CoE 163

Computing Architectures and Algorithms

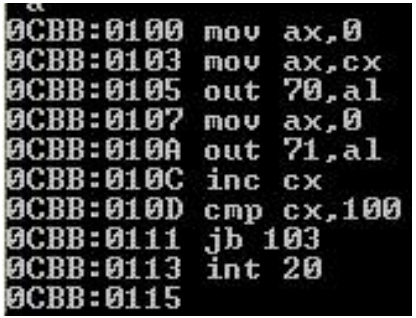
03b: x86 Assembly Introduction

COMPUTER ARCHITECTURE

A *computer architecture* describes how a computer should work, from the hardware to the software.

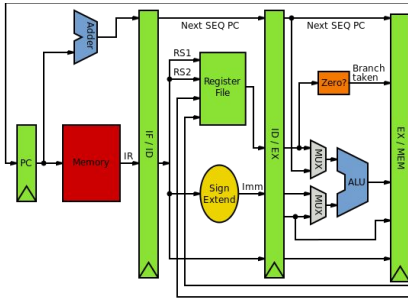
This also includes any peripherals that may be connected to it.

COMPUTER ARCHITECTURE



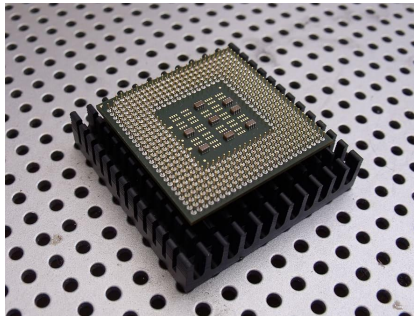
Instruction Set Architecture (ISA)

Abstract model that defines the what a computer should look like from a programmer's perspective.



Microarchitecture

Implementation of an ISA with several factors in consideration, such as component interconnection, cache and memory sizes, etc.



Implementation

Hardware design engineering - the microarch is converted into a concrete and tangible processor.

ISA CLASSIFICATION

ISAs can be classified in different ways,. More commonly, they can be split into two types based on instruction complexity.

- Complex instruction set
- Reduced instruction set

ISA: CISC

A complex instruction set computer (CISC) has many specialized instructions to support different use cases.

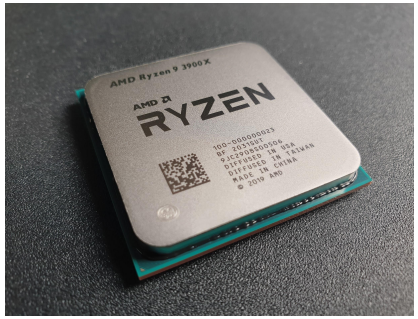
This is easier to program, but hardware should be able to support the huge amount of instructions.

ISA: RISC

A reduced instruction set computer (RISC) has a basic set of instructions - at least enough for it to be Turing complete.

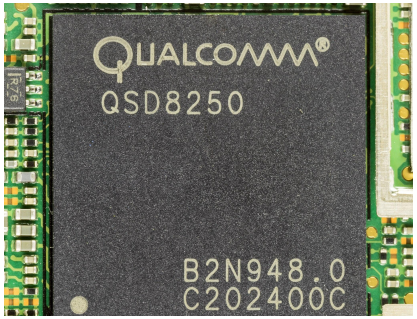
They are easy to implement in hardware, but software has to make do with the reduced number of instructions.

MAINSTREAM ISAs



x86 (CISC)

Most commonly found in desktop and laptop computers



ARM (RISC)

