CS 31: Intro to Systems
ISAs and Assembly

Kevin Webb
Swarthmore College
September 20, 2022
Reading Quiz
Overview

• How to directly interact with hardware

• Instruction set architecture (ISA)
  • Interface between programmer and CPU
  • Established instruction format (assembly lang)

• Assembly programming (ARM64)
Abstraction

User / Programmer
Wants low complexity

Applications
Specific functionality

Software library
Reusable functionality

Operating system
Manage resources

Complex devices
Compute & I/O
Abstraction

Applications
Specific functionality

Operating system
Manage resources

Complex devices
Compute & I/O

This week: Machine Interface

Last week: Circuits, Hardware Implementation
Compilation Steps (.c to a.out)

- **C program (p1.c)**
  - Compiler (**gcc**)
  - Executable code (**a.out**)

**Text**

- Normally compile to **a.out** in a single step: `gcc p1.c`

**Executable**

- Reality is more complex: there are intermediate steps!
Compilation Steps (.c to a.out)

You can see the results of intermediate compilation steps using different gcc flags

CS75

C program (p1.c)

Compiler (gcc -S)

Assembly program (p1.s)

Executable code (a.out)
Assembly Code

Human-readable form of CPU instructions

• Almost a 1-to-1 mapping to Machine Code
• Hides some details:
  • Registers have names rather than numbers
  • Instructions have names rather than variable-size codes

We’re going to use ARM64 assembly

• CS lab machines are x86 (x86-64)

• We have a small cluster of ARM64 machines for CS 31
  • On that cluster, can compile C to ARM64 assembly:
    
gcc -S code.c  # open code.s in editor to view
Compilation Steps (.c to a.out)

You can see the results of intermediate compilation steps using different gcc flags.

- **C program** (`p1.c`)
  - Compiler (`gcc -S`)
  - **Assembly program** (`p1.s`)
    - Assembler (`gcc -c (or as)``)
  - **Object code** (`p1.o`)
    - Linker (`gcc (or ld)`)
      - **Executable code** (`a.out`)
Object / Executable / Machine Code

<table>
<thead>
<tr>
<th>Assembly</th>
<th>Machine Code (Hexadecimal for readability)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub   sp, sp, #0x10</td>
<td>d1 00 43 ff</td>
</tr>
<tr>
<td>mov   w0, #0xa</td>
<td>52 80 01 40</td>
</tr>
<tr>
<td>str   w0, [sp, #12]</td>
<td>b9 00 0f e0</td>
</tr>
<tr>
<td>mov   w0, #0x14</td>
<td>52 80 02 80</td>
</tr>
<tr>
<td>str   w0, [sp, #8]</td>
<td>b9 00 0b e0</td>
</tr>
<tr>
<td>ldr   w1, [sp, #12]</td>
<td>b9 40 0f e1</td>
</tr>
<tr>
<td>ldr   w0, [sp, #8]</td>
<td>b9 40 0b e0</td>
</tr>
<tr>
<td>add   w0, w1, w0</td>
<td>0b 00 00 20</td>
</tr>
<tr>
<td>str   w0, [sp, #12]</td>
<td>b9 00 0f e0</td>
</tr>
<tr>
<td>ldr   w0, [sp, #12]</td>
<td>b9 40 0f e0</td>
</tr>
<tr>
<td>add   sp, sp, #0x10</td>
<td>91 00 43 ff</td>
</tr>
<tr>
<td>ret</td>
<td>d6 5f 03 c0</td>
</tr>
</tbody>
</table>
int main(void) {
    int a = 10;
    int b = 20;
    a = a + b;
    return a;
}
Compilation Steps (.c to a.out)

text

- C program (p1.c)

Compiler (gcc -S)

- Assembly program (p1.s)

Assembler (gcc -c (or as))

binary

- Object code (p1.o)

Linker (gcc (or ld))

Other object files (p2.o, p3.o, ...)

Library obj. code (libc.a)

executable binary

- Executable code (a.out)

High-level language

Interface for speaking to CPU

CPU-specific format (011010...)

Other object files

Library obj. code
Instruction Set Architecture (ISA)

• ISA (or simply architecture):
  Interface between lowest software level and the hardware.

