COMP124

Computer Systems

04 | Loops | Arrays | Subroutines

Dr Stuart Thomason

Loops

- We can loop (iterate) over instructions by jumping backwards in the code
  - Jump instruction moves the instruction pointer back to an earlier point
  - But we need to ensure we don’t end up in an infinite loop

- Loops need to have terminating conditions (just as they do in high level code)
  - They iterate <u>while</u> a certain condition is true
  - Or <u>until</u> a certain condition is true

While Loop

- A loop to add up all the Fibonacci numbers until the total reaches (or exceeds) 1000

  1 1 2 3 5 8 13 21 34 55 89 144 …

```
// assume vars are defined
while (fib2 < 1000) {
    fib0 = fib1;
    fib1 = fib2;
    fib2 = fib1 + fib0;
}
```

```
begin: mov eax, fib2
       cmp eax, 1000
       jge endwh
       mov eax, fib1
       mov fib0, eax
       mov eax, fib2
       mov fib1, eax
       add eax, fib0
       mov fib2, eax
       jmp begin
endwh: nop
```

Do-While Loop

- The do-while loop always has at least one iteration and then tests at the end of the loop

  1 1 2 3 5 8 13 21 34 55 89 144 …

```
// assume vars are defined
do {
    fib0 = fib1;
    fib1 = fib2;
    fib2 = fib1 + fib0;
} while (fib2 < 1000);
```

```
begin: mov eax, fib1
       mov fib0, eax
       mov eax, fib2
       mov fib1, eax
       add eax, fib0
       mov fib2, eax
       cmp eax, 1000
       jl begin
endwh: nop
```

For Loop

- We use a for loop when we know in advance how many iterations we need
- For example, sum all the integers from 1 to 10 inclusive

```
int sum = 0;
for (int x = 1; x <= 10; x++) {
    sum = sum + x;
}
```

```
       mov eax, 1      ; use accumulator to count iterations
floop: add sum, eax    ; update sum
       inc eax         ; increase loop counter
       cmp eax, 10     ; have we reached the end?
       jle floop       ; if not, go round again
```

Second Attempt

- Sometimes we can be more efficient if we reverse the way the algorithm works
- Summing “10 to 1” is the same as summing “1 to 10” and uses fewer instructions

```
       ; first attempt                 ; second attempt
       mov eax, 1                      mov eax, 10
floop: add sum, eax             floop: add sum, eax
       inc eax                         dec eax
       cmp eax, 10                     jnz floop
       jle floop
```

- We no longer need a comparison instruction because we can directly test the zero flag to see if the previous instruction had a zero result

ECX and the Loop Instruction

- The counter register (ECX) is often used in loops (as the name suggests)
- The loop instruction performs a jump based on the value of ECX
  - First it decrements ECX (subtracts one)
  - Then it jumps to the given label if ECX is not zero
- This allows us to make our code even more efficient by saving another instruction

```
       ; third attempt
       mov ecx, 10
floop: add sum, ecx
       loop floop
```

- However, there is a trade off between efficiency and readability
- One reason why we prefer to write high level code in a human-readable language
- A good (modern) compiler will generate optimised low level code for us

Labels & Memory Addresses

- Remember that a label just points to a memory address
- In our C++ code, we declare a variable name and optionally give it a value
  `int age = 21;`
- And in our assembly code we can use the label to refer to the variable in memory
  `mov eax, age`

- But sometimes we want to get the memory address of the variable, not its value (load effective address)
  `lea ebx, age`
- And if we have a memory address stored in a register, we can use register indirect mode to get the value stored in that location
  `mov eax, [ebx]`

Arrays

- Arrays are just items stored in consecutive memory locations
- The amount of memory depends on the data being stored in the array
- In a 32-bit system, each integer takes up four bytes of memory

```
int grades[4] = { 64, 78, 60, 55 };
```

0	0	0	64	0	0	0	78	0	0	0	60	0	0	0	55
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- We can find out the memory address of the array

```
lea ebx, grades
```

- To get the value stored in the second array item, we need to add four to the address

```
add ebx, 4
mov eax, [ebx]
```

Array Processing

- We can put all this knowledge together to loop through an array and sum its contents
- In the C++ part of our code we define the array with four items

```
int grades[4] = { 64, 78, 60, 55 };
```

- In the assembly code we set up the loop

```
       lea ebx, grades
       mov ecx, 4
       mov eax, 0
floop: add eax, [ebx]
       add ebx, 4
       loop floop
```

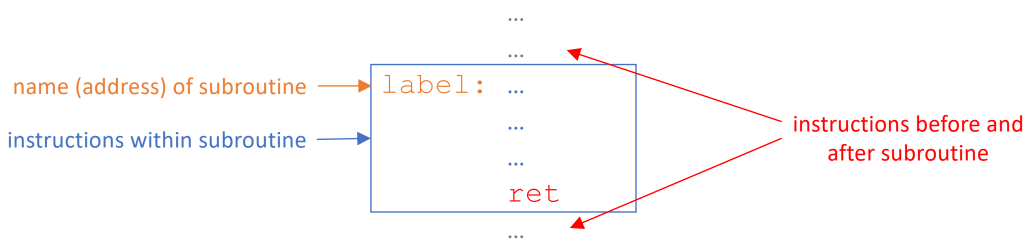
- This uses three registers
  - EAX – stores the sum as we go along
  - EBX – stores the memory address of the current item in the array
  - ECX – used as the loop counter

Subroutines

- High level languages provide a way to call certain lines of code with parameters
  - Saves effort in programming
  - Reduces the size of programs
  - Shares code
  - Encapsulates or packages a specific activity
  - Provides easy access to tried and tested code
- Various names depending on which language (and text book) you use
  - Procedures
  - Methods
  - Functions
- We can define subroutines in assembly code
  - Parameter passing is tricky (we will look at this in the next lecture)
  - Subroutine calls just change the instruction pointer (but how do we get back?)

Assembly Subroutines

- There is no special syntax or meaning around subroutines in assembly code
  - No fancy way to specify parameters and their types
  - No local registers (ie. all code shares the same registers)
  - Simple jump to a label (address) that points to the first instruction in the subroutine
  - Use the call instruction with the label of the subroutine
  - Use the ret instruction to return from the subroutine



  - If you forget to return, the instruction pointer would just carry on to next instruction

Return Address

- When a subroutine is called, the instruction pointer is changed to its address
  - Fetch-execute cycle continues with instructions from that point onward
  - The ret instruction changes the instruction pointer back to the address following the original call instruction
  - So the CPU needs a way to remember where to return to
- What problems can you see with this approach? (Think about it for the next lecture)