Inline Assembly Code

Programming Fundamentals

Dr Kevin Chalmers

School of Computing
Edinburgh Napier University
Edinburgh

k.chalmers@napier.ac.uk

Notes

Outline

1. Computer Architecture
2. Assembly Languages
3. The Stack
4. Inline Assembly in C
5. Summary

Notes

Outline

1. Computer Architecture
2. Assembly Languages
3. The Stack
4. Inline Assembly in C
5. Summary

Notes

Computer Architecture

- A simplistic architecture:
  - Central Processing Unit (CPU)
  - Memory
  - Input-Output systems (I/O)
- These components are connected by a number of buses:
  - A data bus
  - An address bus
  - A control bus

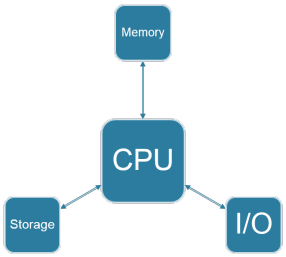


Figure: Simplistic Computer Architecture

Notes

The CPU

- The CPU is the beating heart of our computer system.
- It is the electronic circuit in charge of control, logic, arithmetic and input/output operations.
- Early CPUs could be the size of a cabinet
- Today they are in the form of a miniaturised microprocessor.

Notes

Early Example

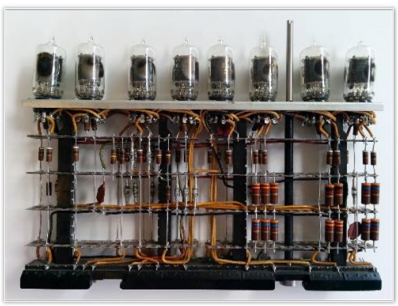


Figure: This is the logical unit of an early IBM 700 series mainframe in the 1950s. You can clearly see the apparent transistors and wires. Wikimedia Commons

Notes

Modern Example

Figure: An Intel Core i7. Today most CPUs are in the form of a miniaturised, integrated microprocessor. Older and newer CPUs share an architecture in common - *the Von Neumann architecture*

Notes

John Von Neumann

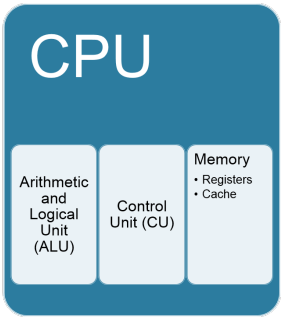
- Born in Hungary in 1903, died in Washington D.C. in 1957
- Mathematician at the University of Berlin, then Princeton
- Worked on early atomic bombs and early computers (ENIAC)

Figure: John Von Neumann
Wikimedia Commons

Notes

The Von Neumann Architecture

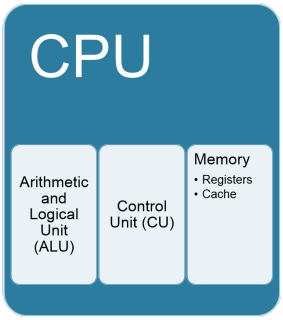
- Most CPUs are inherited from the Von Neumann architecture.
- The Arithmetic and Logical Unit (**ALU**) is in charge of arithmetic and bitwise operations.
- The Control Unit (**CU**) is in charge of reading and writing in memory and directing the ALU.



Notes

The Von Neumann Architecture

- Processors have a (small) amount of memory, independent from the computer's main memory - *registers and caches*.
- The architecture evolved over time, now CPUs can be supported by floating-point units (FPU), graphical processing units (GPU), multiple cores, etc.



Notes

Questions?

Notes

Outline

1 Computer Architecture
2 Assembly Languages
3 The Stack
4 Inline Assembly in C
5 Summary

Notes

Instruction Sets

- Processors are shipped with a set of instructions and the documentation to use them.
- There are two main families of instruction sets:

  Reduced Instruction Set Computing (RISC) where only basic low level instructions are provided. e.g. SPARC, ARM, POWER. ARM is widely used in mobile devices.

  Complex Instruction Set Computing (CISC) where higher level instructions are also provided. The Intel / AMD x86 and x64 architectures are the most common on desktop and laptop systems.
- Theses instructions are executed in machine code

Notes

Assembly Languages

- Assembly languages are low-level programming languages
- Their instructions correspond with their corresponding architecture's instruction set.
  - Therefore, an assembly language is not portable, unlike a higher-level language like C.
- Unlike machine code, assembly languages allow the manipulation of data in a human-readable form: strings, decimal numbers etc.
- There are two common syntaxes when working in x86 / x64 assembly:

  Intel uses an `instruction destination, source` style. Used by Microsoft tools and available in Linux tools

  AT & T uses an `instruction source, destination` style. Default for GNU based assembly normally used in Linux
- There are slight variations between the two. We will look at Intel syntax here.