• Defines specification of the language for controlling CPU state:
  • Provides a set of instructions
  • Makes CPU registers available
  • Allows access to main memory
  • Exports control flow (change what executes next)
Instruction Set Architecture (ISA)

• The agreed-upon interface between all software that runs on the machine and the hardware that executes it.
ISA Examples

- Intel IA-32 (80x86)
- ARM
- MIPS
- PowerPC
- IBM Cell
- Motorola 68k
- Intel IA-64 (Itanium)
- VAX
- SPARC
- Alpha
- IBM 360
How many of these ISAs have you used?  (Don’t worry if you’re not sure. Try to guess based on the types of CPUs/devices you interact with.)

- Intel IA-32 (80x86)
- ARM
- MIPS
- PowerPC
- IBM Cell
- Motorola 68k
- Intel IA-64 (Itanium)
- VAX
- SPARC
- Alpha
- IBM 360

A. 0  
B. 1-2  
C. 3-4  
D. 5-6  
E. 7+
ISA Characteristics

• Above ISA: High-level language (C, Python, ...)
  • Hides ISA from users
  • Allows a program to run on any machine
    (after translation by human and/or compiler)

• Below ISA: Hardware implementing ISA can change (faster, smaller, ...)
  • ISA is like a CPU “family”
ISA Characteristics

• Above ISA: High-level language (C, Python, ...)
  • Hides ISA from users
  • Allows a program to run on any machine
    (after translation by human and/or compiler)

• Below ISA: Hardware implementing ISA can change (faster, smaller, ...)
  • ISA is like a CPU “family”
**Instruction Translation**

**sum.c (High-level C)**

```c
int sum(int x, int y) {
    int res;
    res = x+y;
    return res;
}
```

**sum.s from sum.c:**

```bash
gcc -S sum.c
```

**Instructions to set up the stack frame and get argument values**

**sum.s (Assembly)**

```assembly
sum:
    sub    sp, sp, #0x20
    str    w0, [sp, #12]
    str    w1, [sp, #8]
    ldr    w1, [sp, #12]
    ldr    w0, [sp, #8]
    add    w0, w1, w0
    str    w0, [sp, #28]
    ldr    w0, [sp, #28]
    add    sp, sp, #0x20
    ret
```

**Instructions to return from function**

**An add instruction to compute sum**
ISA Design Questions

sum.c (High-level C)

```c
int sum(int x, int y) {
    int res;
    res = x+y;
    return res;
}
```

sum.s from sum.c:

```
gcc -S sum.c
```

sum.s (Assembly)

```
sum:
    sub   sp, sp, #0x20
    str   w0, [sp, #12]
    str   w1, [sp, #8]
    ldr   w1, [sp, #12]
    ldr   w0, [sp, #8]
    add   w0, w1, w0
    str   w0, [sp, #28]
    ldr   w0, [sp, #28]
    add   sp, sp, #0x20
    ret
```

What should these instructions do?

What is/isn’t allowed by hardware?

How complex should they be?

Example: supporting multiplication.
C statement: A = A*B

Simple instructions:

LOAD REG1, A
LOAD REG2, B
PROD REG1, REG2
STORE A, REG1

Powerful instructions:

MULT A, B

Translation:
Load the values ‘A’ and ‘B’ from memory into registers, compute the product, store the result in memory where ‘A’ was.
Which would you use if you were designing an ISA for your CPU? (Why?)

Simple instructions:
- LOAD REG1, A
- LOAD REG2, B
- PROD REG1, REG2
- STORE A, REG1

Powerful instructions:
- MULT A, B

A. Simple
B. Powerful
C. Something else
RISC versus CISC (Historically)

• Complex Instruction Set Computing (CISC)
  • Large, rich instruction set
  • More complicated instructions built into hardware
  • Multiple clock cycles per instruction
  • Easier for humans to reason about

• Reduced Instruction Set Computing (RISC)
  • Small, highly optimized set of instructions
  • Memory accesses are specific instructions
  • One instruction per clock cycle
  • Compiler: more work, more potential optimization
So . . . Which System “Won”?  

