Assembly Language: Part 1
Context of this Lecture

First half of the semester: “Programming in the large”
Second half: “Under the hood”

Starting Now

C Language

Assembly Language

Machine Language

language levels tour

Later

Application Program

Operating System

Hardware

service levels tour
Lectures vs. Precepts

Approach to studying assembly language:

<table>
<thead>
<tr>
<th>Lectures</th>
<th>Precepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Study <em>partial</em> pgms</td>
<td>Study <em>complete</em> pgms</td>
</tr>
<tr>
<td>Begin with <em>simple</em> constructs; proceed to <em>complex</em> ones</td>
<td>Begin with <em>small</em> pgms; proceed to <em>large</em> ones</td>
</tr>
<tr>
<td>Emphasis on <em>reading</em> code</td>
<td>Emphasis on <em>writing</em> code</td>
</tr>
</tbody>
</table>
Agenda

Language Levels

Architecture

Assembly Language: Performing Arithmetic

Assembly Language: Load/Store and Defining Global Data
High-Level Languages

Characteristics

• Portable
  • To varying degrees
• Complex
  • One statement can do much work – good ratio of functionality to code size
• Human readable
  • Structured – if(), for(), while(), etc.

```c
count = 0;
while (n>1)
{
  count++;
  if (n&1)
    n = n*3+1;
  else
    n = n/2;
}
```
Machine Languages

Characteristics

• Not portable
  • Specific to hardware
• Simple
  • Each instruction does a simple task – poor ratio of functionality to code size
• Not human readable
  • Not structured
  • Requires lots of effort!
  • Requires tool support

```
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
9222 9120 1121 A120 1121 A121 7211 0000
0000 0001 0002 0003 0004 0005 0006 0007
0008 0009 000A 000B 000C 000D 000E 000F
0000 0000 0000 FE10 FACE CAFE ACED CEDE
1234 5678 9ABC DEF0 0000 0000 F00D 0000
0000 0000 EEEE 1111 EEEE 1111 0000 0000
B1B2 F1F5 0000 0000 0000 0000 0000 0000
```
Assembly Languages

Characteristics

• Not portable
  • Each assembly lang instruction maps to one machine lang instruction
• Simple
  • Each instruction does a simple task
• Human readable
  (In the same sense that Polish is human readable, if you know Polish.)

```
loop:
    mov    w1, 0
    cmp    w0, 1
    ble    endloop
    add    w0, w0, #1
    ands   wzr, w0, #1
    beq    else
    add    w2, w0, w0
    add    w0, w0, w2
    add    w0, w0, 1
    b      endif
else:
    asr    w0, w0, 1
endif:
    b      loop
endloop:
```
Q: Why learn assembly language?

A: Knowing assembly language helps you:

• Write faster code
  • In assembly language
  • In a high-level language!
• Write safer code
  • Understanding mechanism of potential security problems helps you avoid them – even in high-level languages
• Understand what’s happening “under the hood”
  • Someone needs to develop future computer systems
  • Maybe that will be you!
• Become more comfortable with levels of abstraction
  • Become a better programmer!
Why Learn ARM Assembly Lang?

Why learn ARMv8 (a.k.a. AARCH64) assembly language?

Pros

• ARM is the most widely used processor in the world (in your phone, in your Chromebook, in the internet-of-things, Armlab)
• ARM has a modern and (relatively) elegant instruction set, compared to the big and ugly x86-64 instruction set

Cons

• x86-64 dominates the desktop/laptop, for now (but there are rumors that Apple is going to shift Macs to ARM…)
Agenda

Language Levels

Architecture

Assembly Language: Performing Arithmetic

Assembly Language: Load/Store and Defining Global Data
John Von Neumann (1903-1957)

In computing
- Stored program computers
  - Cellular automata
  - Self-replication

Other interests
- Mathematics
- Inventor of game theory
- Nuclear physics (hydrogen bomb)

Princeton connection
- Princeton Univ & IAS, 1930-1957

Known for “Von Neumann architecture (1950)”
- In which programs are just data in the memory
- Contrast to the now-obsolete “Harvard architecture”
Von Neumann Architecture

Instructions (encoded within words) are fetched from RAM

Control unit interprets instructions

• to shuffle data between registers and RAM

• to move data from registers to ALU (arithmetic+logic unit) where operations are performed
RAM (Random Access Memory)
Conceptually: large array of bytes (gigabytes+ in modern machines)

- Contains data (program variables, structs, arrays)
- and the program!

