Tutorial 2

Counting Cycles in a Simple Program

1 Instructions and Cycles

The “NARC” which we discussed in the lecture notes showed you an example for a simplified computer with a simplified instruction set, and we discussed the sequence of steps that has to happen to execute some of these instructions. The aim of this exercise is to give you a feel for how this works in a real system. We will look at a small program which is written in C, then compile this to assembler. Looking at the assembler that is generated allows us to see how many instructions are needed to implement this simple program. We will then run the program and measure how long it takes. This then allows us to calculate how many cycles the real computer that we use requires on average to execute each instruction.

1.1 Compiling a Program to Assembler

You can compile a C (or C++) program to assembler by passing the `-S` flag to gcc. It is advisable to compile with some optimisation (`-O2`) when doing this in order to get cleaner, more compact code.

1.2 Counting Cycles

Suppose we know that a particular function has been translated to $X$ instructions. How can we find out how many cycles, on average, each of these instructions takes? We measure how long, in seconds, it took to execute this block of instructions (I am giving you the code to do this). Call this amount of time $t$. Next, we check what the clock speed of the computer is that we are running on (recall from Tutorial 1 that you can find this information in `/proc`). Assume that we discover that we have a 1400 MHz machine, then the average number of cycles per instruction is $t \times 1000.0 \times 1400/X$. This quantity is also called $cpi$ (cycles per instruction).

1.3 The Fibonacci program

Download the following C++ program from


```c
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_DATA 100

double time_in_seconds( ) {
    double result;
    struct timeval tv;
    struct timezone tz;
    gettimeofday( &tv, &tz );
    result = (double) tv.tv_sec;
```
result += ((double) tv.tv_usec / 1000000.0);
return result;
}

long fib1( const int n ) {
if( n == 0 ) {
  return 1;
} else {
  return fib1( n - 1 ) + fib1( n - 2 );
}
}

long fib2( const int n ) {
if( n == 0 ) {
  data[n] = 1;
  return data[n];
} else {
  data[n] = fib2( n - 1 ) + fib2( n - 2 );
  return data[n];
}
}

int main( int argc, char *argv[] ) {
if( argc != 2 ) {
  fprintf( stderr, "Usage: ./fibonacci <n>\n" );
  exit( 1 );
}

int n = atoi( argv[1] );

double time1 = time_in_seconds();
printf( "Fib1(%d) = %ld.\n", n, fib1( n ) );

double time2 = time_in_seconds();
printf( "Fib2(%d) = %ld.\n", n, fib2( n ) );

return 0;
}

Compile this program with gcc -Wall -O2 -o fibonacci fibonacci.c. Run the program once to make sure it works.

1.4 Counting Instructions

Compile this program to assembler, as explained above. Look at it and see whether you can find the fib1 function. Your next task is to calculate (or, you can write a little C function that does this for you) for many instructions it takes to calculate \( fib1(n) \) for a given \( n \). To help with this, I include below an annotated version of the generated assembler (yours may be slightly different — count instructions for your version).

fib1:  ; Start of fib1 function
    pushl %ebp  ; (Manage stack)
    movl $1, %eax
    movl %esp, %ebp
    pushl %esi
    pushl %ebx
    movl $(&%ebp), %esi
    testl %esi, %esi  ; Check whether n == 0
    je .L2            ; Jump to L2 if not
    movl $1, %eax
    cmp $1, %esi      ; Check if n == 1
    je .L2            ; Jump to L2 if not
45          subl $12, %esp ;  | Else
46          leal -1(%esi), %eax ;
47          pushl %eax ;
48          call fib1 ;
49          movl %eax, %ebx ;
50          leal -2(%esi), %eax ;
51          movl %eax, (%esp) ;
52          call fib1 ;
53          leal (%eax,%ebx), %eax ;
54          .L2:
55          leal -8(%ebp), %esp ; Code that runs in all
56          popl %ebx ; cases
57          popl %esi
58          popl %ebp
59          ret ; End of function fib1

1.5  Cycles per Instruction for Different Values of n

Now modify the fibonacci.c program so that apart from printing out the \( n \)th Fibonacci number, it also prints the average number of cycles per instruction that it took to calculate this number. what can you observe when doing this for different values of \( n \)?