Computer Architecture

Shahzeb Asif

June 17, 2017

Introduction

Classes

There are several classes of computers:

- Personal mobile device (PMD)
  - These are things like smartphones.
  - Cost is the primary concern followed by power efficiency.
- Desktop Computing
  - The biggest computer market up until the rise of PMDs.
  - Desktops focus on price and performance.
- Servers
  - These are large computing systems with a focus on reliability (availability), scalability, and throughput.
- Clusters
  - These are special systems that consist of dozens upon dozens of desktops or servers connected together.
  - A cluster is often cheaper than a similarly powerful single-computer.
  - The focus is on availability, price, performance, and power.
- Embedded
  - These are to mice what a cluster is to an elephant.
  - Tiny computers hidden in every day devices.
  - Emphasis is on cost.
  - These do not run third party software.

Growth

Computers have been getting more and more powerful for a long time. Processor were becoming more powerful at a ridiculous 52%/year from 1986 to 2003. They slowed down to about 22%/year after 2003.

Note that we're talking about single-processors. The reason for the slow downs was reaching the physical limits of power dissipation through air cooling and lack of instruction level parallelism.

Instruction level parallelism (ILP) is about running instructions that don't depend on one another simultaneously on the same processor.

Clock speeds follow a similar pattern as just processor power. Processor speeds were increasing at a dramatic 40%/year from 1987 to 2003. But they slowed down to a minuscule 1%/year after 2003. The primary reason was the shift to exploit parallelism rather than single unit performance.

Compounding

It's worth mentioning that all these rates are compounded year after year.

Remember the compounding formula.

$$A = P \times (r)^t \tag{1}$$

In the equation above:

- A is amount.
- P is principal.
- r is the rate.
  - If something is increasing by 20% every year then $r = 1.2$.
  - If something is decreasing by 20% every year then $r = 0.8$.
- t is the number of periods.
  - It could be years, days, hours, etc. But the unit of periods must be the same as the period for the rate.

CPU

The CPU is the central processing unit that is like the brain of a computer.

It contains (among many others):

- Registers
  - Very quick access memory.
- ALU
  - The arithmetic and logic unit.
  - Handles operations at the hardware level and exploits hardware efficiency on top of software efficiency.

- PC (program counter)
  - A special register that contains the address of the next instruction to execute.

Buses

Buses are the wires that connect all the different parts of a computer together.

There are different types of buses:

- Data buses
- Address buses
- Control buses

Design

Power and Energy

Electrical power and energy have become the biggest concerns with processors now. Processors tend to generate a lot of heat which needs to be taken away or reduced.

The thermal design power (TDP) is the sustained power consumption. It is usually a value greater than the average power consumption but lower than the peak power consumption.

There are two types of energy consumption in processors:

- Dynamic energy
  - This is spent switching transistors.
  - $E_{dynamic} = \frac{1}{2} \times Capacitive_Load \times Voltage^2$
- Static energy
  - This is leaked from transistors.

Dynamic power is the energy per unit of time. $P_{dynamic} = E_{dynamic} \times f$

Comparing Energy

To compare efficiency of two processors, we compare the energy not the power.

$$E_{efficiency} = \frac{E_{new}}{E_{old}} \tag{2}$$

$$P_{efficiency} = \frac{P_{new}}{P_{old}} \tag{3}$$

Reducing Power

There are several techniques for reducing dynamic power:

- Idling properly
  - This may involve things like turning off the FP unit when it's not being used.
- Dynamic voltage-frequency scaling
- Low power for DRAM, disks
- Overclocking

Saving static power is more tricky because static power is leaked even with reduced clock frequency or when the processor is idling.

Parallelism

There are two different classes of parallelism:

- Data-level parallelism (DLP)
  - Data that can be operated on at the same time such as adding two arrays.
- Task-level parallelism (TLP)
  - Performing many independent tasks at the same time.

We can perform the two classes of parallelism in several different ways:

- Instruction-level parallelism (ILP)
- Vector architecture/graphic processor unit (GPU)
  - These can apply a single instruction to a large amount of data.
- Thread-level parallelism
- Request-level parallelism

Flynn's Taxonomy

This is a simple system meant to categories how a unit handles data and instructions:

- Single instruction stream, single data stream (SISD)
  - An example is the single-processor systems.
- Single instruction stream, multiple data stream (SIMD)
  - GPUs are an example of SIMD.
- Multiple instruction stream, single data stream (MISD)
  - No commercial use.
- Multiple instruction stream, multiple data stream (MIMD)

- Tightly-coupled
  - Exploit thread-level parallelism.
- Loosely-coupled
  - Exploit request-level parallelism.