• Most ISAs (after mid/late 1980’s) are RISC

• The ubiquitous Intel x86 is CISC
  • Tablets and smartphones (ARM) taking over?

• x86 breaks down CISC assembly into multiple, RISC-like, machine language instructions

• Distinction between RISC and CISC is less clear
  • Some RISC instruction sets have more instructions than some CISC sets
ISA Examples

• Intel IA-32 (CISC)
• ARM (RISC)
• MIPS (RISC)
• PowerPC (RISC)
• IBM Cell (RISC)
• Motorola 68k (CISC)

• Intel IA-64 (Neither)
• VAX (CISC)
• SPARC (RISC)
• Alpha (RISC)
• IBM 360 (CISC)
ISA Characteristics

• Above ISA: High-level language (C, Python, ...)
  • Hides ISA from users
  • Allows a program to run on any machine
    (after translation by human and/or compiler)

• Below ISA: Hardware implementing ISA can change (faster, smaller, ...)
  • ISA is like a CPU “family”
Intel x86 Family (IA-32)

Intel i386 (1985)
- 12 MHz - 40 MHz
- ~300,000 transistors
- Component size: 1.5 μm

Intel Core i9 12900k (late 2021)
- ~4,000 MHz - 5,000 MHz
- ~3,000,000,000 transistors
- Component size: ~7 nm

Everything in this family uses the same ISA (Same instructions)!
This semester... ARM!

- ARM is less complex than x86
- ARM is everywhere (e.g., smart phones)
- Specifically, we'll be using AArch64 (64-bit ARM, ARM64)
Processor State in Registers (ARM64)

• Working memory for currently executing program

• Address of next instruction to execute (PC)

• Status of recent ALU tests:
  • N: result is negative
  • Z: result is zero
  • C (carry): unsigned overflow
  • V: signed overflow

<table>
<thead>
<tr>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>...</th>
<th>x28</th>
</tr>
</thead>
<tbody>
<tr>
<td>x29</td>
<td>x30</td>
<td>x31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>zr</td>
<td>pc</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

General-purpose registers

Zero Register (always 0)

Program Counter (PC)

Condition codes (aka flags)
ARM64 Register Conventions

• Even though x0 - x31 are general-purpose, some are unofficially reserved for specific purposes...

• x29 - x31 used to keep track of function / stack information (more on this later)
Component registers

- x0 - x31 are 64-bit registers
- Sometimes, you might only want to store 32 bits (e.g., int variable)
- You can access the lower 32 bits of a register with a prefix of w rather than x (e.g., w0, w1, ..., w28, w29)
- When accessed this way, the upper bits will always be 0

General-purpose registers

- Frame Pointer
- Link Register
- Stack Pointer
- Zero Register (always 0)
- Program Counter (PC)

Condition codes (aka flags)
Assembly Programmer’s View of State

CPU

<table>
<thead>
<tr>
<th>Registers</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>value</td>
<td></td>
</tr>
<tr>
<td>x0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pc</td>
<td>next instr addr</td>
<td></td>
</tr>
<tr>
<td>N, C, Z, V</td>
<td>cond. codes</td>
<td></td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x00000001</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
| 0xffffffff | Program:
\begin{itemize}
  \item data
  \item instrs
  \item stack
\end{itemize}|

Memory:

- Byte addressable array
- Program code and data
- Execution stack
Types of IA32 Instructions

- Data movement (move values between registers or memory)
  - Move (mov): move data from one register to another
  - Load (ldr): move data from memory to register
  - Store (str): move data from register to memory
Data Movement

Move values between memory and registers or between two registers.

Program Counter (PC): Memory address of next instr

Instruction Register (IR): Instruction contents (bits)

Data in
WE
Data in
WE
Data in
WE
Data in
WE

64-bit Register #0
64-bit Register #1
64-bit Register #2
64-bit Register #3

... Register File

MUX
MUX
MUX

ALU

(Memory)
Types of IA32 Instructions

• Data movement (move values between registers or memory)

• Arithmetic (use ALU to compute a value)
  • addition (add)
  • subtract (sub)
  • Many more...
Arithmetic

Use ALU to compute a value, store result in a register.