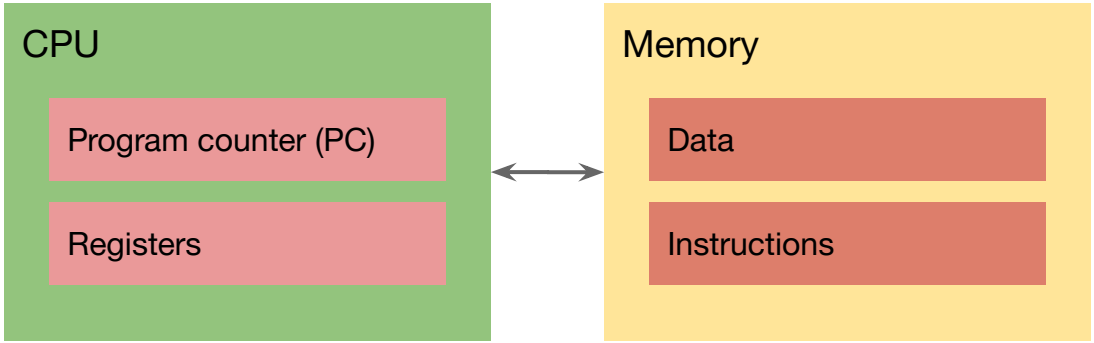
Most commonly found in cellphones, tablets, and small single-board computers like the Raspberry Pi

MIPS (RISC)

Most commonly found in gaming consoles and small electronics such as routers

MIPS is recently abandoned in favor of RISC-V

ASSEMBLY PROGRAMMING MODEL



x86_64 ISA

In a past course, MIPS was introduced (which is currently being abandoned). However, desktop and laptops use 64-bit x86 processors.

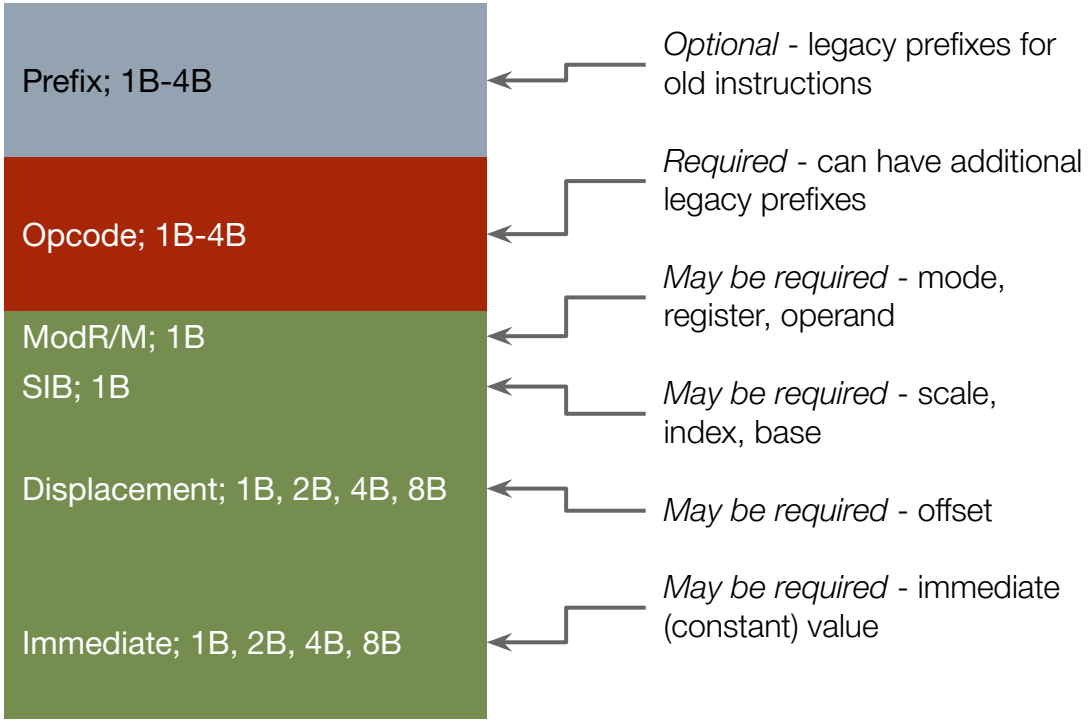
To be able to maximize programs, we should also be able to read and understand programs written in the x86_64 ISA.

x86_64 ISA

An instruction can have at most 15 bytes. It is little-endian (i.e. legacy prefixes/LSB at lowest address).

