Computer Systems 1
Lecture 9

The Stored Program Computer

Dr. John T. O'Donnell
School of Computing Science
University of Glasgow

Topics

Machine language: representing instructions in memory

- The actual bits representing an instruction (written in hex) — 0d69 – is machine language
- The actual hardware runs the machine language — it's just looking at the numbers
- The text notation with names — add R13,R6,R9 — is assembly language
- Assembly language is for humans, machine language is for machines
- Both specify the program in complete detail, down to the last bit

What's in the memory?

- All your program's data
  - Variables
  - Data structures, arrays, lists
- And also the machine language program itself!

The stored program computer

The program is stored inside the computer's main memory, along with the data.

An alternative approach is to have a separate memory to hold the program, but experience has shown that to be inferior for general purpose computers. (Special-purpose computers often do this.)

Instruction formats: different types of instruction

- Sigma16 has three kinds of instruction:
  - RRR instructions use the registers
  - RX instructions use the memory
  - EXP instructions use registers and constant
- Each kind of instruction is called an instruction format
- All the instructions with the same format are similar
- Each instruction format has a standard representation in the memory.

Instruction formats: representing instructions

- The machine language program is in the memory
- So we need to represent each instruction as a word
- An instruction format is a systematic way to represent an instruction using a string of bits, on one or more words
- Every instruction is either RRR, RX, or EXP
  - An RRR instruction is represented in one word (remember, a word is 16 bits)
  - An RX or EXP instruction is represented in two words
- We just need to learn three ways to represent an instruction!
- For now, we just need RRR and RX (EXP is needed only for some more advanced instructions, which we'll see later)

Fields of an instruction word

- An instruction word has 16 bits
- There are four fields, each 4 bits
- We write the value in a field using hexadecimal
  1. hex digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f
  2. represent numbers 1, . . ., 15
- The fields have standard names:
  - op — holds the operation code
  - d — usually holds the destination register
  - a — usually holds the first source operand register
  - b — usually holds the second source operand register

RRR instructions

- Every RRR instruction consists of
    - An operation (e.g. add)
    - Three register operands: a destination and two operands
    - The instruction performs the operation on the operands and puts the result in the destination

Representing RRR

- Example: add R3,R12,R5
- We need to specify which RRR instruction this is. Is it add? sub? mul? another?
- This is done with an operation code — a number that says what the operation is
- There are about a dozen RRR instructions, so a 4-bit operation code suffices
- We also need to specify three registers: destination and two source operands
- There are 16 registers, so a particular one can be specified by 4 bits
- Total requirements: 4 fields, each 4 bits — total 16 bits
- An RRR instruction exactly fills one word

Some RRR instructions

- All RRR instructions have the same form, just the operation differs
  - add R2,R2,R5 ; R2 = R2 + R5
  - sub R3,R1,R3 ; R3 = R1 - R3
  - mul R8,R6,R7 ; R8 = R6 * R7
- In "add R2,R5,R9" we call R5 the first operand, R9 the second operand, and R2 the destination
- It's ok to use the same register as an operand and destination!
- Later we'll see some more RRR instructions, but they all have the same form as these do

A few RRR operation codes

mnemonic	operation code
add	0
sub	1
mul	2
div	3
⋮	⋮
trap	d

Don't memorise this table! You just need to understand how it's used.

Example of RRR

```
add  R13,R6,R9
```

- Since each field of the instruction is 4 bits, written as a hex digit
- The opcode (operation code) is 0
- Destination register is 13 (hex d)
- Source operand registers are 6 and 9 (hex 6 and 9)
- So the instruction is:

$$\boxed{\text{0d69}}$$

RX instructions

- Every RX instruction contains two operands:
  - A register
  - A memory location
- We have seen several so far:
  - lea R5,19[R0] ; R5 = 19
  - load R1,x[R0] ; R1 = x
  - store R3,z[R0] ; z = R3
  - jump finished[R0] ; goto finished

RX instructions

A typical RX instruction: load R1,x[R0]

- The first operand (e.g. R1 here) is called the *destination register*, just like for RRR instructions
- The second operand x[R0] specifies a memory address.
- Each variable is kept in memory at a specific location: we talk about the address of a variable
- The memory operand has two parts:
  - The variable x is a name for the address where x is kept — called the displacement
  - The R0 part is just a register, called the index register

Format of RX instruction

```
load R1,x[R0]
```

- There are two words in the machine language code
- The first word has 4 fields: op, d, a, b
  - op contains f for every RX instruction
  - d contains the register operand (in the example, 1)
  - a contains the index register (in the example, 0)
  - b contains a code indicating *which* RX instruction this is (1 means load)
- The second word contains the displacement (address) (in the example, the address of x)

Suppose x has memory address 0008. Then the machine code for load R1,x[R0] is:

```
f101
0008
```

