

03 | Assembly Code | Jumps | Conditions

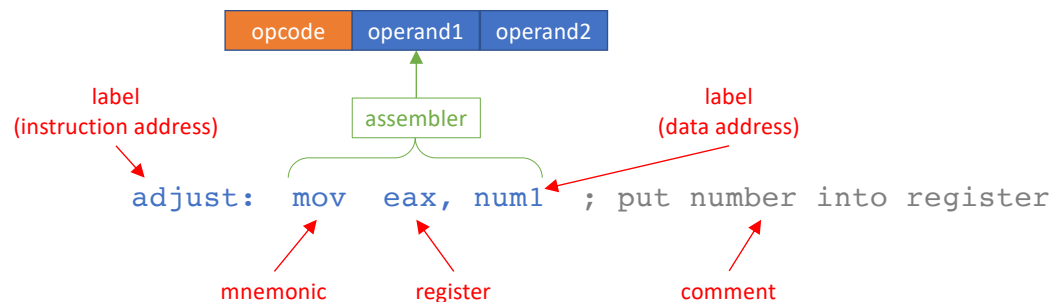
Dr Stuart Thomason

Machine Code Programming

- The CPU sees only binary numbers in memory
- Writing machine code directly in binary would be far too difficult and error prone
- So we use [assembly language](#) instead
 - Still very low level
 - Opcodes are represented by [mnemonics](#) (human-readable names)
 - Registers are given names
 - Memory addresses are specified using [labels](#)
- We use an assembler to turn our code into an executable sequence of instructions
 - Not quite the same as compilation of a high level program
 - Each line of assembly code translates directly into one machine code instruction
 - So we are still programming at a fairly low level

Example Line of Code

- A **label** can refer to the address of an instruction or the address of a data item
- The **mnemonic** and its **operands** are directly translated into machine code



- Comments start with a semi-colon
- Instruction labels and comments are optional
- So a basic line of code will have one mnemonic and zero, one, or two operands

Assembly Programming

- We will write assembly code for the **32-bit Intel x86** architecture using **Intel** syntax
 - The Intel 64-bit architecture is called x64 (so it's easy to get them mixed up)
 - Some text books and online resources use AT&T syntax (which will not work)
 - If you see **%** and **\$** symbols then you're looking at AT&T syntax

<code>mov eax, 2</code> (Intel)	<code>mov \$2, %eax</code> (AT&T)
------------------------------------	--------------------------------------

- To make it easier to write, compile and debug assembly code, we will place it inside a C++ wrapper and use Visual Studio
 - Allows us to declare variables in C and access functions in the C standard library
 - You do not need to know C or C++
 - Just use the template wrapper and put your assembly code in the right place
 - Our assembly code goes inside the `_asm` block (the inline assembler)

Example C++ Program

```
#include <stdio.h>
#include <stdlib.h>

int main (void) {

    int num = 10;

    _asm {
        mov eax, num
        add eax, 12
        mov num, eax
    }

    return 0;
}
```

Annotations:

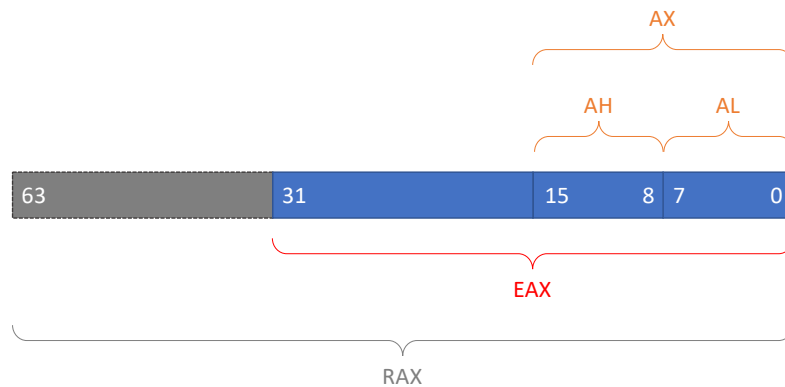
- `int num = 10;`: declare variables outside the assembly code block (using C syntax, which looks the same as Java)
- `_asm { ... }`: inline assembly code block
- `mov eax, num`, `add eax, 12`, `mov num, eax`: assembly code (Intel syntax)
- `return 0;`: return zero to the operating system to indicate successful termination

Intel x86 Registers

- The Intel x86 CPU has lots of registers (more than we will use on this module)
- Already mentioned **IP** (instruction pointer) and **IR** (instruction register)
- There are four main general purpose registers
 - **EAX** – accumulator register
 - **EBX** – base register
 - **ECX** – counter register
 - **EDX** – data register
- Although they have designated meanings, you can use them for whatever you like
 - We usually store calculations in the accumulator register
 - And we use the counter register for keeping track of loop iterations
 - But this is only by convention – we can use these registers to store anything
- There are a few other registers that we will introduce in later lectures

The Accumulator

- Most CPUs have an accumulator register
- The accumulator (**EAX**) is used for general purpose data storage and computation
- You can refer to it by different names depending on which bits you want to access




Example Code Fragments

- Put (move) 42 into the accumulator
`mov eax, 42`
- Move lowest 16 bits of a variable into the accumulator (`count` is the variable label)
`mov ax, count`
- Move ASCII value of 'x' into lowest byte of the accumulator
`mov al, 'x'`
- Increment the value in the accumulator (ie. add one to it)
`inc eax`
- Add 10 to whatever's in the accumulator
`add eax, 10`
- Note that the first operand is the destination and the second operand is the source (opposite of AT&T syntax)

