CS1101: Lecture 33

The ISA Level: Data Types, Instruction Formats and Addressing

Dr. Barry O'Sullivan
b.osullivan@cs.ucc.ie

Course Homepage
http://www.cs.ucc.ie/~osullb/cs1101

Department of Computer Science, University College Cork

Lecture Outline

- Data Types
  - Numeric Data Types
  - Nonnumeric Data Types
- Instruction Formats
  - Examples of Instruction Formats
  - Instruction and Word Length
  - A Brief Look at Assembly Language
- Addressing
  - Immediate Addressing
  - Direct Addressing
  - Register Addressing
  - Register Indirect Addressing
  - Indexed Addressing
  - Based-Indexed Addressing
- **Reading**: Tanenbaum, Chapter 5, Section 2, 3 & 4.

Introduction

- At the ISA level, a variety of different data types are used to represent data.
- A key issue is whether or not there is hardware support for a particular data type.
- Hardware support means that one or more instructions expect data in a particular format, and the user is not free to pick a different format.
- Another issue is precision – what if we wanted to total the transactions on Bill Gates' deposit account? :)

Introduction

- Using 32-bit arithmetic would not work here because the numbers involved are larger than 2^{32} (about 4 billion).
- We could to use two 32-bit integers to represent each number, giving 64 bits in all.
- However, if the machine does not support this kind of double precision number, all arithmetic on them will have to be done in software, thus, without a required hardware representation.
- Today, we will look at data types are supported by the hardware, and thus for which specific formats are required.

Numeric Data Types

- Data types can be divided into two categories: numeric and nonnumeric.
- Chief among the numeric data types are the integers, which come in many lengths, typically 8, 16, 32, and 64 bits.
- Most modern computers store integers in two's complement binary notation.
- Some computers support **unsigned** integers as well as **signed** integers.
- For an unsigned integer, there is no sign bit and all the bits contain data – thus the range of a 32-bit word is 0 to $2^{32} - 1$, inclusive.
- In contrast, a two's complement signed 32-bit integer can only handle numbers up to $2^{31} - 1$, but it can also handle negative numbers.

Numeric Data Types

- For numbers that cannot be expressed as an integer, floating-point numbers are used.
- They have lengths of 32, 64, or sometimes 128 bits.
- Most computers have instructions for doing floating-point arithmetic.
- Many computers have separate registers for holding integer operands and for holding floating-point operands.

Nonnumeric Data Types

- Modern computers are often used for nonnumerical applications, such as word processing or database management.
- Thus, characters are clearly important here although not every computer provides hardware support for them.
- The most common character codes are ASCII and UNICODE.
- These support 7-bit characters and 16-bit characters, respectively.
- It is not uncommon for the ISA level to have special instructions that are intended for handling character strings.
- The instructions can perform copy, search, edit and other functions on the strings.

Nonnumeric Data Types

- Boolean values are also important.
- Two values: TRUE or FALSE.
- In theory, a single bit can represent a Boolean, with 0 as false and 1 as true (or vice versa).
- In practice, a byte or word is used per Boolean value because individual bits in a byte do not have their own addresses and thus are hard to access.
- A common system uses the convention that 0 means false and everything else means true.

Nonnumeric Data Types

- Our last data type is the pointer, which is just a machine address.
- We have already seen pointers.
- When we discussed stacks we came across pointers SP and LV.
- Accessing a variable at a fixed distance from a pointer, which is the way **ILOAD** works, is extremely common on all machines.

Instruction Formats

- An instruction consists of an opcode, usually along with some additional information such as where operands come from, and where results go to.
- The general subject of specifying where the operands are (i.e., their addresses) is called **addressing**.
- Instructions always have an opcode to tell what the instruction does.
- There can be zero, one, two, or three addresses present.

Examples of Instruction Formats

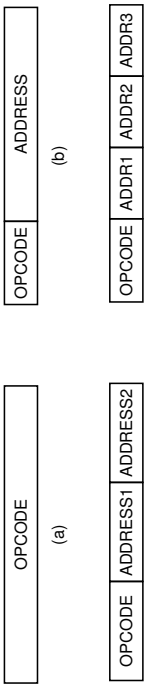

Figure 5-9. Four common instruction formats: (a) Zero-address instruction. (b) One-address instruction (c) Two-address instruction. (d) Three-address instruction.

Instruction Formats

- On some machines, all instructions have the same length; on others there may be many different lengths.
- Instructions may be shorter than, the same length as, or longer than the word length.
- Having all the instructions be the same length is simpler and makes decoding easier but often wastes space, since all instructions then have to be as long as the longest one.