Operation codes for RX instructions

- Recall, for RRR the op field contains a number saying *which* RRR instruction it is
- For RX, the op field *always contains f*
- So how does the machine know *which* RX instruction it is?
- Answer: there is a secondary code in the b field

mnemonic	RX operation code (in b field)
lea	0
load	1
store	2
⋮	⋮

Assembly language

- Humans write assembly language
  - The program is text: add R4,R2,R12
  - It's easier to read
  - You don't need to remember all the codes
  - Memory addresses are *much easier* to handle
- The machine executes machine language
  - The program is words containing 16-bit numbers: 042c
  - It's possible for a digital circuit (the computer) to execute
  - No names for instructions or variables: everything is a number

The assembler

- A human writes a machine-level program in assembly language
- A software application called the assembler reads it in, and translates it to machine language
- What does the assembler do?
  - When it sees an instruction mnemonic like add or div, it replaces it with the operation code (0, 3, or whatever).
  - The assembler helps with variable names — the machine language needs addresses (numbers) and the assembler calculates them

Assembly language

- Each statement corresponds to one instruction
- You can use names (add, div) rather than numeric codes (0, 3)
- You can use variable names (x, y, sum) rather than memory addresses (02c3, 18d2)
- You write a program in assemply language
- The *assembler* translates it into machine language
- What's the relationship between compilers and assemblers?
    - Compilers translate between languages that are very different
    - Assemblers translate between very similar languages

A sequence of RRR instructions

Assembly language

```
add     R3,R5,R1
sub     R4,R2,R3
mul     R1,R9,R10
```

Run the assembler. . .

Machine language

```
0351
1423
219a
```

Variable names and addresses

- Each variable needs to be declared with a data statement
- x data 23
- This means: allocate a word in memory for x and initialize it to 23
- The data statements should come after the trap instruction that terminates the program

Instructions in assembly language

- The syntax is simple, but you have to follow the form of the instructions exactly!
- RRR instructions
  - Typical example: add R8,R2,R12
  - R8 is the destination (where the result goes)
  - R2 and R12 are the sources (the operands to be added)
- RX instructions
  - RX instructions specify a register and a memory location
  - Typical example: load R3,x[R0]
  - Meaning of load: R3 = x
  - store R3,y[R0]
  - Meaning of store: y = R3
  - load copies from memory to register
  - store copies from register to memory

How the assembler allocates memory

1. The assembler maintains a variable called the location counter. This is the address where it will place the next piece of code.
2. Initially the location counter is 0.
3. The assembler reads through each line of code
    1. If there is a label, it remembers that the value of the label is the current value of the location counter. This goes into the symbol table
    2. The assembler decides how many words of memory this line of assembly code will require (add needs one word, load needs two), and adds this to the location counter.
4. Then the assembler reads through the assembly language program *a second time*
5. Now it generates the words of object code for each statement. If there is a reference to a label that appears farther on (e.g. load x, or jump loop) it looks up the value of the label in the symbol table.

Program structure

- A complete program needs
  - Good comments explaining what it is
  - The actual program — a sequence of instructions
  - An instruction to stop the program: trap R0,R0,R0
  - Declarations of the variables: data statements
- Why do we put the instructions first, and define the variables at the end?
  - The assembler can find the definitions because it reads the program twice: the first pass finds all the labels, the second pass generates the machine language code
  - The computer will start executing at memory address 0, so there had better be in instruction there, not data!

Example program Add

```
; Program Add
; A minimal program that adds two integer variables

; Execution starts at location 0, where the first instruction will be
; placed when the program is executed.

      load   R1,x[R0]    ; R1 := x
      load   R2,y[R0]    ; R2 := y
      add    R3,R1,R2    ; R3 := x + y
      store  R3,z[R0]    ; z := x + y
      trap   R0,R0,R0    ; terminate

; Static variables are placed in memory after the program

x     data  23
y     data  14
z     data  99
```

Snapshot of memory: example program Add

address	contents	what the contents mean
0000	f101	first word of load R1,x[R0]
0001	0008	second word: address of x
0002	f201	first word of load R2,y[R0]
0003	0009	second word: address of y
0004	0312	add R3,R1,R2
0005	f302	first word of store R3,z[R0]
0006	000a	second word: address of z
0007	d000	trap R0,R0,R0
0008	0017	x = 23
0009	000e	y = 14
000a	0063	z = 99

Boot: reading in the program

- The program is placed in memory starting at location 0
- The program should finish by executing the instruction "trap R0,R0,R0"
- Normally, trap R0,R0,R0 should be the last instruction of the program (i.e. the program begins execution with the first instruction, and ends execution with the last, although it may jump around during execution)
- After the trap R0,R0,R0 come the data statements, which tell the assembler the names of the variables and their initial values
- These conventions were typical for early computers; later we will discuss how the operating system interacts with user programs

Control registers

- Some of the registers in the computer are accessible to the programmer: R0, R1, R2, . . ., R15
- There are several more registers that the machine uses to keep track of what it's doing
- These are called "control registers"
- They are (mostly) invisible to the program

Keeping track of where you are

- When you “hand execute” a program, you need to know
  - Where you are (point a finger at the current instruction)
  - What you’re doing (read the current instruction)
- *The computer needs to know this too!*
- The PC register (“program counter”) contains the address of the next instruction to be executed
- The IR (“instruction register”) contains the instruction being executed right now
- If an RX instruction is being executed, the ADR (“address register”) contains the memory address of the second operand.

