Princeton University
Computer Science 217: Introduction to Programming Systems

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

Later
Application Program
Operating System
Hardware

Lectures vs. Precepts
Approach to studying assembly language:

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

Agenda
Language Levels
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

```assembly
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.)

```
mov w1, 0
Loop:
  cmp w0, 1
  ble endwhile
  add w0, w0, #1
  and w2, w0, w0
  add w0, w0, w2
  add w0, w0, 1
  b endif
endif:
  asr w0, w0, 1
while:
  b loop
endloop:
```

Why Learn Assembly Language?

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…)

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
Von Neumann Architecture

**RAM (Random Access Memory)**
- Contains data (program variables, structs, arrays)
- Contains the program!
Instructions are fetched from RAM

**Control Unit**

**CPU**

**Registers**

**Data bus**

**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

**Control Unit**

**CPU**

** Registers**

**ALU**

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

**SP Register**

Special-purpose register...
- Contains SP (Stack Pointer): address of top (low address) of current function’s stack frame

**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. X29) 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**

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, bge, ...
  - (See Assembly Language: Part 2 lecture)

Agenda

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

ALU

ALU

Operation

dest

ALU

PSTATE

src1

src2

RAM

Control Unit

CPU

Registers

Data bus

Instruction Format

Many instructions have this format:

```
name[,s] dest, src1, src2
name[,s] dest, src1, imm
```

- 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:

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

Assembly code:

```
add x3, x2, x1
lsl x3, x3, 1
```

Assume that...

- length stored in x1
- width stored in x2
- perim stored in x3

We’ll see later how to make this happen

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

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

We’ll see later how to make this happen

\[ z = x - y; \]
\[ z = x \times y; \]
\[ z = x / y; \]
\[ z = x \& y; \]
\[ z = x | y; \]
\[ z = x ^ y; \]
\[ z = x \gg y; \]

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

\[ \text{mov} \ x3, x1 \]
\[ \text{neg} \ x3, x1 \]

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: \( \text{sdiv} \) vs \( \text{udiv} \) instructions

Assembly code using “w” registers:

32-bit Arithmetic

Assume that...
- \( \text{length} \) stored in \( w1 \)
- \( \text{width} \) stored in \( w2 \)
- \( \text{perim} \) stored in \( w3 \)

We’ll see later how to make this happen

Assembly code using “w” registers:

8- and 16-bit Arithmetic?

No specialized instructions
- Use “w” registers
- Specialized “load” and “store” instructions for transfer of shorter data types from \( f \) 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):

```asm
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

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

Sections
- `.data`: read-write
  - `.word`: 32-bit integer
- `.rodata`: read-only, initialized to zero
- `.bss`: read-write, initialized to zero

Stack and heap work differently!

Global symbol
- Declare "main" to be a globally-visible label

Generating addresses
- `adr` instruction stores address of a label in a register
Loads and Stores

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

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;
}

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

.return value
Passed in register w0
Defining Data: 

**BSS Section**

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

```asm
.section "bss"
c: .skip 1
s: .skip 2
i: .skip 4
l: .skip 8
```

Notes:
- `.section` instruction (to announce BSS section)
- `.skip` instruction

---

**RODATA Section**

```c
"hello\n";
```

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

Notes:
- `.section` instruction (to announce RODATA section)
- `.string` instruction

---

**Signed vs Unsigned, 8- and 16-bit**

```asm
ldrb dest, [src]
ldrh dest, [src]
strb src, [dest]
strh src, [dest]
ldrsb dest, [src]
ldrsh dest, [src]
ldrsw dest, [src]
```

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`

---

**Appendix**

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”

Some other systems use big endian
- **Most** significant byte of multi-byte entity is stored at lowest memory address
  - “Big end goes first”

---
Byte Order Example 1

```c
#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

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

.section .text
adr x0, foo
ldrb w1, [x0]
```

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

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

---

Byte Order Example 3

Note:
Flawed code; uses word instructions to manipulate a one-byte memory area

```assembly
.section .data
foo:
    .byte 1

.section .text
adr x0, foo
ldr w1, [x0]
```

What would happen?