Notes

Intel Assembly Instructions

Instruction	Parameters	Description
`mov`	`destination, source`	Moves the value stored in `source` to the `destination`. *At least one of the two parameters must be a register.*
`add`	`destination, source`	Adds the value `source` to the value `destination`. The result is stored in `destination`. `destination` *must be a register.*
`push`	`source`	Pushes the value stored in `source` onto the local stack.
`pop`	`destination`	Pops a value from the local stack into the `destination`.

Notes

Intel Assembly Instructions

Instruction	Parameters	Description
`call`	`name`	Executes the given procedure
`jmp`	`location`	Causes control to jump to the location given. Jumping allows us to implement branching instructions such as `if` and `while`
`cmp`	`value1, value2`	Compares the two values. Sets relevant flags on the CPU based on the outcome of the comparison. Allows conditional jumping.
`jge`	`location`	A jump instruction that jumps if the result of a comparison set the *greater than* flag or the *equal to* flag.

Notes

Exercise

Notes

Questions?

Notes

Outline

1. Computer Architecture
2. Assembly Languages
3. The Stack
4. Inline Assembly in C
5. Summary

Notes

Stack and Heap

- The memory available to us when executing an application is divided into sub-zones among which are *the stack* and *the heap*.
- The stack is a limited space of memory managing routine calls and automatic variables. Values are pushed to the stack, or popped from the stack, using a *Last-In-First-Out* (**FIFO**) basis. The stack is controlled by the scope of values.
- The heap is the rest of the addressable memory. We will study the heap further in the pointers lecture.
- Up until now, you have generally been working in the stack (to keep it simple).

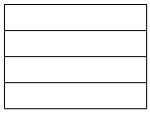
Notes

Inner Workings of the Stack

- Here is an example of multiple scopes and variables:

Scope

```
void func()
{
    int x = 500;
    int y = 1000;
    {
        int z = x + y;
    }
    // Outside inner scope
}
// Outside func()
```



Notes

Inner Workings of the Stack

- Here is an example of multiple scopes and variables:

Scope

```
void func()
{
    int x = 500;
    int y = 1000;
    {
        int z = x + y;
    }
    // Outside inner scope
}
// Outside func()
```

500

Notes

Inner Workings of the Stack

- Here is an example of multiple scopes and variables:

Scope

```
void func()
{
    int x = 500;
    int y = 1000;
    {
        int z = x + y;
    }
    // Outside inner scope
}
// Outside func()
```

1000
500

Notes

Inner Workings of the Stack

- Here is an example of multiple scopes and variables:

Scope

```
void func()
{
    int x = 500;
    int y = 1000;
    {
        int z = x + y;
    }
    // Outside inner scope
}
// Outside func()
```

1500
1000
500

Notes

Inner Workings of the Stack

- Here is an example of multiple scopes and variables:

Scope

```
void func()
{
    int x = 500;
    int y = 1000;
    {
        int z = x + y;
    }
    // Outside inner scope
}
// Outside func()
```

1000
500

Notes

Inner Workings of the Stack

- Here is an example of multiple scopes and variables:

Scope

```
void func()
{
    int x = 500;
    int y = 1000;
    {
        int z = x + y;
    }
    // Outside inner scope
}
// Outside func()
```


Notes

Questions?

Notes

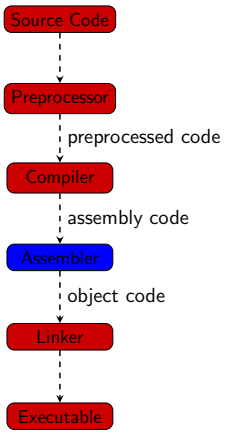
Outline

1. Computer Architecture
2. Assembly Languages
3. The Stack
4. Inline Assembly in C
5. Summary

Notes

Inline Assembly

- Remember that creating an executable using C goes through multiple stages
- The *compilation* stage translates C code into Assembly code, that will be translated to machine code at a later stage.
- Because of this, it is fairly straightforward to include assembly code inside a C source file.



Notes

Inline Assembly

- Embedding inline assembly code in a C file differs across compilers. We will use Microsoft as an example.
- Using Microsoft compilers, we would use the following syntax:

Inline Assembly using `cl`

```
__asm
{
    // assembly code here...
}
```

- On GCC and clang, the following syntax is in use:

Inline Assembly using GNU

```
__asm__("line 1"
        "line 2"
        "etc");
```

Notes

Example - Copying a Value

Copying a Value Using Assembly Code

```
int x = 500;
int y = 0;
__asm
{
    mov eax, x
    mov y, eax
}
```

- The two lines of assembly code opposite do the following:
  1. Take the value of x (500) and load it into the eax register (mov)
  2. Take the value of the eax register and load it into y
- The final result is equivalent to `y = x;`
- This is similar to how the compiler would map your C code to ASM.