Instructions are fetched from RAM
Von Neumann Architecture

**Registers**

Small amount of storage on the CPU (tens of words in modern machines)

- Much faster than RAM
- Top of the “storage hierarchy”: above RAM, disk, etc.

ALU (arithmetic+logic unit) instructions operate on registers
Registers and RAM

Typical pattern:
• **Load** data from RAM to registers
• **Manipulate** data in registers
• **Store** data from registers to RAM

On AARCH64, this pattern is enforced
• “Manipulation” instructions can only access registers
• This is known as a **Load/store architecture**
• Characteristic of “RISC” (Reduced Instruction Set Computer) vs. “CISC” (Complex Instruction Set Computer) architectures, e.g. x86
Registers (ARM-64 architecture)

- \( x_0 \) \( w_0 \)
- \( x_1 \) \( w_1 \)
- \( \ldots \)
- \( x_{29} \) (FP) \( w_{29} \)
- \( x_{30} \) (LR) \( w_{30} \)
- \( xzr \) (all zeros) \( wzr \)
- \( sp \) (stack pointer)
- \( pc \) (program counter)
- \( nzcv \)
- \( pstate \)
General-Purpose Registers

X0 .. X30
- 64-bit registers
- Scratch space for instructions, parameter passing to/from functions, return address for function calls, etc.
- Some have special purposes defined in hardware (e.g. X30) or defined by software convention (e.g. X29)
- Also available as 32-bit versions: W0 .. W30

XZR
- On read: all zeros
- On write: data thrown away
SP Register

Special-purpose register…

- Contains **SP (Stack Pointer)**: address of top (low address) of current function’s stack frame

Allows use of the STACK section of memory

*(See *Assembly Language: Function Calls* lecture)*
PC Register

Special-purpose register…
- Contains **PC (Program Counter)**
- Stores the location of the next instruction
  - Address (in TEXT section) of machine-language instructions to be executed next
- Value changed:
  - Automatically to implement sequential control flow
  - By branch instructions to implement selection, repetition
PSTATE Register

Special-purpose register…

- Contains **condition flags:**
  - n (Negative),  z (Zero),  c (Carry),  v (Overflow)
- Affected by compare (**cmp**) instruction
  - And many others, if requested
- Used by conditional branch instructions
  - **beq**, **bne**, **blo**, **bhi**, **ble**, **bge**, …
  - (See **Assembly Language: Part 2** lecture)
Agenda

Language Levels

Architecture

**Assembly Language: Performing Arithmetic**

**Assembly Language: Load/Store and Defining Global Data**
Many instructions have this format:

\[
\text{name}\{,s\} \text{ dest, src1, src2} \\
\text{name}\{,s\} \text{ dest, src1, immed}
\]

- **name**: name of the instruction (add, sub, mul, and, etc.)
- **s**: if present, specifies that condition flags should be set

- dest and src1, src2 are \( x \) registers: 64-bit operation
- dest and src1, src2 are \( w \) registers: 32-bit operation

- src2 may be a constant ("immediate" value) instead of a register
64-bit Arithmetic

C code:
```c
static long length;
static long width;
static long perim;
...
perim =
(length + width) * 2;
```

Assume that...
- length stored in x1
- width stored in x2
- perim stored in x3

We’ll see later how to make this happen

Assembly code:
```asm
add x3, x1, x2
ls1 x3, x3, 1
```

Recall use of left shift by 1 bit to multiply by 2
More Arithmetic

```java
static long x;
static long y;
static long z;
...
z = x - y;
z = x * y;
z = x / y;
z = x & y;
z = x | y;
z = x ^ y;
z = x >> y;
```

Assume that…
- x stored in x1
- y stored in x2
- z stored in x3

We’ll see later how to make this happen

Note arithmetic shift!
Logical right shift with `lsr` instruction
More Arithmetic: Shortcuts

Assume that...
- x stored in x1
- y stored in x2
- z stored in x3

We’ll see later how to make this happen

static long x;
static long y;
static long z;
...
z = x;
z = -x;

mov x3, x1
neg x3, x1
orr x3, xzr, x1
sub x3, xzr, x1

These are actually assembler shortcuts for instructions with XZR!
Signed vs Unsigned?