Program Counter (PC): Memory address of next instr

Instruction Register (IR): Instruction contents (bits)

Register File

64-bit Register #0
64-bit Register #1
64-bit Register #2
64-bit Register #3

MUX

MUX

ALU

Data in
Data in
Data in
Data in

WE
WE
WE
WE

0:
1:
2:
3:
4:
...
N-1:

(Memory)
Types of IA32 Instructions

• Data movement (move values between registers or memory)

• Arithmetic (use ALU to compute a value)

• Control (change PC based on ALU condition code state)
  • branch (b): change PC to value
  • branch if equal (b. eq): change PC if condition codes indicate equality
  • branch if less than or equal (b. le): same as above, but for less than or equal
Control

Change PC based on ALU condition code state.

Program Counter (PC): Memory address of next instr

Instruction Register (IR): Instruction contents (bits)

64-bit Register #0
MUX
ALU
64-bit Register #1
MUX

64-bit Register #2

64-bit Register #3

...
Types of IA32 Instructions

• Data movement (move values between registers or memory)

• Arithmetic (use ALU to compute a value)

• Control (change PC based on ALU condition code state)

• Stack / Function call (We’ll cover these in detail later)
  • Shortcut instructions for common operations
Addressing Modes

• Instructions need to be told where to get operands or store results

• Variety of options for how to address those locations
Addressing Mode: Register

• Instructions can refer to the name of a register

• Example:
  • mov x4, x15  (Copy the contents of x15 into x4 -- overwrites x4, no change to x15)
Addressing Mode: Immediate

• Also known as "Literal" or "Constant" mode

• Allows programmer to hard-code a number

• Can be either decimal (# prefix) or hexadecimal (#0x prefix)

• Examples:
  • mov x0, #42  (Move the decimal constant 42 into register x0)
  • add x1, x3, #0x10  (Add hex 0x10 to contents of x3, store result in x1)
Addressing Mode: Memory

• Access the contents of memory by using a memory address contained in a register.

• Can only be used for the load and store family of instructions
  • load: data moves from memory into register (load from memory)
  • store: data moves from register into memory (store to memory)
  • other instructions cannot access memory directly (e.g, add)
Addressing Mode: Memory

• Access the contents of memory by using a memory address contained in a register.

• Several different forms for accessing memory

1. access address in register: [register]

• Examples:
  • ldr x1, [x7] (Access memory at the address stored in register x7, load data there into x1)
  • str x20, [x2] (Store the contents of register x20 at the memory address stored in x2)
Addressing Mode: Memory

• `ldr x1, [x7]` (Access memory at the address stored in register x7, load data there into x1)
Addressing Mode: Memory

• `ldr x1, [x7]` (Access memory at the address stored in register x7, load data there into x1)

1. Index into memory using the address in x7.
Addressing Mode: Memory

- `ldr x1, [x7]` (Access memory at the address stored in register x7, load data there into x1)

**CPU Registers**

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>42</td>
</tr>
<tr>
<td>x7</td>
<td>0x1A68</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Addressing Mode: Memory

2. access address in register + immediate: [register, #constant]

3. access address in register + register: [register, register]

• Examples:
  • `ldr x3, [sp, #8]` (Take the value in sp (x31), add 8 to it, and treat the sum as a memory address. Load the data found at that memory address into register x3.)
  
  • `str x2, [x1, x7]` (Take the value in x1, add the value stored in x7 to it, and treat the sum as a memory address. Load the data found at that memory address into register x2.)
Addressing Mode: Memory

- `ldr x3, [sp, #8]` (Take the value in `sp` (x31), add 8 to it, and treat the sum as a memory address. Load the data found at that memory address into register x3.)
Addressing Mode: Memory

• `ldr x3, [sp, #8]` (Take the value in sp (x31), add 8 to it, and treat the sum as a memory address. Load the data found at that memory address into register x3.)

```plaintext
<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>sp (x31)</td>
<td>0x1A64</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
```

CPU Registers

1. Compute address: `0xA64 + 8 => 0xA6C`
Addressing Mode: Memory

- `ldr x3, [sp, #8]` (Take the value in `sp` (x31), add 8 to it, and treat the sum as a memory address. Load the data found at that memory address into register x3.)