Notes

Example - Swapping Values

- There are two commands in x86 / x64 assemblt to work with the stack:
  - `push` *pushes* (stores) a value on the stack
  - `pop` *pops* (removes) a value from the stack
- The following example will show how to swap the values of two variables without creating a third variable using the stack.

Notes

Example - Swapping Values

Swapping Values using the Stack

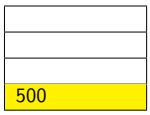
```
int x = 500;
int y = 200;
__asm
{
    // Push x onto stack
    push x
    // Push y onto stack
    push y
    // Pop stack into x
    pop x
    // Pop stack into y
    pop y
}
```

Notes

Example - Swapping Values

Swapping Values using the Stack

```
int x = 500;
int y = 200;
__asm
{
    // Push x onto stack
    push x
    // Push y onto stack
    push y
    // Pop stack into x
    pop x
    // Pop stack into y
    pop y
}
```

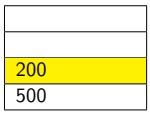


Notes

Example - Swapping Values

Swapping Values using the Stack

```
int x = 500;
int y = 200;
__asm
{
    // Push x onto stack
    push x
    // Push y onto stack
    push y
    // Pop stack into x
    pop x
    // Pop stack into y
    pop y
}
```

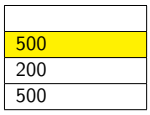


Notes

Example - Swapping Values

Swapping Values using the Stack

```
int x = 500;
int y = 200;
__asm
{
    // Push x onto stack
    push x
    // Push y onto stack
    push y
    // Pop stack into x
    pop x
    // Pop stack into y
    pop y
}
```



Notes

Example - Swapping Values

Swapping Values using the Stack

```
int x = 500;
int y = 200;
__asm
{
    // Push x onto stack
    push x
    // Push y onto stack
    push y
    // Pop stack into x
    pop x
    // Pop stack into y
    pop y
}
```

200
500
200
500

Notes

Example - Swapping Values

Swapping Values using the Stack

```
int x = 500;
int y = 200;
__asm
{
    // Push x onto stack
    push x
    // Push y onto stack
    push y
    // Pop stack into x
    pop x
    // Pop stack into y
    pop y
}
```

500
200
200

Notes

Example - Swapping Values

Swapping Values using the Stack

```
int x = 500;
int y = 200;
__asm
{
    // Push x onto stack
    push x
    // Push y onto stack
    push y
    // Pop stack into x
    pop x
    // Pop stack into y
    pop y
}
```

500
200

Notes

Example - Calling a Function

- In x86/x64 ASM, we can call standard C functions using the `call` instruction.
- Passing parameters to the function must happen in the following way:
  1. Set the stack with relevant parameter(s) (in reverse order)
  2. Call the procedure
  3. Clean the stack of any parameters

Notes

Example - Calling a Function

Calling a Function in Assembly

```
char *message = "Hello,
    World\n";
__asm
{
    push message;
    call printf;
    pop ebx;
}
```

- As an example, consider calling the `printf` function:
  1. Push the "Hello World" string onto the stack
  2. Call the `printf` procedure
  3. Pop the stack into the `ebx` register (clean-up)
- This is the same as calling `printf("Hello World\n");` in C.

Notes

Example - Returning a Value

Getting a Return Value in Assembly

```
int sub(int x, int y)
{
    return x     y;
}

int main()
{
    int result = 0;
    __asm
    {
        push 500
        push 200
        call sub
        mov result, eax
        pop ebx
        pop ebx
    }
    // result is now -300
}
```

- Return values from functions (and from applications) are stored in the eax register
- As an example consider the subtraction application:
  1. Push 500 onto the stack
  2. Push 200 onto the stack
  3. Call subroutine
     - In this case, $x = 200$ and $y = 500$ as we work in reverse order
  4. Get the return value from the eax register into result
  5. Cleanup the stack

Notes

Questions?

Notes

Outline

1. Computer Architecture
2. Assembly Languages
3. The Stack
4. Inline Assembly in C
5. Summary

Notes

Summary

- All high-level languages are compiled into machine instructions that can be executed by CPUs.
- C is one of the lowest-level of high-level languages. The examples in this module are here to show you how close to the machine C can be.
- Inline assembly has a limited interest, except in very specific cases. Modern compilers with the right flags are often better than humans at optimisation!
- It is useful to know how C and ASM code can map together. Sometimes, the only debugger output you get is in ASM and having some knowledge can help.

Notes

To do...

- **Remember** - do the end of unit exercises. Don't ignore these. They help understanding.
- Workbook - applied examples of inline assembly, also additional structures not covered in the lecture (loops, `structs`).
- Tutorial
- Next lecture - will cover includes, pre-processor directives, and applications with multiple source files.

Notes

Notes

Notes