Lecture 2: Instruction Semantics

Spring 2020
Jason Tang
Topics

- CPU operations
- Instruction types
- Functional calls
Instructions

- Although there are many different CPU designers, they all share similar instruction sets:
  - Fundamental operations that all computers must provide
  - Designers have same goal of finding a machine language that maximizes performance while minimizing cost
- This class is based around the ARMv8-A
  - Designed by Arm holdings, is a British semiconductor company
  - Used in nearly all modern smartphones, tablets, and smartwatches
Instruction Set Architecture (ISA)

• ISA: abstract interface between hardware and lowest level of software, encompassing all necessary information to write and run a machine language program

  • Includes instructions, registers, memory access, I/O, etc.

  • Enables different implementations of ISA to run the same software

• Application Binary Interface (ABI): user portion of instruction set that defines how software should use the ISA to pass data between themselves and the underlying operating system

  • Defines a standard for binary portability across computers, OS revisions, and compiler revisions
Changes in CPU Popularity

- In this decade, the shift has been from high performance computers to handheld and embedded systems

Operations of Computer Hardware

• Assembly language is a symbolic representation of what the processor actually understands

• Different assembly language syntaxes can exist for the same ISA
  
  • See AT&T vs Intel debate

• In ARMv8-A, each assembly line translates to a single instruction

• Example C code and resulting assembly:

  $$f = (g + h) - (i + j)$$

  \[
  \begin{align*}
  &\text{add } w1, w1, w0 \\
  &\text{add } w0, w2, w3 \\
  &\text{sub } w0, w1, w0
  \end{align*}
  \]
Operands of Computer Hardware

• Registers are the bricks of computer construction

  • Hardware design primitives visible to programmers

• Size and number of registers differ by architectures

  • For ARMv8-a, the 31 general purpose registers are 64 bits

  • When referring to the lower 32-bits, use notation $w_n$; to refer to all 64 bits use $x_n$

• For many operations, the destination register is listed first

\[
\begin{align*}
\text{add} & : w1, w1, w0 \\
\text{add} & : w0, w2, w3 \\
\text{sub} & : w0, w1, w0
\end{align*}
\]
Common Addressing Modes

- **Direct** (or immediate): value encoded in instruction

- **Register**: value stored within register

- **Register indirect**: value stored in memory at the address given by a register

- **Register displacement**: base (a register) + offset (an immediate value)
  - **Scaled displacement**: base + offset \( \times \) scale

- **PC-relative**: Program Counter + offset \( \times \) scale
Complex Data Representation

• Given limited number of registers, remaining data (e.g., structs and arrays) are kept in memory

• Data transfer instructions to move data in memory to / from a register are traditionally called load / store

• Accessing a specific memory address is done indirectly, via register displacement

• Example in C: \( g = h + A[8] \)

  • Let \( A \) be an array of \texttt{int}s (32-bits each), and its starting address is in register \( w0 \)

  • Then resulting assembly is:

    \[
    \text{ldr} \quad w0, \ [x0, \ #32] \\
    \text{add} \quad x2, \ x0, \ x1
    \]
Compilation and Address Formation

- Compiler keeps frequently used data values in registers
  - In many architectures (including ARMv8-A), arithmetic operands must be a register, not a location in memory
  - Thus compiler keeps frequently accessed variables in registers
  - Stores everything else (including larger data structures and arrays) in main memory
- Address in memory = starting address + (index × element size)

```c
int A[] = { ... };
... A[8];
```

```assembly
1dr w0, [w0, #32]
```
Byte Addresses

• When operating on data that is not \textit{word aligned} within memory (e.g., an individual byte), need to specify which byte within the word to use

  • \textbf{Big-endian} (also known as network order): Motorola 68k, SuperH

  • \textbf{Little-endian}: x86-64

  • \textbf{Bi-endian}: can switch endianness as necessary

    • PowerPC starts in big-endian, can switch to little-endian

    • ARMv7 and ARMv8-A start in little-endian, can switch to big-endian
Example: to store the value 0x12345678 in memory starting at address 0x1000:

<table>
<thead>
<tr>
<th>memory cell</th>
<th>0x1000</th>
<th>0x1001</th>
<th>0x1002</th>
<th>0x1003</th>
</tr>
</thead>
<tbody>
<tr>
<td>big-endian</td>
<td>0x12</td>
<td>0x34</td>
<td>0x56</td>
<td>0x78</td>
</tr>
<tr>
<td>little-endian</td>
<td>0x78</td>
<td>0x56</td>
<td>0x34</td>
<td>0x12</td>
</tr>
</tbody>
</table>

Watch out for “mixed-endian” machines that may have other encoding schemes.