**CPU Registers**

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td><code>sp</code> (x31)</td>
<td>0x1A64</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

1. Compute address:  
0x1A64 + 8 => 0x1A6C
Addressing Mode: Memory

- `ldr x3, [sp, #8]` (Take the value in `sp` (x31), add 8 to it, and treat the sum as a memory address. Load the data found at that memory address into register x3.)

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3</td>
<td>38</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>sp (x31)</td>
<td>0x1A64</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

1. Compute address: 0x1A64 + 8 => 0x1A6C
2. Copy value at that address to x3.
Other ways of accessing memory

• Other memory forms, see table at bottom of book section 9.1.

• Other instructions that load / store two registers at once, see table 2 in book section 9.2.

• In general, I'll expect you to be able to read / understand those instructions with the help of the cheat sheet. You don't need to generate them on your own though. They often help with compiler optimizations.
Let’s try a few examples...
What will the machine state be after executing these instructions?

(Bonus: write an equivalent one-line expression in C code)

```
ldr x0, [sp, #8]
ldr x1, [sp, #16]
lsl x1, x1, #3
mul x1, x0, x1
ldr x0, [sp]
add x1, x0, x1
str x1, [sp, #16]
```

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Value</th>
<th>C Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1260</td>
<td>2</td>
<td>x</td>
</tr>
<tr>
<td>0x1268</td>
<td>3</td>
<td>y</td>
</tr>
<tr>
<td>0x1270</td>
<td>2</td>
<td>z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>x0</td>
</tr>
<tr>
<td>x1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>x31 (sp)</td>
</tr>
</tbody>
</table>
What will the machine state be after executing these instructions?  
(Bonus: write an equivalent one-line expression in C code)

```
ldr x0, [sp, #8]
ldr x1, [sp, #16]
lsr x1, x1, #3
mul x1, x0, x1
ldr x0, [sp]
add x1, x0, x1
str x1, [sp, #16]
```

A. 

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>2</td>
</tr>
<tr>
<td>x1</td>
<td>50</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>x31 (sp)</td>
<td>0x1260</td>
</tr>
</tbody>
</table>

B. 

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>50</td>
</tr>
<tr>
<td>x1</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>x31 (sp)</td>
<td>0x1260</td>
</tr>
</tbody>
</table>

C. 

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>3</td>
</tr>
<tr>
<td>x1</td>
<td>48</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>x31 (sp)</td>
<td>0x1260</td>
</tr>
</tbody>
</table>
What will the machine state be after executing these instructions?  
(Bonus: write an equivalent one-line expression in C code)

```assembly
ldr x0, [sp, #8]
ldr x1, [sp, #16]
lsl x1, x1, #3
mul x1, x0, x1
ldr x0, [sp]
add x1, x0, x1
str x1, [sp, #16]
```

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Value</th>
<th>C variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1260</td>
<td>2</td>
<td>x</td>
</tr>
<tr>
<td>0x1268</td>
<td>3</td>
<td>y</td>
</tr>
<tr>
<td>0x1270</td>
<td>2</td>
<td>z</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Registers Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>x31 (sp)</td>
<td>0x1260</td>
</tr>
</tbody>
</table>
What will the machine state be after executing these instructions?

(Bonus: write an equivalent one-line expression in C code)

```c
ldr x0, [sp, #8]  x0 ← y
ldr x1, [sp, #16]  x1 ← z
lsl x1, x1, #3  x1 ← x1 << 3
mul x1, x0, x1  x1 ← x0 * x1
ldr x0, [sp]  x0 ← x
add x1, x0, x1  x1 ← x0 + x1
str x1, [sp, #16]  z ← x1
```

C Expression:

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Value</th>
<th>C variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1260</td>
<td>2</td>
<td>x</td>
</tr>
<tr>
<td>0x1268</td>
<td>3</td>
<td>y</td>
</tr>
<tr>
<td>0x1270</td>
<td>50</td>
<td>z</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>2</td>
</tr>
<tr>
<td>x1</td>
<td>50</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>x31 (sp)</td>
<td>0x1260</td>
</tr>
</tbody>
</table>
What will the machine state be after executing these instructions?

