CS61C Summer 2014 Discussion 9 Notes

Written by Andrew Luo and CS61C course staff

Power

The primary source of power dissipation for CMOS technology is *dynamic power*, the power consumed during transistor switching:

$$\text{Power} = \text{Capacitive Load} \times \text{Voltage}^2 \times \text{Switching Frequency} = \text{C} \times \text{V}^2 \times \text{f}$$

Parallelism

Software can be classified as **sequential**, where a single computational process is carried out, or **concurrent**, where collections of interacting computational processes are executed. Hardware can be classified as **serial** or **parallel**, depending on whether one or more computations can be carried out simultaneously. These software and hardware classifications are independent of each other (eg, a sequential process can run on hardware that may execute multiple instructions at a time).

Job-level (process-level) parallelism, where independent programs are run on different processors can take advantage of **multiprocessors**, computers with more than one processor. In particular, today we use the term **multicore microprocessor** in reference to multiprocessors with multiple processors (cores) on the same IC (chip).

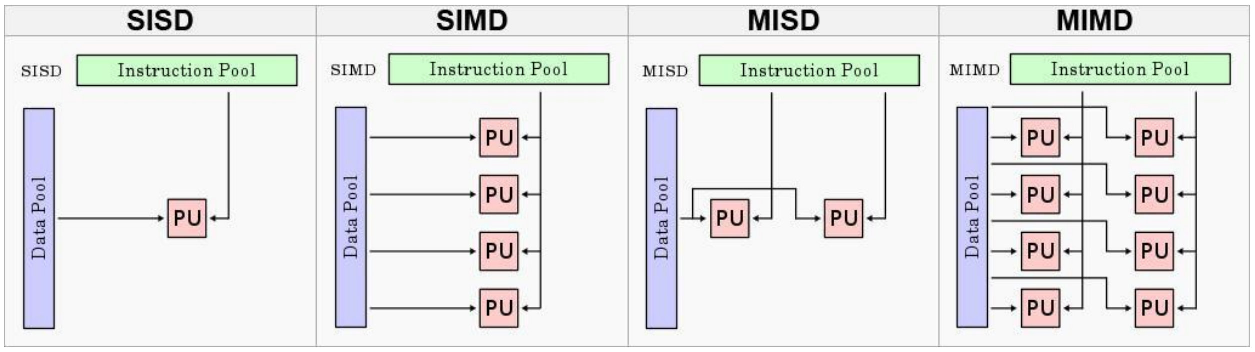
Thread Level Parallelism (TLP): Executing different processes (threads) of the same program on different processors. Threads can communicate with each other.

Instruction Level Parallelism (ILP): Performing different instructions in a program simultaneously.

Data Level Parallelism (DLP): Operating on independent data simultaneously.

Flynn's Taxonomy

Flynn's taxonomy is a classification of computer architectures into Single/Multiple Instruction and Single/Multiple Data stream. The following image is taken from Wikipedia:



So far we have been assuming use of SISD, or just a normal uniprocessor machine. MISD is generally unused, so the main areas of interest are **SIMD and MIMD**.

Amdahl’s Law

In general terms, **Amdahl’s Law** states (quoting P&H) that “the performance enhancement possible with a given improvement is limited by the amount that the improved feature is used.” Let “exec time” stand for execution time:

$$\text{exec time after improvement} = \frac{\text{exec time affected by improvement}}{\text{amount of improvement}} + \text{exec time unaffected}$$

As stated above, it’s clear to see that this supports the common design principle of *making the common case fast*.

Restated in terms of parallelism, the potential speedup from parallelization is limited by the amount a program can be parallelized and the level of parallelization (here amount of improvement = number of processors). Let be the fraction of the original execution time affected by the improvement (% of process that can be parallelized) and be the number of processors. Then the amount of speed up going from 1 to N processors is given by:

$$S(F, N) = \frac{1}{(1 - F) + \frac{F}{N}}$$

This equation has the following limits:

$$\lim_{F \to 1} S(F, N) = N \qquad\qquad \lim_{F \to \infty} S(F, N) = \frac{1}{1 - F}$$

SIMD and SSE

SSE, or Streaming SIMD Instructions, is an instruction set extension developed by Intel that SIMD instructions for floating point as well as integer data types (there was actually MMX before SSE, but it was much less popular). SIMD instruction extensions and SSE have been expanded throughout the years, with the original SSE (1999), SSE2 (2001), SSE3 (2004), SSSE3 (2006), SSE4 (2006), AES-NI (2008), AVX (2008), FMA (2013), BMI (2012). SSE introduced 8 new 128-bit registers that allow the manipulation of up to 4 32-bit (DWORD[1]) values, 2 64-bit (QWORD) values, 8 16-bit (WORD) values, or 16 8-bit (BYTE) values in 1 instruction.

Example:

```
float a[4], b[4], c[4]; //assume these are initialized

for (size_t i = 0; i < 4; i++)
{
    c[i] = a[i] + b[i]; //Non-SSE
}
```

```
__declspec(align(16)) float a[4];
__declspec(align(16)) float b[4];
__declspec(align(16)) float c[4];

_mm_store_ps(c, _mm_add_ps(_mm_load_ps(a),
_mm_load_ps(b))); //SSE
```

OpenMP

OpenMP is a cross-platform compiler extension to make writing multithreaded programs easier. Before OpenMP, writing multithreaded code generally required using platform-specific APIs (CreateThread on Windows or POSIX/pthreads on Linux). OpenMP directives are used to specify the start/end of a “parallel section” (where threads should be forked/joined).

Example:

```
#pragma openmp parallel for //assume a, b, and c are declared and initialized as in the example above
for (size_t i = 0; i < 4; i++) c[i] = a[i] + b[i];
```

[1] Note that this is different from a MIPS word – a MIPS word is 32-bits whereas an x86 word is historically 16-bits.