Benchmarks

It's difficult to compare the performance of two different processors. The clock rate isn't entirely accurate but neither is the number of cores or any other metric.

There is no definitive method of showing one processor is better than another but that's why we use benchmarks.

We run a task and measure how long it takes for a processor to complete it.

$$R = \frac{ExecutionTime_B}{ExecutionTime_A} \tag{4}$$

Note that R is just a ratio.

For multiple benchmarks, we use the geometric mean **not** the arithmetic mean.

Amdahl's Law

Amdahl's law formalizes the idea that reducing time spent in bottlenecks will have the greatest effect on the performance.

Calculating Improvements

This is best illustrated with an example.

Let's say a processor spends 40% of its time on task A and the rest on task B.

You can cut time spent on A by half or time spent on B to a third but not both. Which one should you pick?

In this case, it's as simple as a weighted average.

$$Original = (1 \times 0.4)_A + (1 \times 0.6)_B = 1 \tag{5}$$

$$Possibility_1 = (0.5 \times 0.4)_A + (1 \times 0.6)_B = 0.8 \tag{6}$$

$$Possibility_2 = (1 \times 0.4)_A + (0.33 \times 0.6)_B = 0.6 \tag{7}$$

We can formally compare them as well.

$$Improvement_1 = \frac{Original}{Possibility_1} = \frac{1}{0.8} = 1.25 \tag{8}$$

$$Improvement_2 = \frac{Original}{Possibility_2} = \frac{1}{0.6} = 1.67 \tag{9}$$

The first possibility improves on the original by a factor of 1.25 but possibility two improves on the original by a factor of 1.67.

Note that we divided to find the improvement. We did not subtract.

In this scenario, possibility two in better.

Dependability

Dependability is a big concern with processors. To evaluate dependability, we use some mean time to failure and other measures.

- Mean time to failure (MTTF) is the time it takes for a device to fail.
- Mean time to repair (MTTR) is the time it takes to get a failed device back to running.
- Mean time between failures (MTBF) is just $MTTF + MTTR$.
- Availability is $\frac{MTTF}{MTBF}$

Multiple Components

Computer systems rarely exist in isolation. As such, the dependability often depends on several parts any of which may fail.

We can use simple statistics to find the average time to failure for a system of multiple parts.

$$Y_{total} = Y_1 + Y_2 + ... + Y_n \tag{10}$$

$$Y = rate = \frac{1}{MTTF} \tag{11}$$

The only tricky part is to ensure that the rate is for the same amount of time, i.e. all Y/rates are per hour or per day etc.

And remember that Y_{total} is just a rate. To get the actual time to failure for the system, you must take the inverse.

Memory

A computer contains many different types of memory.

There is a sort of hierarchy that gets larger, slower, and cheaper as we go down:

1. Registers
2. L1 cache (SRAM)
3. L2 cache (SRAM)
4. L3 cache (SRAM)
5. Main memory (DRAM)
6. Local disks (HDD/SDD)

7. Remote disks

Ideally, we want to limit how much we use the memory lower in the hierarchy because it's slower and expensive to use.

Latency

When we talk about memory performance, we usually talk about access time and cycle time.

Access time is the time between a read is requested and when the information arrives.

Cycle time is the minimum time between requests to memory.

RAM

RAM or random access memory is usually what we're talking about when we say memory. This is the main memory that all programs are placed in.

RAM has three buses attaches to it:

- Address lines
- Data lines
- Control lines

SRAM

SRAM is static RAM. It is bistable which means it is stable only in one of two positions. If it is unstable it will move until it is stable. Because SRAM is bistable, it can handle stresses fairly well.

SRAM is implemented with a six-transistor circuit so it requires power to maintain data.

SRAM is used in registers, caches, buffers, etc.

DRAM

DRAM is dynamic RAM. DRAM is usually used as the main memory in computers.

DRAM is made with capacitors so it can be made to be very dense. The disadvantage is that DRAM is very unstable because the capacitor can lose its charge very easily and cannot be recovered.

Even reading a bit in DRAM will destroy it so reading must be followed by writing.