(Bonus: write an equivalent one-line expression in C code)

\[
\begin{align*}
\text{ldr } x0, [sp, \#8] & \quad x0 \leftarrow y \\
\text{ldr } x1, [sp, \#16] & \quad x1 \leftarrow z \\
\text{lsl } x1, x1, \#3 & \quad x1 \leftarrow x1 \ll 3 \\
\text{mul } x1, x0, x1 & \quad x1 \leftarrow x0 \times x1 \\
\text{ldr } x0, [sp] & \quad x0 \leftarrow x \\
\text{add } x1, x0, x1 & \quad x1 \leftarrow x0 + x1 \\
\text{str } x1, [sp, \#16] & \quad z \leftarrow x1
\end{align*}
\]

C Expression: 
\[z = z \times 8 \times y + x\]
What will the machine state be after executing these instructions?

```
ldr x0, [sp]
ldr x1, [sp, #-16]
orr x0, x0, #15
neg x1, x0
stp x0, x1, [sp, #8]
```
How might you execute this C statement in ARM64 assembly?

\[ z = x \ ^ \ y \]
How might you execute this C statement in ARM64 assembly?

\[ z = x \oplus y \]

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Value</td>
</tr>
<tr>
<td>x0</td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>x31 (sp)</td>
<td>0x3230</td>
</tr>
</tbody>
</table>

A: 
1. `ldr x0, [sp, #8]`
2. `ldr x1, [sp]`
3. `eor x0, x0, x1`
4. `str x0, [sp, #-8]`

B: 
1. `ldr x0, [sp, #-8]`
2. `ldr x1, [sp]`
3. `eor x0, x0, x1`
4. `str x0, [sp, #8]`

C: I came up with some other way.
How might you execute this C statement in ARM64 assembly?

\[ z = (z - 5) \land \lnot y \]
These are each just one example of many ways to execute these C statements in ARM64 assembly!
Control Flow

• Previous examples focused on:
  • data movement (mov, ldr, str)
  • arithmetic (add, sub, orr, neg, lsl, etc.)

• Up next: branching (aka jumping)!

  (Changing which instruction we execute next.)
Relevant XKCD

```
I could restructure the program's flow
or use one little 'goto' instead.

Eh, screw good practice. How bad can it be?
goto main_sub3;

*compile*
```

xkcd #292
Unconditional branching / goto

int func(void) {
    int a = 10;
    int b = 20;
    goto label1;
    a = a + b;
    label1:
    return a;
}

A label is a place you might jump to.

Labels ignored except for goto/branches.

(Skipped over if encountered)

int x = 20;
Label1:
    int y = x + 30;
Label2:
    printf("%d, %d\n", x, y);
ARM64 Labels

- Label represents a place to which you might branch
  - The assembler determines the address of the label
  - The address will often be displayed as an offset from the start of a function

- For "local" labels (the kind you'll be writing for if/else and loops):
  - Convention says to prefix them with a . character
  - e.g., .L1:
Unconditional branching / \texttt{goto}

```c
int func(void) {
    int a = 10;
    int b = 20;
    goto label1;
    a = a + b;
    label1:
    return a;
}
```

Note, we declared '\texttt{int}' variables (32-bit), so we use the \texttt{w} register prefix!
Unconditional branching / \texttt{goto}

- Uses for unconditional branching:
  - (intentional) infinite loop
  - break
  - continue
  - function calls (handled differently than just \texttt{b} instruction)

- Often, we only want to branch when \textit{something} is true / false.