Instruction and Word Length

Figure 5-10. Some possible relationships between instruction and word length.

A Brief Look at Assembly Language

- Here is a Java code fragment:

```
if (i == 0)
    k = 1;
else
    k = 2;
```

- Its translation to a generic assembly language:

```
        CMP i,0    ; compare i to 0
        BNE Else   ; branch to Else is not equal
Then:   MOV k,1    ; move 1 to k
        BR  Next   ; unconditional branch to Next
Else:   MOV k,2    ; move 2 to k
Next:
```


Addressing

- Instructions generally have one, two or three operands.
- The operands are addressed using one of the following modes:
  - Immediate
  - Direct
  - Register
  - Indexed
  - Other mode
- Some machines have a large number of complex addressing modes.
- We will consider a few addressing modes here.

Immediate Addressing

- The simplest way for an instruction to specify an operand is for the address part of the instruction actually to contain the operand itself rather than an address or other information describing where the operand is.
- Such an operand is called an **immediate operand** because it is automatically fetched from memory at the same time the instruction itself is fetched.
- Example:

```
MOV R1,4
```

- Advantage – no extra memory reference to fetch the operand.
- Disadvantage – only a constant can be supplied this way.

Direct Addressing

- A method for specifying an operand in memory is just to give its full address.
- This mode is called **direct addressing**.
- Like immediate addressing, direct addressing is restricted in its use: the instruction will always access exactly the same memory location.
- So while the value can change, the location cannot.
- Thus direct addressing can only be used to access **global variables** whose address is known at compile time.

Register Addressing

- Register addressing is conceptually the same as direct addressing but specifies a register instead of a memory location.
- Because registers are so important (due to fast access and short addresses) this addressing mode is the most common one on most computers.
- Many compilers go to great lengths to determine which variables will be accessed most often (for example, a loop index) and put these variables in registers.
- This addressing mode is known simply as **register mode**.

Register Indirect Addressing

- In this mode, the operand being specified comes from memory or goes to memory, but its address is not hardwored into the instruction, as in direct addressing.
- Instead, the address is contained in a register.
- When an address is used in this manner, it is called a **pointer**.
- A big advantage of register indirect addressing is that it can reference memory without paying the price of having a full memory address in the instruction.

Example of Register Indirect Addressing

- Consider an program which steps through the elements of a 1024-element one-dimensional integer array to compute the sum of the elements in register R1.
- We will indirectly register through R2 to access the elements of the array
- Here is the assembly program:

Example of Register Indirect Addressing

```
        MOV R1,#0       ; accumulate the sum in R1, initially 0
        MOV R2,#A       ; R2 = address of the array A
        MOV R3,#A+4096  ; R3 = address of the first word beyond A
LOOP:   ADD R1,(R2)     ; register indirect through R2 to get operand
        ADD R2,#4       ; increment R2 by one word (4 bytes)
        CMP R2,R3       ; are we done yet?
        BLT LOOP        ; if R2 < R3, we are not done, so continue
```


An Example of Indexed Addressing

```
        MOV R1,#0       ; accumulate the OR in R1, initially 0
        MOV R2,#0       ; R2 = index, i, of current product: A[i] AND B[i]
        MOV R3,#4096    ; R3 = first index not to use
LOOP:   MOV R4,A(R2)    ; R4 = A[i]
        AND R4,B(R2)    ; R4 = A[i] AND B[i]
        OR  R1,R4       ; OR all the Boolean products in R1
        ADD R2,#4       ; i = i + 4 (step through units of 1 word = 4 bytes)
        CMP R2,R3       ; are we done yet?
        BLT LOOP        ; if R2 < R3, we are not done, so continue
```


Indexed Addressing

- It is frequently useful to be able to reference memory words at a known offset from a register.
- Addressing memory by giving a register (explicit or implicit) plus a constant offset is called **indexed addressing**.
- **Example:** consider the following calculation:
- We have two one-dimensional arrays of 1024 words each, A and B, and we wish to compute A_i AND B_i for all the pairs and then OR these 1024 Boolean products together to see if there is at least one nonzero pair in the set.
- Here is the assembly program.

Based-Indexed Addressing

- Some machines have an addressing mode in which the memory address is computed by adding up two registers plus an (optional) offset.
- Sometimes this mode is called **based-indexed addressing**.
- One of the registers is the **base** and the other is the **index**.
- Such a mode would have been useful in our example here.
- Outside the loop we could have put the address of A in R5 and the address of B in R6.
- Then we could have replaced the instruction at LOOP and its successor with

```
LOOP: MOV R4,(R2+R5)
      AND R4,(R2+R6)
```