Scientific Computing: Not Only for the NERDS

Zhongming Qu April 27, 2014

Contents

1 Brief Introduction

Computers usherred us into this so-called information era. Computers are not new. Broadly speaking, any 'thing' that assists people in calculation can be count as computer, or a simpler version, calculator. Computers have really changed the way we do things. We use them everywhere. And in most situations them are fairly easy to use.

The purpose of this lecture is, however, to 1) reveal the dirty internals of computers as much as possible, 2) help build an adequate level of appreciation of the complexity of modern computer systems, 3) demystify some common incorrect beliefs regarding computers, and 4) inspire ideas. This lecture aims at a very general audience. No prior knowledge is required to understand anything. Absolute newbies, computer lovers, amateur and semi-expert programmers, and maybe even computer experts can have some refreshment on their existing perspectives, if any, regarding computers.

2 What's in a CPU?

2.1 1+1=2 Revisited

What is really going on when we do a `1+1=2` calculation?

2.2 Operator and Operands

[graph]

2.3 Compound Arithmetic Expressions

[1+3*5] precedence, associativity

2.4 Tree Structure of Compound Arithmetic Expressions

[graph:1+3*5]

2.5 Parsing/Flattening Trees of Arithmetic Expressions

[lecture:1+3*5]

2.6 Example

`cpu/allis.exe`:

```c
#include <stdio.h>

int fun(int *n)
{
#pragma forceinline
        return n[0] + n[1] * n[2];
}

int funx(const int *n)
{
#pragma noinline
        return n[0] + n[1] * n[2];
}

int main(int argc, char const* argv[])
{
        int n[3]={1,3,5};
        const int nx[3]={1,3,5};
        printf("%d\n",fun(n));
        printf("%d\n",funx(nx));
        return 0;
}
```

excerpt from `cpu/allis.s`:

```
# -- Begin  _Z4funxPKi
...
# parameter 1: %rdi
###     return n[0] + n[1] * n[2];
        movl      4(%rdi), %eax                                 #11.16
        imull     8(%rdi), %eax                                 #11.23
        addl      (%rdi), %eax                                  #11.23
        ret                                                     #11.23
...
# -- End  _Z4funxPKi
```

A complete/near-complete exposition of the above assembly listing would lead to the discussion of many aspects on the CPU. We will explain each and every of them in the following sections.

2.7 Instructions Set Architecture (ISA)

instruction, opcode : `0010 0111 1101 0111`
machine language = the set of opcodes
assembly code: `addl opd1, opd2`
Take the `x86_64` architecture as an example:
int classification: `x86_64` instruction sets: see
`http://en.wikipedia.org/wiki/X86_instruction_listings`

[in lecture] proper reference to `http://x86.renejeschke.de/`

arithmetic(integral,floating point): add, sub, mul, imul
logic,flow control: cmp, jmp(goto)
load/store,memory,cache,tlb,etc.: mov, wb
register classification:
`x86_64` registers: see
`https://www.tortall.net/projects/yasm/manual/html/arch-x86-registers.html`
`https://software.intel.com/en-us/articles/introduction-to-x64-assembly`
`http://en.wikipedia.org/wiki/Advanced_Vector_Extensions`
general purpose (arithmetic operand, address container)
special/misc (interrupt/signal, mode, flags, etc.)

2.8 Machine Representation of Numbers

Integral numbers are easy.
Floating number are not that easy.
The IEEE754 standard says:
`http://steve.hollasch.net/cgindex/coding/ieeefloat.html`
The floating point operations: add, sub, mul, div, fsqrt
The design and implementation ensures that the resulting numbers are the best approximation to an exact intermediary result. The largest possible relative error is $2^{-52} = 2.2 \times 10^{-16}$. Therefore, the number of significant digits is 16 for double precision numbers.

2.9 Vendors and Instruction Sets

[MMX,SEE,SEE2,SEE3,SEE4,SEE4.1,SEE4.2,AVX,AVX2]
[3DNow!]
[PPC]
[ARM]
[Loongson] (!!)
[MIPS,MIPS64]

3 Compiling, Assembling, and Linking

3.1 Usual work flow

preprocess, compile, assemble, link

3.2 Preprocess

Replace `include` with the included files and macros with their defs.
Roughly speaking, nothing happens in this step.

3.3 Compile and Assemble

Translate human-readable languages such as Fortran/C/C++ into machine readable languages, or machine languages.

Usually, these two steps are done by the compilers at one go. So, frequently we no longer distinguish them and just call them collectively as compiling.

Each .c/.cpp file is compiled into one and only one .o file in this step.

However, we still cannot run the resulting files. The resulted files are not executables.

3.4 Link

Link .o files, add ”starting sections”, make an executable file.

3.5 Libraries, Shared Objects

[static library vs. dynamic library]
[Windows dll, Linux so, MaxOSX dylib]
[good libraries, bad libraries]

3.6 matlab, Mathematica, python, shell, ruby, R, etc.

Interpretive, translate on the fly, easily slow by a factor of 20-100 compared to compiled programs.

4 `cpu/allis.exe` Revisited

4.1 Several Basic Optimizations

[function inline: fun, funx]
[-O0, -O3, compiler optimization: manual switch in lecture]

5 Memory, Cache, TLB

5.1 `mem/bw/time_sumcopy.exe`

Takes ¡2min to run.

6 Parallelism: OverView

6.1 Instruction and Thread

6.2 Thread and Process

6.3 Thread and Core

[different from thread and process]

6.4 Core and Processor

6.5 Processor and Node

6.6 Node and Cluster

6.7 Cluster and Cloud

Appendices

A History of Unix, Linux, and C

A.1 AT&T, Bell Labrotary, and Unix

A.2 Free Software Foundation

A.3 Linux and GNU

A.4 POSIX and SuSv

References