- Need a way to compare values, branch based on comparison results.
Condition Codes (or "Flags")

• Set in two ways:
  1. In response to explicit comparison instructions
  2. As “side effects” produced by ALU with instructions suffixed by \texttt{s}
     • e.g., \texttt{adds} \hspace{1em} \texttt{subs}

• ARM64, condition codes tell you:
  • \texttt{N}: the result of the ALU operation is negative (high-order bit is 1)
  • \texttt{Z}: the result of the ALU operation is zero
  • \texttt{C (carry)}: the result, \textit{if interpreted as unsigned}, has overflowed
  • \texttt{V}: the result, \textit{if interpreted as signed}, has overflowed
Instructions that set condition codes

1. Arithmetic/logic side effects (adds, subs, ands, etc.)

2. CMP and TEST:
   - `$cmp a, b$` like computing $a - b$ without storing result
     - Sets V if overflow, Sets C if carry-out,
       Sets Z if result is zero, Sets N if result is negative
   
   `$tst a, b$` like computing $a \& b$ without storing result
     - Sets Z if result is zero, sets N if result is negative
       V and C flags are zero (there is no overflow with $\&$)
Which flags would this `subs` set?

- Suppose x0 holds 5, x1 holds 7

```
subs x0, #5
```

If the result is zero (Z)
If the result’s first bit is set (negative if signed) (N)
If the result overflowed (assuming unsigned) (C)
If the result overflowed (assuming signed) (V)

A. Z
B. N
C. C and Z
D. C and N
E. C, N, and V
Which flags would this `cmp` set?

- Suppose `x0` holds 5, `x1` holds 7

```
cmp x0, x1
```

If the result is zero (Z)
If the result’s first bit is set (negative if signed) (N)
If the result overflowed (assuming unsigned) (C)
If the result overflowed (assuming signed) (V)

A. Z
B. N
C. C and Z
D. C and N
E. C, N, and V
Conditional Branching

• b.SUFFIX: branch based on which condition codes are set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>b.eq</td>
<td>Z</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>b.ne</td>
<td>~Z</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>b.mi</td>
<td>N</td>
<td>Negative</td>
</tr>
<tr>
<td>b.pl</td>
<td>~N</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>b.gt</td>
<td>~(N^V) &amp; ~Z</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>b.ge</td>
<td>~(N^V)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>b.lt</td>
<td>(N^V)</td>
<td>~Z</td>
</tr>
<tr>
<td>b.le</td>
<td>(N^V)</td>
<td>Less or Equal (Signed)</td>
</tr>
</tbody>
</table>

See book section 9.4.1

You do not need to memorize these!
Example Scenario

long userval;
scanf("%ld", &userval);

if (userval == 42) {
    userval += 5;
} else {
    userval -= 10;
}
...

• Suppose user gives us a value via scanf
• We want to check to see if it equals 42
  • If so, add 5
  • If not, subtract 10
How might we use branches/CCs for this?

Assume userval is stored in register x0 at this point.

long userval;
scanf("%ld", &userval);

if (userval == 42) {
    userval += 5;
} else {
    userval -= 10;
}
How would we use jumps/CCs for this?

Assume `userval` is stored in register x0 at this point.

```c
long userval;
scanf("%ld", &userval);

if (userval == 42) {
    userval += 5;
} else {
    userval -= 10;
}
```

(A) ```c
    cmp x0, #42
    b.eq .L2
    .L1:
    sub x0, #10
    b .DONE
    .L2:
    add x0, #5
    .DONE:
    ...
```  

(B) ```c
    cmp x0, #42
    b.ne .L2
    .L1:
    sub x0, #10
    b .DONE
    .L2:
    add x0, #5
    .DONE:
    ...
```  

(C) ```c
    cmp x0, #42
    b.ne .L2
    .L1:
    add x0, #5
    b .DONE
    .L2:
    sub x0, #10
    .DONE:
    ...
```
Loops

• We’ll look at these in the lab!
Summary

• ISA defines what programmer can do on hardware
  • Which instructions are available
  • How to access state (registers, memory, etc.)
  • This is the architecture’s assembly language

• In this course, we’ll be using ARM64 (AArch64)
  • Instructions for:
    • moving data (mov, ldr, str)
    • arithmetic (add, sub, mul, orr, lsl, etc.)
    • control (b, b.eq, b.ne, etc.)
  • Condition codes for making control decisions
    • If the result is zero (Z)
    • If the result’s first bit is set (negative if signed) (N)
    • If the result overflowed (assuming unsigned) (C)
    • If the result overflowed (assuming signed) (V)