ELEC2204

Computer Engineering

Updated: February 8, 2017

Contents

1 Performance

1.1 MIPS: Millions of Instructions Per Second

- ⊖ Doesn't take into account the complexity of the instructions.
- ⊖ Varies between programs on the same computer; so cannot allocate a single MIPS to a machine under test.

$$\begin{aligned}\text{MIPS} &= \frac{\text{Instruction Count}}{\text{Execution Time} \times 10^6} \\ &= \frac{\text{Instruction Count}}{\frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}} \times 10^6} = \frac{\text{Clock Rate}}{\text{CPI} \times 10^6}\end{aligned}$$

1.2 MFLOPS: Million Floating Point Operations Per Second

This measure of performance looks at the number of floating point operations completed per second. This doesn't take other tasks into account and not all floating point operations are implemented on all machines, eg. one machine may require more operations to do a task than another and would therefore have a higher MFLOPS rating.

$$\text{FLOPS} = \frac{\text{Floating Point Operations}}{\text{Execution Time} \times 10^6}$$

1.3 Response Time and Throughput

Response time is how long it takes to complete a task. Throughput is the total work done per unit time, eg. tasks/transactions per hour.

1.4 Relative Performance

$$\begin{aligned}\text{Performance} &= \frac{1}{\text{Execution Time}} \\ \frac{\text{Performance}_x}{\text{Performance}_y} &= \frac{\text{Execution Time}_y}{\text{Execution Time}_x} = n\end{aligned}$$

For example, time taken to run a program on machine A is 10s but 15s on machine B. This means that A is 1.5 times faster than B.

1.5 Measuring Execution Time

There are two ways to measure execution time of a program.

- Elapsed Time: Total response time including all aspects, processing, I/O, OS overhead, idle time etc.
- CPU Time: Time spent processing a given job. Comprises of user CPU time and system CPU time. Different programs are affected differently by CPU and system performance.

1.6 CPU Clocking

$$\begin{aligned}\text{CPU Time} &= \text{CPU Clock Cycles} \times \text{Clock Cycle Time}\\ &= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}\end{aligned}$$

Performance of a system can be improved by reducing the number of clock cycles or increasing the clock rate. Hardware designer often makes trade off between clock rate and cycle count.

Computer A: 2GHz clock, 10s CPU time
Designing Computer B: Aim for 6s CPU time, faster clock causes 1.2 $\times$ clock cycles.
Determine the required clock rate of Computer B.

$$\begin{aligned}\text{Clock Rate}_B &= \frac{\text{Clock Cycles}_A}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6\text{s}}\\ \text{Clock Cycles}_A &= \text{CPU Time}_A \times \text{Clock Rate}_A\\ &= 10\text{s} \times 2\text{GHz} = 20 \times 10^9\\ \text{Clock Rate}_B &= \frac{1.2 \times 20 \times 10^9}{6\text{s}} = \boxed{4\text{GHz}}\end{aligned}$$

1.7 Instruction Count and CPI

Instruction count for a program is determined by the program, ISA and compiler. Average cycles per instruction are determined by CPU hardware.

Computer A: Cycle Time = 250ps, CPI = 2.0
Computer B: Cycle Time = 500ps, CPI = 1.2
Same ISA
Which is faster and by how much?

$$\begin{aligned}\text{CPU Time}_A &= \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A\\ &= \text{I} \times 2.0 \times 250\text{ps} = \text{I} \times 500\text{ps}\\ \text{CPU Time}_B &= \text{I} \times 1.2 \times 500\text{ps} = \text{I} \times 600\text{ps}\\ \frac{\text{CPU Time}_B}{\text{CPU Time}_A} &= \frac{\text{I} \times 600\text{ps}}{\text{I} \times 500\text{ps}} = \boxed{1.2}\end{aligned}$$

Therefore, A is 1.2 times faster than B.