Normal (32-bit) x86 are also encoded in the same way.

x86_64 ISA


Prefix; 1B-4B
Opcode; 1B-4B
ModR/M; 1B
SIB; 1B
Displacement; 1B, 2B, 4B, 8B
Immediate; 1B, 2B, 4B, 8B
Optional - legacy prefixes for old instructions
Required - can have additional legacy prefixes
May be required - mode, register, operand
May be required - scale, index, base
May be required - offset
May be required - immediate (constant) value

x86_64 REGISTERS

A *register* is a small memory cell within the CPU. There are 16 64-bit *general-purpose registers* in x86_64.

Some of these registers have special purposes, such as the stack pointer `rsi`.

The registers have “root words” (`a, b, c, d, si, di, bp, 8-15`) affixed by a length identifier (`r, e, w, d, h, l, b`).

x86_64 REGISTERS

A portion of each of the 16 registers can be referred to by writing the relevant affix, such as getting a quad (8 bytes), double word (4 bytes), word (2 bytes), and byte.

x86_64 SYNTAX

Unlike MIPS, the destination is implied to contain one of the operands to instructions that require at least two operands

x86 has two major syntax flavors: AT&T and Intel.

- AT&T is common in Linux systems.
- Intel is common in Windows systems.

For the rest of the discussion, we will be using Intel syntax.

x86_64 SYNTAX

Intel

- Order is dest, src
- Uses registers as is
- Uses immediates as is
- Hex numbers are suffixed by “h”
- Memory dereferencing uses square brackets []
- Memory addressing uses math operations
- Uses type keywords (e.g. DWORD PTR, WORD PTR) to denote operand size

AT&T

- Order is src, dest
- Registers are prefixed by a percent %
- Immediates are prefixed by a dollar sign $
- Hex numbers are prefixed by “0x”
- Memory dereferencing uses parentheses ()
- Memory addressing uses function syntax
- Uses operation affixes (e.g. movd, movl) to denote operand size

x86_64 SYNTAX

`mov`e a 32-bit value of 1 into `eax`

```
mov eax, 1
movl $1, %eax
```

`mov`e the value in address `ebx + 3` into `eax`

```
mov eax, [ebx + 3]
movl 3(%ebx), eax
```

`add` the values in addresses `ebx + ecx * 0x2` and `eax`, then store the answer into `eax`

```
add eax, [ebx + ecx * 2h]
addl (%ebx, %ecx, 0x2), %eax
```

`mov`e the 64-bit value of 2 in address held by `ebx`

```
mov QWORD PTR [ebx], 2
movq $2, (ebx)
```

x86_64 ASSEMBLERS

There are several x86 assemblers available. The following lists the most popular ones

- GNU Assembler (GAS)
- Flat Assembler (FASM)
- Netwide Assembler (NASM)
- Microsoft Macro Assembler (MASM)

ASSEMBLY IN C/C++

GCC supports writing and compiling of assembly code within C/C++.

It is also possible to write the assembly code separately as a `.s` file.

ASSEMBLY IN C/C++

Subroutines in assembly should have an equivalent prototype in the C/C++ part of the code.

C++

```
// outside main()
extern "C" int add_ints(int, int);

[...]
// in main()
int c = add_ints(a, b)
cout << "add_ints() returned " << c << endl;
[...]
```

ASSEMBLY IN C/C++

Subroutines should be written using the calling convention suitable for the operating system. The Windows convention is used in this example

Intel ASM

```
; Windows
add_ints:
    mov rax, rcx
    mov rax, rdx

    ret
```

ASSEMBLY IN C/C++

Using NASM, the asm code should be first compiled into object code. This object code is then used with gcc to compile the program into an executable.

CMD

```
$ nasm.exe -f win64 hello.s
$ g++ hello.cpp hello.obj -o hello
```

RESOURCES

- x86 introduction from the University of Washington (US)
- x86 reference from the University of Virginia (US)

CoE 163

Computing Architectures and Algorithms

03b: x86 Assembly Introduction