Assembly Language

Part 1
### Approach to studying assembly language:

<table>
<thead>
<tr>
<th>Lectures</th>
<th>Precepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Study partial programs</td>
<td>Study complete programs</td>
</tr>
<tr>
<td>Begin with simple constructs; proceed to complex ones</td>
<td>Begin with small programs; 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

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.

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

Characteristics

• Not portable (hardware-specific)
• 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 language instruction maps to one machine 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    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:
```
Why Learn Assembly Language?

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 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 ... soon in Macs.)
• 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)
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 and statistics
- Inventor of game theory
- Nuclear physics

Princeton connection
- Princeton University & IAS, 1930-1957
- [https://paw.princeton.edu/article/early-history-computing-princeton](https://paw.princeton.edu/article/early-history-computing-princeton)

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

Registers

- Small amount of storage on the CPU
  - Top of the “storage hierarchy”
  - Very {small, expensive, fast}

ALU instructions operate on registers
ALU Arithmetic Example

operation → ALU → flags

src1 → ALU → dest

src2 → ALU → dest

Control Unit

ALU

Registers

CPU

Data bus

RAM
Von Neumann Architecture

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
TOY REFERENCE CARD

INSTRUCTION FORMATS

| Format  RR: | opcode | d | a | t | (0-6, A-B) |
| Format  A:  | opcode | d | a | addr | (7-9, C-F) |

ARITHMETIC and LOGICAL
1: add
2: subtract
3: and
4: xor
5: shift left
6: shift right

TRANSFER between registers
7: load address
8: load
9: store
A: load indirect
B: store indirect

CONTROL
0: halt
C: branch zero
D: branch positive
E: jump register
F: jump and link

Word size. The TOY machine has two types of storage: main memory and registers. Each entity stores one word of information. On the TOY machine, a word is a sequence of 16 bits. Typically, we interpret these 16 bits as a hexadecimal integer in the range 0000 through FFFF. Using two's complement notation, we can also interpret it as a decimal integer in the range -32,768 to +32,767. See Section 5.1 for a refresher on number representations and two's complement integers.

Main memory. The TOY machine has 256 words of main memory. Each memory location is labeled with a unique memory address. By convention, we use the 256 hexadecimal integers in the range 00 through FF. Think of a memory location as a mailbox, and a memory address as a postal address. Main memory is used to store instructions and data.

Registers. The TOY machine has 16 registers, indexed from 0 through F. Registers are much like main memory: each register stores one 16-bit word. However, registers provide a faster form of storage than main memory. Registers are used as scratch space during computation and play the role of variables in the TOY language. Register 0 is a special register whose output value is always 0.

Program counter. The program counter, or pc, is an extra register that keeps track of the next instruction to be executed. It stores 8 bits, corresponding to a hexadecimal integer in the range 00 through FF. This integer stores the memory address of the next instruction to execute.

Register 0 always reads 0.

Loads from M[FF] come from stdin.
Stores to M[FF] go to stdout.

16-bit registers (two's complement)
16-bit memory locations
8-bit program counter

https://introcs.cs.princeton.edu/java/62toy/
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**
  (as opposed to “register-memory” architectures)
- Characteristic of “RISC” (Reduced Instruction Set Computer) vs. “CISC” (Complex Instruction Set Computer) architectures, e.g. x86
Registers (ARM-64 architecture)

- x0
- w0
- x1
- w1
- ...
- x29 (FP)
- w29
- x30 (LR)
- w30
- xzr (all zeros)
- wzr
- sp (stack pointer)
- pc (program counter)
- nzcv
- pstate

You mean "branching", right?
General-Purpose 64-bit Registers

X0 ... X30

- Scratch space for instructions, parameter passing to/from functions, return address for function calls, etc.
- Some have special roles 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
- Also available as 32-bit version: WZR
SP Register

Special-purpose register...

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

Allows use of the STACK section of memory

(See Assembly Language: Function Calls lecture later)
Special-purpose register...

- **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
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
ALU Arithmetic Example

operation → ALU → flags

src1 → ALU → dest

src2 → ALU → dest

RAM

Control Unit

ALU

Registers

CPU

Data bus
Instruction Format

Many instructions have this format:

```
name{,s} dest, src1, src2
name{,s} 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...
- there’s a good reason for having variables with file scope, process duration
- length stored in x1
- width stored in x2
- perim stored in x3

We’ll see later how to make this happen

Assembly code:

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

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

Note arithmetic shift!
Logical right shift would be 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?

```c
static long x;
static unsigned long y;
...
x++;
y--;
```

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

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

- dest and src are registers!
- Contents of registers in [brackets] must be memory addresses
  - Every memory access is through a “pointer”!
Signed vs Unsigned, 8- and 16-bit

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
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
Our First Full Program*

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

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

```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
```

* Sorry, I know by convention it should be “Hello, World!”. You’ll see that in precept.
Memory sections

Sections (Stack/heap are different!)
- **.rodata**: read-only
- **.data**: read-write
- **.bss**: read-write (initialized to 0)
- **.text**: read-only, program code

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

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

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

.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
```
Variable definitions

```
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

`.word`: 32-bit int and initial value

See appendix for variables in other sections, with other types.
main()

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

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

Global visibility

.globl: Declare "main" to be a globally-visible label

.globl main

.perim: .word 0

.globl 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

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

.section .text
.globl 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;
}

Generating addresses

adr: put address of
a label in a register

.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 x0 as a “pointer” 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;
}

Return a value

ret: return to the caller*, with register 0 holding the return value

* or, in A6, not.

Return
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;
}
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;
}

Return to caller
ret instruction
Appendix 1

DEFINING DATA:
OTHER SECTIONS AND SIZES
Defining Data: DATA Section 1

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

Notes:

```
.section " .data "

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

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

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

Notes:
- Can place label on same line as next instruction
- `.global` directive can also apply to variables, not just functions
### Defining Data: BSS Section

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

**Notes:**
- `.section` directive (to announce BSS section)
- `.skip` directive (to specify number of bytes)

```assembly
.section ".bss"
  c:
    .skip 1
  s:
    .skip 2
  i:
    .skip 4
  l:
    .skip 8
```
Notes:
- `.section` directive (to announce RODATA section)
- `.string` directive
Appendix 2

BYTE ORDER: BIG-ENDIAN VS LITTLE-ENDIAN
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
```

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”

```
1000 00000000
1001 00000000
1002 00000000
1003 00000000
```

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]);
}
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 returned from w0?

What would be the value returned from w0 if AARCH64 were big endian?

```
.section "\data"
foo: .word 7

.section "\text"
.global "main"
main:
  adr x0, foo
  ldrb w0, [x0]
  ret
```
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 curated/hand-written assembly language examples
    • Chapters 2-5 of Pyeatt and Ughetta book
    • Study compiler-generated assembly language code (complicated, YMMV)
      • gcc217 –S somefile.c