Assume that...

- x stored in x1
- y stored in x2

Mostly the same algorithms, same instructions!
- Can set different condition flags in PSTATE
- Exception is division: \texttt{sdiv} vs \texttt{udiv} instructions
32-bit Arithmetic

static int length;
static int width;
static int perim;
...
perim =
(length + width) * 2;

Assume that…
- length stored in w1
- width stored in w2
- perim stored in w3

We’ll see later how to make this happen

Assembly code using “w” registers:

```
add w3, w1, w2
lsl w3, w3, 1
```
8- and 16-bit Arithmetic?

```c
static char x;
static short y;
...
x++;
y--;
```

No specialized instructions

- Use “w” registers
- Specialized “load” and “store” instructions for transfer of shorter data types from / to memory – we’ll see these later
- Corresponds to C language semantics: all arithmetic is implicitly done on (at least) ints
Agenda

Language Levels
Architecture
Assembly Language: Performing Arithmetic
Assembly Language: Load/Store and Defining Global Data
Loads and Stores

Most basic way to load (from RAM) and store (to RAM):

```
ldr dest, [src]
str src, [dest]
```

- dest and src are registers!
- Registers in [brackets] contain memory addresses
  - Every memory access is through a “pointer”!

- How to get correct memory address into register?
  - Depends on whether data is on stack (local variables), heap (dynamically-allocated memory), or global / static
  - For today, we’ll look only at the global / static case
Loads and Stores

```c
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
```

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr    x0, length
    ldr    w1, [x0]
    adr    x0, width
    ldr    w2, [x0]
    add    w1, w1, w2
    lsl    w1, w1, 1
    adr    x0, perim
    str    w1, [x0]
    mov    w0, 0
    ret
```
Loads and Stores

```c
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
    (length + width) * 2;
    return 0;
}
```

Sections
.data: read-write
.rodata: read-only
.bss: read-write, initialized to zero
.text: read-only, program code

Stack and heap work differently!
Loads and Stores

```c
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
       (length + width) * 2;
    return 0;
}
```

Declaring data

“Labels” for locations in memory

```assembly
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr x0, length
    ldr w1, [x0]
    adr x0, width
    ldr w2, [x0]
    add w1, w1, w2
    lsl w1, w1, 1
    adr x0, perim
    str w1, [x0]
    mov w0, 0
    ret
```
loads and stores

static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}

.global main
main:
adr    x0, length
ldr    w1, [x0]
adr    x0, width
ldr    w2, [x0]
add    w1, w1, w2
lsl    w1, w1, 1
adr    x0, perim
str    w1, [x0]
mov    w0, 0
ret

global symbol
Declare “main” to be a globally-visible label
Loads and Stores

```c
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
```

Generating addresses
adh instruction stores address of a label in a register

```assembly
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr x0, length
    ldr w1, [x0]
    adr x0, width
    ldr w2, [x0]
    add w1, w1, w2
    lsl w1, w1, 1
    adr x0, perim
    str w1, [x0]
    mov w0, 0
    ret
```
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}

Load and store
Use “pointer” in x0 to load from and store to memory
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
    (length + width) * 2;
    return 0;
}
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}

.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr x0, length
    ldr w1, [x0]
    adr x0, width
    ldr w2, [x0]
    add w1, w1, w2
    lsl w1, w1, 1
    adr x0, perim
    str w1, [x0]
    mov w0, 0
    ret
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
Loads and Stores

static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}

Return value
Passed in register w0
Loads and Stores

```c
static int length = 1;
static int width = 2;
static int perim = 0;

int main()
{
    perim =
        (length + width) * 2;
    return 0;
}
```

```
.section .data
length: .word 1
width: .word 2
perim: .word 0

.section .text
.global main
main:
    adr x0, length
    ldr w1, [x0]
    adr x0, width
    ldr w2, [x0]
    add w1, w1, w2
    lsl w1, w1, 1
    adr x0, perim
    str w1, [x0]
    mov w0, 0
    ret
```

Return to caller

ret instruction
# Defining Data: DATA Section 1

```c
static char c = 'a';
static short s = 12;
static int i = 345;
static long l = 6789;
```