Valid Move Operations

- The source operand is not changed or erased (so it's more like a copy than a move)
- You can move from one register to another
`mov eax, ebx`
- Or from memory to a register (`foo` is the label of the memory declared as a C variable)
`mov eax, foo`
- Or from a register to memory (overwriting whatever was in that memory)
`mov foo, eax`
- Or put a numeric value into a register
`mov eax, 12`
- Or put a numeric value into memory
`mov foo, 12`
- You cannot move directly from memory to memory (must use a register in between)

Basic Maths Operations

- Remember that a single line of high level code turns into multiple machine instructions
`int num = count1 + count2 - 10;`
- We need to use a register to perform each step of the calculation
`mov eax, count1`
`add eax, count2`
`sub eax, 10`
`mov num, eax`
- The accumulator is well named because it 'accumulates' the result of each step
- Addition and subtraction work as you would expect with the given register operand
- Multiplication only takes one operand (the thing to multiply by)
 - Always applies the calculation to the accumulator
 - And stores the result back into the accumulator
`mov eax, 10`
`mul 12`  stores 120 in accumulator

Division

- You need to set up a few things before you can do the calculation (eg. $120 \div 9$)



- Dividend formed from data register (high 32-bits) and accumulator (low 32-bits)
- Divisor must be stored in another register
- This performs integer division (so there could be a remainder)
- Result is stored in the accumulator and remainder is stored in data register

```
mov ebx, 9  
mov edx, 0 ← must clear contents of high 32-bits  
mov eax, 120  
div ebx ← stores 13 in accumulator and 3 in data register
```

- Operation will set status flags if the result is too big or you try to divide by zero

Status Flags Register

- The Intel x86 CPU has a special register where each bit represents a true/false status
- It has many flags that we won't use on this module
- Of interest to us are...
 - CF** – carry flag – previous operation had a carry from the most significant bit
 - ZF** – zero flag – previous operation had a zero result
 - SF** – sign flag – previous operation was positive (0) or negative (1)
 - OF** – overflow flag – previous operation result was too big to fit in memory
- We can use jump instructions to check flags and take appropriate action

Unconditional Jump

- An unconditional jump will move the instruction pointer to the given address label

```
    mov eax, 10
begin: add eax, 10
      mul 10
      jmp begin
```
- The above code would never terminate because the unconditional jump moves the instruction pointer back to a previous line of code
- Eventually the value in the accumulator would get too big and overflow
- Jumping is unrestricted, so you should take care to avoid 'spaghetti' code with jumps all over the place
- This is the basis of implementing loops in assembly code, but obviously we want them to terminate properly, so we need to look at a few more jump instructions first

Conditional Jumps

- A conditional jump will only happen if a certain condition is true
- If the condition is false, the instruction pointer just moves to the next instruction
- We can write code that behaves the same as the 'if' statement in high level languages
- There are various jump instructions that test different status flags
 - `jc` – jump if carry flag is set
 - `jnc` – jump if carry flag is not set
 - `jz` – jump if zero flag is set
 - `jnz` – jump if zero flag is not set
 - `js` – jump if sign flag is set
 - `jns` – jump if sign flag is not set
 - `jo` – jump if overflow flag is set
 - `jno` – jump if overflow flag is not set

Conditional Jump Example

- Implementing assembly version (right) of a high level piece of code (left)

```
// assume num is defined
num = num - 10;
if (num == 0) {
    num = 100;
}
                                mov eax, num
                                sub eax, 10
                                jnz store
                                mov eax, 100
                                store: mov num, eax
```

- The end result depends on the value of `num` at the start
 - If `num` is `10`, the code replaces it with `100`
 - Otherwise it is unchanged
- In the high level code, it's easy to see the code block that belongs to the 'if' statement
- In the assembly code, the condition is 'reversed' to make it more efficient (ie. if the result is not zero we jump over a line of code, as we do in the high level code)

Comparing Values

- The `cmp` instruction compares two values
- Internally, it subtracts one from the other without changing either operand
- If both values are the same, this sets the zero flag
`cmp eax, ebx`
- By placing this immediately before a jump instruction, we can respond to the result
 - `je` – jump if the operands are equal
 - `jne` – jump if the operands are not equal
 - `jg` – jump if the first operand is greater (equivalent to `jnl`)
 - `jle` – jump if the first operand is less than or equal (equivalent to `jng`)
 - `jl` – jump if first operand is less than (equivalent to `jnge`)
 - `jge` – jump if first operand is greater than or equal (equivalent to `jnl`)
- Remember, these jump instructions only work as expected if they immediately follow a compare instruction

Implementing IF-ELSE in Assembly

- High level code (left) becomes a collection of jumps in assembly code (right)

<pre>// assume vars are defined if (num > 0) { pos = pos + num; } else { neg = neg + num; }</pre>	<pre>mov eax, num cmp eax, 0 jg postv negtv: add neg, eax jmp endif postv: add pos, eax endif:</pre>
--	--

- Assume this is part of a loop that reads numbers from the user and keeps running totals of the positive and negative numbers entered
- If the number is negative, the first jump ([jg](#)) doesn't happen, but we must then jump over ([jmp](#)) the code to handle positive numbers