Following PC and IR control registers

- Try running a simple program
- Step through the execution
- Before each instruction executes, look at the PC and IR registers
- Notice that PC always contains the address of the next instruction and IR always contains the current instruction
- The control registers help to understand in detail what the machine is doing.

- The syntax of assembly language is simple and rigid
- See the document Sigma16 Programming Reference, where you will find these notes and additional tips and techniques

Fields separated by spaces

- An assembly language statement has four fields, separated by space
    - label (optional) – if present, must begin in leftmost character
    - operation load, add, etc.
    - operands: R1,R2,R3 or R1,x[R0]
    - comments: ; x = 2 * (a+b)
- There cannot be any spaces inside a field
    - R1,R12,R5 is ok
    - R1, R12,R5 is wrong

```
loop    load    R1,count[R0]      ; R1 = count
        add     R1,R1,R2          ; R1 = R1 + 1
```

The assember first breaks each statement into the four fields; then it looks at the operation and operands.

Correct form of operand field

- RRR
  - Exactly three registers separated by commas
  - Example: R8,R13,R0
- RX
  - Two operands: first is a register, second is an address
  - Address is a name or constant followed by [register]
  - Example: R12,array[R6]

Each of these statements is wrong!

```
   add   R2, R8, R9      Spaces in the operand field
   store x[R0],R5        First operand must be register, second is address
 loop load R1,x[R0]      Space before the label
   jumpt R6,loop         Need register after address:  loop[R0]
   jal   R14, fcn[R0]    Space in operand field
```

If you forget some detail, look at one of the example programs

Writing constants

- In assembly language, you can write constants in either decimal or hexadecimal
  - decimal: 50
  - hexadecimal: $0032

Examples:

```
   lea    R1,40[R0]       ; R1 = 40
   lea    R2,$ffff[R0]    ; R2 = -1

x  data   25
y  data   $2c9e
```

Good style

- It isn't enough just to get the assembler to accept your program without error messages
- Your program should be clear and easy to read
- This requires good style
- Good style saves time writing the program and getting it to work
- A sloppy program looks unprofessional

Comments

- In Sigma16, a semicolon ; indicates that the rest of the line is a comment
- You can have a full line comment: just put ; at the beginning
- You should use good comments in all programs, regardless of language
- But they are even more important in machine language, because the code needs more explanation
- At the beginning of the program, use comments to give the name of the program and to say what it does
- Use a comment on every instruction to explain what it's doing

Indent your code consistently

Each field should be lined up vertically, like this:

```
    load   R1,three[R0]  ; R1 = 3
    load   R2,x[R0]      ; R2 = x
    mul    R3,R1,R2      ; R3 = 3*x
    store  R3,y[R0]      ; y = 3*x
    trap   R0,R0,R0      ; stop the program
```

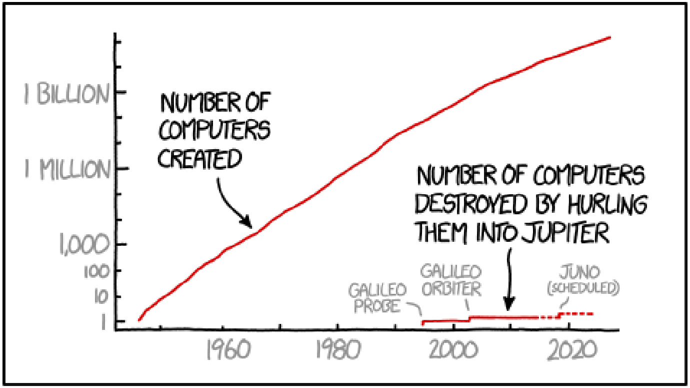
Not like this:

```
    load   R1,three[R0]     ; R1 = 3
  load  R2,x[R0] ; R2 = x
       mul R3,R1,R2           ; R3 = 3*x
 store         R3,y[R0]      ; y = 3*x
   trap  R0,R0,R0      ; stop the program
```

The exact number of spaces each field is indented isn't important; what's important is to make the program neat and readable.

Use spaces, not tabs

- To indent your code, always use spaces
- Don't use tabs!
- In general, never use tabs except in the (rare) cases they are actually required
    - The tab character was introduced to try to mimic the tab key on old mechanical typewriters
    - But software does not handle tab consistently
    - If you use tabs, your can look good in one application and a mess in another
- It's easy to indent with spaces, and it works everywhere!



NASA NEEDS TO PICK UP THE PACE IF THEY EVER WANT TO FINISH THE JOB.

https://xkcd.com/1727/