DRAM must be refreshed every 10 to 100 milliseconds because the capacitors can leak their charge. To refresh, every bit must be read and then rewritten.

DRAM is much cheaper than SRAM but it has a slower cycle time because we must write after we read. DRAM is also much more sensitive than SRAM.

Organization

DRAM memory is usually organized in banks. Each bank contains many supercells of w bits.

These supercells in a bank are usually organized in a rectangular structure with rows and columns.

DRAM is also organized into modules with many banks. Each bank will contain a portion of a chunk of data. For example, the first byte of data might be on the first bank at supercell $(2, 2)$, the second byte will be on the second bank at the supercell $(2, 2)$,... All the supercells are located at the same address but different banks contain different parts of the data.

Retrieval

Retrieval occurs in a few steps:

1. Memory controller sends the row number to the memory.
2. The memory copies that row into its internal row buffer.
3. The memory controller sends the column number to the memory.
4. The memory will return the correct supercell according to the column number.

The advantage of sending the address in two steps is that we need fewer address lines but it does mean longer access times.

ROM

ROM or read only memory is also very important in computers. SRAM and DRAM are wiped when power is cut off. Memory that doesn't lose its state after power is cut off is called non-volatile memory.

There are several different types of non-volatile memory:

- Programmable ROM (PROM)
  - Only programmable once by blowing a fuse.
- Erasable programmable ROM (EPROM)
  - Programmable using special device.
  - Wiped with UV light.
  - Can be written to < 1000 times.
- Electrically erasable PROM (EEPROM)
  - Uses electricity to wipe and write information.
  - Can be written to $< 10^5$ times.

Note that there are limits to how many times EPROM and EEPROM can be programmed.

Solid State Disks

Solid State Disks (SSD) are based on flash memory.

The memory in SSDs is organized in pages within blocks. Each page may be 512 - 4 KB and a block contains 32 - 128 pages.

Reads are done on entire pages at a time. Writing to a block involves first erasing the entire block and then writing information. Specifically, this involves copying old information to a newly erased block. As expected, writes are very slow compared to reading.

Each block can be written to about 10^5 times.

Rotating Disk Drives

A Hard Disk Drive (HDD) is an example of a rotating disk drive. There are a few different components of an HDD:

- **Platters** are reflective metal discs that actually contain the data. These will be described in detail later.
- The **spindle** is the axis about which the platters rotate. The spindle spins around itself.
- The **arm** is the arm for the laser reading device to read the data from the platters.
- The **actuator** rotates to allow the arm to be at different angles so that it can read different locations on the platters.

Organization of Data

All the data is stored on the two surfaces of a platter that is coated with a magnetic recording material.

On each surface of a platter, there are several tracks. These can be thought of as the grooves on a vinyl record. These tracks are at different distances from the spindle.

Each track contains data **sectors** interleaved with **gaps** that contain information about the sector.

A collection of tracks may be organized in **cylinders**. All tracks in a cylinder have an equal number of sectors. These may also be called **zones**.

Because disk organization can get pretty complicated, there is an abstraction layer for OSes. A **HD controller** provides an abstraction for access to the disk. Instead of specifying surface number, track number, and sector number, the OS can simply tell the HD controller of a block number which the HD controller converts into an actual location on the disk.

The HD controller's abstraction allows the OS to treat the disk as a list of block numbers without caring about where the actual blocks are.

Capacity

Disk capacity is the number of bits that can be stored on a hard disk.

There are a few different factors that affect capacity:

- **Recording density** ($\frac{bits}{in}$) - bits that fit into a 1-inch segment of a track.

- **Track density** ($\frac{tracks}{in}$) - tracks that fit into a 1-inch radial segment.
- **Areal density** ($\frac{bits}{in^2}$) - bits that fit into an in^2 of the platter surface.

We can actually calculate capacity using a simple formula.

$$Capacity = (\frac{bytes}{sector}) \times (\frac{avg.sectors}{track}) \times (\frac{tracks}{surface}) \times (\frac{surfaces}{platter}) \times (\frac{platters}{disk}) \tag{12}$$

Operation

There are multiple platters with arms sandwiched between them for a single HDD. The platters are rotated by the spindle that they are attached to.

Because the arms are all in the same cylinder on each platter surface, they can all be written to or read at the same time.

Seek time is the time it takes for the arm to rotate to the right cylinder. Seek time is usually $\leq 20ms$.

