 0 = 32-bit
- Immediate value in bits 5-20: 0
- Destination register in bits 0-4: 0
Everything Else is Similar…

```
$ 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

0000000000000000024 <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

AARCH64 Machine Language
AARCH64 Machine Language after Assembly
AARCH64 Machine Language after Linking
Buffer overrun vulnerabilities
From Assembler to Linker

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 this 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
Linker Relocation

**Relocation**

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

Link program; run objdump

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

detecta:   file format elf64-littleaarch64

Contents of section .rodata:
000710 01000200 00000000 00000000 00000000 .................
000720 54797065 20612063 6861723a 20004869 Type a char: .Hi
000730 0a00 

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
Run objdump to see instructions

Addresses, not offsets
Examining Machine Lang: 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:   7100041f    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 Lang: 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: f940003fe    ldr   x30, [sp]
  40067c: 910043ff    add   sp, sp, #0x10
  400680: d65f03c0    ret

No relocation records!
Let’s see what the linker did with them…
$ objdump --disassemble --reloc detecta

detecta: file format elf64-littleaarch64

...
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 11001000b = 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

...  

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
bl 4004e0 <printf@plt>

- **opcode**: branch and link
- *Relative* address in bits 0-25: 26-bit two’s complement of 1011111\textsubscript{b}. But remember to shift left by two bits (see earlier slides)! This gives –1 0111 1100\textsubscript{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

... 0000000000000400650 <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: 51300000 mov w0, #0x0  
   000000000040066c: f94003fe ldr x30, [sp]  
   0000000000400670: 910043ff add sp, sp, #0x10  
   0000000000400674: d65f03c0 ret

Everything Else is Similar…
```

68
Agenda

AARCH64 Machine Language
AARCH64 Machine Language after Assembly
AARCH64 Machine Language after Linking

Buffer overrun vulnerabilities
#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;
}

$ ./a.out

What is your name?

John Smith

Thank you, John Smith.

The answer to life, the universe, and everything is 42
Why People With Long Names Have Problems

#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;
}

$ ./a.out
What is your name?
Szymon Rusinkiewicz
Thank you, Szymon Rusinkiewicz.
The answer to life, the universe, and everything is 8020841
Explanation: Stack Frame Layout

When there are too many characters, program carelessly writes beyond space “belonging” to name.

- Overwrites other variables
- This is a buffer overrun, or stack smash
- The program has a security bug!

```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;
}
```
#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;
}
It Gets Worse…

Buffer overrun can overwrite return address of a previous stack frame!

- Value can be an invalid address, leading to a segfault,…

```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;
}
```
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 point to malicious code.

```
#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;
}
```

```
Or here...
```

```
Malicious code here...
```

```
Return addr
name ...

.c

.magic

.i

.Old SP

.SP

0
```

-.bss

Buffer overrun can overwrite return address of a previous stack frame!
Attacking a Web Server

URLs
Input in web forms
Crypto keys for SSL
etc.

for(i=0;p[i];i++)
search[i]=p[i];

Client PC
Web Server

this is a really long search term that overflows a buffer
Attacking a Web Browser

HTML keywords
Images
Image names
URLs
etc.

Client PC

Web Server @ badguy.com

for(i=0;p[i];i++)
gif[i]=p[i];

Earn $$$ Thousands working at home!
Attacking Everything in Sight

for(i=0;p[i];i++)
gif[i]=p[i];

Client PC

The Internet
@ badguy.com

E-mail client
PDF viewer
Operating-system kernel
TCP/IP stack

Any application that ever sees input directly from the outside
Defenses Against This Attack

**Best:** program in languages that make array-out-of-bounds impossible (Java, C#, ML, python, ....)

If you must program in C: use discipline and software analysis tools to check bounds of array subscripts

Otherwise, stopgap security patches:
- Operating system randomizes initial stack pointer
- “No-execute” memory permission
- “Canaries” at end of stack frames

None of these would have prevented the “Heartbleed” attack
Asgt. 5: Attack the “Grader” Program

```c
enum {BUFSIZE = 48};
char grade = 'D';
char name[BUFSIZE];

/* Read a string into name */
void readString() {
    char buf[BUFSIZE];
    int i = 0;  int c;

    /* Read string into buf[] */
    for (;;) {
        c = fgetc(stdin);
        if (c == EOF || c == '\n')
            break;
        buf[i] = c;
        i++;
    }
    buf[i] = '\0';

    /* Copy buf[] to name[] */
    for (i = 0; i < BUFSIZE; i++)
        name[i] = buf[i];
}

/* Prompt for name and read it */
void getName() {
    printf("What is your name?\n");
    readString();
}
```

Unchecked write to buffer!
Asgt. 5: Attack the “Grader” Program

```
int main(void) {
    getname();
    if (strcmp(name, "Andrew Appel") == 0)
        grade = 'B';
    printf("%c is your grade.\n", grade);
    printf("Thank you, %s.\n", name);
    return 0;
}

$ ./grader
What is your name?
Bob
D is your grade.
Thank you, Bob.
$ ./grader
What is your name?
Andrew Appel
B is your grade.
Thank you, Andrew Appel.
```
int main(void) {
    getname();
    if (strcmp(name, "Andrew Appel") == 0)
        grade = 'B';
    printf("%c is your grade.\n", grade);
    printf("Thank you, %s.\n", name);
    return 0;
}

$ ./grader
What is your name?
Bob\0(#@&$%*#&(*^!@%*!!(&#$%(@ *
B is your grade.
Thank you, Bob.

$ ./grader
What is your name?
Susan\0?!*!?????????!%?!!(^%*(^/*^?
A is your grade.
Thank you, Susan.
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