In ARMv8-A, `ldr` loads an entire word, while `ldrb` loads a single byte.
Stored Program Concept

- Modern computers are built on two key principles:
  - Instructions and data are represented as numbers (specifically, in binary)
  - Instructions and data are stored in memory as numbers

- Compiler transforms code into instructions (code generation), and then the assembler translates instructions (assembles) into underlying numeric representation for that instruction set architecture.

\[
f = (g + h) - (i + j)
\]

![Diagram showing the transformation of code and assembly instructions](image)
ARMv8-A Instruction Encoding

- In ARMv8-A, [most] instructions are encoded with exactly 32 bits (a fixed instruction length)

- x86-64 has variable instruction length

- Instruction encoding defines how instructions are numerically represented

- Example: register-type instructions are encoded as:

<table>
<thead>
<tr>
<th>R-Type</th>
<th>opcode</th>
<th>R_m</th>
<th>shamt</th>
<th>R_n</th>
<th>R_d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
</tr>
</tbody>
</table>

\[
\text{add w1, w1, w0 (0x0b000021)}
\]

```
000001011000 00000 000000 00001 00001
```

- **add**
- second operand = R0
- shift amount = 0
- first operand = R1
- dest register = R1
ARMv8-A Instruction Encoding

- Data transfers (loads and stores) use similar encoding
  - Offset must be from –256 to +255 (a range of $2^9$ bits) of base register

<table>
<thead>
<tr>
<th>D-Type</th>
<th>opcode</th>
<th>address</th>
<th>op2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11 bits</td>
<td>9 bits</td>
<td>2 bits</td>
<td>5 bits</td>
<td>5 bits</td>
</tr>
</tbody>
</table>

- Immediate instructions contain a direct value instead of a register number
  - Immediate value limited to 12 bits

<table>
<thead>
<tr>
<th>I-Type</th>
<th>opcode</th>
<th>immediate</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10 bits</td>
<td>12 bits</td>
<td>5 bits</td>
<td>5 bits</td>
</tr>
</tbody>
</table>
Branching

- Several ways to change program flow, like C’s `if` and `goto` statements
- Use `b` to unconditionally jump to a label
- Use `cbz` (compare and branch if zero) and `cbnz` (compare and branch if not zero) to conditionally jump to a label

```assembly
if (x == 0) 
  x = 10
else
  x += 20

else:
  add x0, x0, #20
end_if:
  /* end of if stmt */
  cbnz x0, else
  mov x0, #10
  b end_if
```
Condition Codes

• Many instructions can affect condition codes:

<table>
<thead>
<tr>
<th>condition bit</th>
<th>set when most recent instruction resulted in:</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero (Z)</td>
<td>equal</td>
</tr>
<tr>
<td>carry (C)</td>
<td>carry out of most significant bit</td>
</tr>
<tr>
<td>negative (N)</td>
<td>most significant bit was 1</td>
</tr>
<tr>
<td>overflow (V)</td>
<td>an overflow</td>
</tr>
</tbody>
</table>

• Use a combination of these codes to express full range of comparisons: less than (lt), greater than (gt), less than or equal (le), greater than or equal (ge), equal (eq), and not equal (ne)

• Example: b.lt to branch if most recent condition resulted in less than
Setting Condition Codes

• Many arithmetic instructions can have a `s` suffix to set condition code

• Can also directly compare a register to an immediate value via `cmp` instruction

• Example: calculate $3^x$ using a loop

```plaintext
result = 1;
while (x > 0) {
    result *= 3;
    x--;
}
```

```
start:
    mov    x0, #1
    mov    x2, #3
    cbz    x1, end

    mul    x0, x0, x2
    subs   x1, x1, #1
    b.gt   start

end:
    /* end of while */
```
Function Calls

• Calling a function depends upon the calling convention for the language, compiler, and instruction set

  • Need to agree on where store function arguments (register/stack/memory), where to store resulting value, and which registers are to be preserved by the caller and which by the callee

• When updating the calling stack, need to decide who updates the stack pointer (caller or callee) and by how much (prologue and epilogue)

  • For many architectures, the stack must be aligned to some boundary (every 4 bytes or every 8 bytes)
ARMv8-A Procedure Call Standard

• First eight arguments to function are stored in registers \texttt{x0} through \texttt{x7}
  
  • Use stack if more space is needed for incoming parameters

• \texttt{x0} holds the result of the function

• \texttt{x19} through \texttt{x29} are callee-saved

• \texttt{x30} is the link register

• \texttt{SP} is the stack register (special register 31)

• All other registers are caller-saved
Function Prologue and Epilogue

- Callee is responsible to moving stack pointer to hold local variables (prologue) and then cleaning up after itself afterwards (epilogue)

- Prologue and epilogue are added by the compiler; they are not hardware enforced

```c
static int __attribute__((noinline)) x(int g, int h, int i, int j) {
    int f = (g + h) - (i + j);
    return f;
}
```

```
sub   sp, sp, #0x20
str   w0, [sp, #12]
str   w1, [sp, #8]
str   w2, [sp, #4]
str   w3, [sp]
ldr   w1, [sp, #12]
ldr   w0, [sp, #8]
add   w1, w1, w0
ldr   w2, [sp, #4]
ldr   w0, [sp]
add   w0, w2, w0
sub   w0, w1, w0
str   w0, [sp, #28]
ldr   w0, [sp, #28]
add   sp, sp, #0x20
ret
```
Calling Functions

- Architectures vary on how to call and return from a function, specifically on where to store/load the return address
  
  - On x86-64, return address is pushed onto the stack upon `call; ret` instruction pops top-most value from stack and jumps to that address
  
  - On ARMv8-A, return address is stored in `x30` upon `bl; ret` instruction uses address in `x30` as target address
  
  - Callee is responsible for saving return address if it is not a leaf function

```
mov     x0, #0x0a
mov     x1, #0x14
mov     x2, #0x1e
mov     x3, #0x28
bl      some_func
some_func(10, 20, 30, 40);
```