Rotational latency is the time it takes for the entire spindle to rotate until the arm can read the required sector. The worst case is the time it takes for a full rotation. Average case is $\frac{worstcase}{2}$.

Transfer time is the time it takes to read a single sector. This is usually tiny compared to the seek time and rotational latency.

The **access time** is the time it takes to perform the entire operation.

$$T_{access} = T_{avg\ seek} + T_{avg\ rotational\ freq.} + T_{avg\ transfer} \tag{13}$$

RAID

RAID is a way to organize more than one disk to improve various aspects of data storage. RAID exploits parallelism and redundancy to improve performance and reliability. On the downside, they are more expensive.

RAID stands for redundant array of inexpensive disks. There are different types of RAID systems.

RAID 0

RAID 0 just spreads data over several disks. This improves performance but there is no redundancy.

RAID 1

RAID 1 has data disks and mirror disks that are exact copies of the data disks. This approach means you need twice as many disks as you would normally but it means that there is redundancy and a performance boost.

RAID 2

RAID 2 has data disks and error correction disks that contain redundant data. An error correcting code is used to recover data. This system is not usually used in practice.

RAID 3

RAID 3 has n data disks and 1 redundant disk. The redundant disk contains a byte that is the XORed result of all the data disks' bytes in the same location. This approach allows you to recover data as long as only one disk fails. Reading is quicker but writing is slow because a new parity bit must be written.

RAID 4

RAID 4 is exactly like RAID 3 but instead of XORing bytes, RAID 4 XORs entire blocks from the data disks. The parity disk is still a bottleneck for writing.

RAID 5

RAID 5 is like RAID 4 but the parity blocks are distributed across all the disks. This helps reduce the bottleneck of change the parity bits when writing.

RAID 6

RAID 6 is like RAID 5 but there are two different XORed values stored instead of just one.

Nested RAIDs

RAID systems can also be nested. An example is RAID 1+0. RAID 1 means there are exact copies of the data disks. And RAID 0 means the data is distributed across more than one disk. RAID 1+0, then, has data distributed across more than one disk but also has redundant disks for each disk.

Cache

Caching is a very important concept that must be understood to write programs that have good performance.

Locality

Locality is used to describe programs' use of the cache.

Temporal Locality

Temporal locality is when a recently used item is likely to be used again very soon. This is good because it means the values with temporal locality can be stored in cache which is much faster to access than the main memory.

An example of temporal locality is a local sum variable that is increased in a loop.

Spatial Locality

Spatial locality is when the next item read is very close to the current item. This is good because caches usually store entire blocks of data at a time rather than single bytes.

An example of spatial locality is accessing arrays in a sequential fashion.

For reference, C, C++, Mathematica, Pascal, and Python all store 2D arrays row by row. Fortran and MATLAB store 2D arrays as column by column. Java stores references to arrays in another array.

Generally speaking, in C, the right most index of an array should be changed the most often.

Locality & Performance

Locality can have a huge impact on performance. I/O is usually the biggest bottleneck in modern programs.

An illustration of how important locality is a memory mountain.

Note that the **read throughput** is the rate of bytes read. The **stride** is the step on each iteration, e.g. a stride of one means sequential reading.

It is clear on the memory mountain that a loop using spatial locality (ij) is much faster than the reverse (ji) even when the working set is large.

To get optimal performance, use locality whenever possible, i.e. keep the working set small and use small strides.

Memory Heirarchy

The memory heirarchy is a way to describe the heirarchy of small, expensive memory to large, cheap memory. A specific memory type is used as a cache for the memory right below it in the memory heirarchy.

Hits & Misses

The way caches work is that when some data is used from memory, its block is also copied into cache for later use.

But to describe how a cache is used by a program, we use the idea of hits and misses.

Hits occur when the block we want is contained in the cache.

Misses occur when the block we want is not in the cache.

There are a few different types of misses:

- **Cold misses** occur when the cache is empty.
- **Conflict misses** are a little more complicated. Caches usually store blocks of data rather than just singular bytes. These blocks are usually organized as block 1 containing bytes 0-3 and block 2 containing bytes 5-8 and so on. Conflict misses occur when you constantly refer to different blocks every time.
- **Capacity miss** is when the **working set**, data currently being used, is larger than the actual cache capacity.

Local miss rate is the $\frac{misses}{requests_{local}}$. The local requests are the requests made only to that cache.