1.8 CPI in More Detail

If different instruction classes take different numbers of cycles,

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

Weighted average CPI,

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}} \right)$$

1.9 Performance Summary

Performance depends on the algorithm of the program, programming language, compiler and instruction set architecture.

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycles}}$$

2 Arithmetic

2.1 Floating Point Numbers

Floating point numbers consist of 1 sign bit, an exponent bias, an implied 1 bit (which we will understand from the example conversion below) and finally the fractional part of the number.

	S	E-Bias	1	Fraction
IEEE	[1]	[8]	[1]	[23]

Figure 1: Floating point number representation.

2.2 Converting Decimal to Floating Point

- Convert integer part to binary.
- Convert fractional part by multiplying by 2 each time and shifting the decimal point right one until there is no fractional part left.
- Append 2^0 to the end of the number, this has no effect.
- Normalise the number by moving your binary point to one bit from the left (be sure to adjust exponent accordingly).
- The fractional field is then the bits to the right of the binary point. Pad the remaining bits with 0's, so for IEEE 32-bit floats the fractional field has 23 bits.
- Add an exponent bias of $2^{k-1} - 1$ where k is the number of bits in the exponent field. For IEEE 32-bit float, $k = 8$, so the bias is $2^{8-1} - 1 = 127$.
- Set the sign bit as `1` if the number is negative, `0` otherwies.

Convert 2.625 to our 8-bit floating point format.

Integer part, $2_{10} = 10_2$.
Fraction part,

$$
\begin{aligned}
0.625 \times 2 &= 1.25 \quad \boxed{1} \\
0.25 \times 2 &= 0.5 \quad \boxed{0} \\
0.5 \times 2 &= 1.0 \quad \boxed{1}
\end{aligned}
$$

So, $0.625_{10} = 0.101_2$ and $2.625_{10} = 10.101_2$.

$$
\begin{aligned}
\text{Add exponent part, } 10.101_2 &= 10.101_2 \times 2^0 \\
\text{Normalise, } 10.101_2 \times 2^0 &= 1.0101_2 \times 2^1 \\
\text{Fractional field, } & 0101_2 \\
\text{Exponent, } 1 + 127 &= 128_{10} = 10000000_2 \\
\text{Sign bit, } & 0
\end{aligned}
$$

$\therefore 2.625_{10} = \boxed{0}\boxed{10000000}\boxed{01010000000000000000000}$

2.3 Adders

Ripple Adder

An n-bit ripple adder is constructed by cascading n full adder cells. The size of this is proportional to the number of cells you have. However since cell $n+1$ takes the carry bit from the previous addition the delay will also be proportional to n.

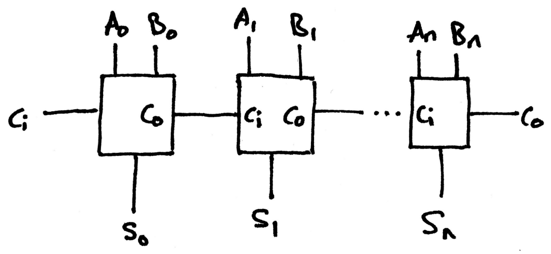

Figure 2: Cascading full adders.

Carry–Lookahead Adder

A carry-lookahead adder works by calculating the carry of each adding stage separately. This makes the delay constant regardless of the number of bits but also low (only chip propagation delays). However each stage is geometrically larger than the previous and it is said that size $\propto n^2$.

2.4 Multicycling Addition

2.5 Floating Point Addition/Subtraction

2.6 Floating Point Multiplication/Division

2.7 Microcoded Operations

Microcode allows us to add simple functionality that isn't already present in hardware. Below is an example of a 32-bit unsigned square root in C code.

```
int fn sqrt(int val)
{
    int mask = 0x00008000;
    int best = 0x00000000;
    if (val <= 0) return 0;
    while (mask != 0) {
        if ((best+mask^2) <= val) best |= mask;
        mask >>= 1;
    }
    return (int)best;
}
```