```assembly
.section "".data"
c:
   .byte 'a'
s:
   .short 12
i:
   .word 345
l:
   .quad 6789
```

**Notes:**
- `.section` instruction (to announce DATA section)
- label definition (marks a spot in RAM)
- `.byte` instruction (1 byte)
- `.short` instruction (2 bytes)
- `.word` instruction (4 bytes)
- `.quad` instruction (8 bytes)
Defining Data: DATA Section 2

```c
char c = 'a';
short s = 12;
int i = 345;
long l = 6789;
```

```plaintext
.sect .".data"
.global c
c: .byte 'a'
.global s
s: .short 12
.global i
i: .word 345
.global l
l: .quad 6789
```

Notes:
Can place label on same line as next instruction
```
.global instruction
```
Defining Data: BSS Section

```c
static char c;
static short s;
static int i;
static long l;
```

Notes:
- `.section` instruction (to announce BSS section)
- `.skip` instruction
Defining Data: RODATA Section

```
...  "hello\n"...;
...

)section ".rodata"
helloLabel:
  .string "hello\n"
```

Notes:

- `.section` instruction (to announce RODATA section)
- `.string` instruction
### Signed vs Unsigned, 8- and 16-bit

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ldrb</code></td>
<td>dest, [src]</td>
</tr>
<tr>
<td><code>ldrh</code></td>
<td>dest, [src]</td>
</tr>
<tr>
<td><code>strb</code></td>
<td>src, [dest]</td>
</tr>
<tr>
<td><code>strh</code></td>
<td>src, [dest]</td>
</tr>
<tr>
<td><code>ldrsb</code></td>
<td>dest, [src]</td>
</tr>
<tr>
<td><code>ldrsh</code></td>
<td>dest, [src]</td>
</tr>
<tr>
<td><code>ldrsw</code></td>
<td>dest, [src]</td>
</tr>
</tbody>
</table>

**Special instructions for reading/writing bytes (8 bit), shorts (“half-words”: 16 bit)**

- See appendix of these slides for information on ordering: little-endian vs. big-endian

**Special instructions for signed reads**

- “Sign-extend” byte, half-word, or word to 32 or 64 bits
Summary

Language levels

The basics of computer architecture
  • Enough to understand AARCH64 assembly language

The basics of AARCH64 assembly language
  • Instructions to perform arithmetic
  • Instructions to define global data and perform data transfer

To learn more
  • Study more assembly language examples
    • Chapters 2-5 of Pyeatt and Ughetta book
  • Study compiler-generated assembly language code
    • gcc217 -S somefile.c
Big-endian vs little-endian byte order
Byte Order

AARCH64 is a **little endian** architecture

- **Least** significant byte of multi-byte entity is stored at lowest memory address
- “Little end goes first”

```
00000101
00000000
00000000
00000000
1000
1001
1002
1003
```
The int 5 at address 1000:

Some other systems use **big endian**

- **Most** significant byte of multi-byte entity is stored at lowest memory address
- “Big end goes first”

```
00000000
00000000
00000000
00000101
1000
1001
1002
1003
```
The int 5 at address 1000:
#include <stdio.h>

int main(void)
{
    unsigned int i = 0x003377ff;
    unsigned char *p;
    int j;
    p = (unsigned char *)&i;
    for (j = 0; j < 4; j++)
        printf("Byte %d: %2x\n", j, p[j]);
}

Output on a little-endian machine

| Byte 0: ff |
| Byte 1: 77 |
| Byte 2: 33 |
| Byte 3: 00 |

Output on a big-endian machine

| Byte 0: 00 |
| Byte 1: 33 |
| Byte 2: 77 |
| Byte 3: ff |
Byte Order Example 2

Note:
Flawed code; uses “b” instructions to load from a four-byte memory area

AARCH64 is little endian, so what will be the value in x1?

What would be the value in x1 if AARCH64 were big endian?

```assembly
section ".data"
foo: .word 1
...

section ".text"
...
adr x0, foo
ldrb w1, [x0]
```
Note:
Flawed code; uses word instructions to manipulate a one-byte memory area

What would happen?

```
.section "data"
foo: .byte 1
...
.section "text"
...
adr  x0, foo
ldr  w1, [x0]
```