Global miss rate is the $\frac{misses}{requests_{global}}$. The global requests are the requests made by the processor to any cache level.

Organization

Caches are organized into sets, lines, and blocks.

Each cache contains S sets where $S = 2^s$. Each set contains E lines where $E = 2^e$. Each line contains B bytes where $B = 2^b$. Each line also contains a valid bit and a tag.

In this fashion, the capacity of a cache is as follows: $Capacity = S \times E \times B$.

The address of a location in cache has three sections:

- A **set index** is used to identify which set you want.
- A **tag** is used to identify which line you want in a set.
- A **block offset** is used to offset from the first byte in a block.

Clearly, the number of bits for the set index, tag, and the block offset in an address depend on how large the cache is and how it is organized.

E-way associative caches describe the number of lines that a cache's set contains. Direct mapped caches only contain one line (E = 1).

As an example we can look at a direct mapped cache. It has four sets and two bytes per line. And because it is a direct mapped cache, it only has one line per set. A fully associative cache has a single set which means the address need not contain the s bits.

In this example, we only need two bits for the set number. We need a single bit for the byte offset. And a single bit for the tag. The total address can be only four bits long.

Simulating a Cache

There is a specific process that the cache goes through when it is trying to find a specific block.

Steps:

1. Use set number to find the set.
2. Check the tag against each valid line in set.
3. Use the block offset in the correct block and return the byte(s).

What if we have a cache miss? Generally, an old block is evicted if no empty space is available and the data from memory is loaded into this newly evicted spot.

Cache Writes

What if we want to write to memory but it's in the cache. What do we do?

We have two options:

- **Write through** immediately writes the block to memory.
- **Write back** modifies the cache version of the data and defers the write until the cache block is about to be evicted.

Write through is simpler to implement but it is slower and requires more bus transactions.

What about the case when the block we want is not in the cache. We have two options here as well:

- **Write allocate** loads the block into cache and modifies it in the cache only.
- **No-write allocate** modifies the block directly in memory.

Generally, write through and no-write allocate are paired together and write back and write allocate are paired together.

Types of Caches

There are two different types of caches in modern CPUs:

- **d-cache** stores the program data.
- **i-cache** stores the instructions.

Sometimes a **unified cache** is used that contains both data and instructions.

Performance

There are a few different measures we can look at for performance.

$$CPU_{time} = \frac{CPU clock cycles for a program}{Clock rate} \tag{14}$$

$$CPI = \frac{CPU clock cycles for a program}{Instruction count} \tag{15}$$

$$CPU_{time} = Instruction count \times Cycles per instruction \times Clock cycle time \tag{16}$$

$$CPU_{time} = \frac{Instructions}{Program} \times \frac{Clock cycles}{Instruction} \times \frac{Seconds}{Clock cycle} \times \frac{Seconds}{Program} \tag{17}$$

Note that usually different instructions use different clock times. So the most accurate method for calculation CPU_{time} is to use a sumamtion of each instruction and its clock cycles.

Another important measure is average memory access time.

$$AccessTime_{average} = HitTime + MissRate \times MissPenalty \tag{18}$$

The miss penalty comes from having to go to memory and get whatever data you want. If there's a second level cache, then the miss penalty is just the $HitTime_{L2} + MissRate_{L2} \times MissPenalty_{L2}$.

Another complication is when it takes longer to return a certain block, the miss penalty becomes dependant on the time it takes to return the block. Let's say that it takes 4 clock cycles to return 16 bytes and we have 64 byte blocks. The miss penalty will be $MissPenalty = BasePenalty + \frac{64}{16} \times 4$.

Improving Performance

There are a few different tactics to improving the cache performance:

- Bigger block size - increases spatial locality but takes longer to update block after cache miss so increases penality.
- Bigger caches - more likely to find what we want in the cache but it is more expensive and the hit time is larger.
- Higher associativity - is similar to bigger caches with the same drawbacks.
- Multilevel caches - help reduce the overall access time. The miss penalty for the first level is the access time and the miss penalty for the second level and so on.

There are some more advanced optimizations as well:

- Small and simple first level caches - minimize hit time by increasing associativity.
- Way prediction - predict which information might be used next. It can make a massive impact on performance.
- Piplined cache
- Non-blocking caches - allow hits in the request queue to be finished before going to fetch memory for misses.
- Multiple banks - allow simultaneous access to bytes speeding up hit time.
- Critical word first and early restart - send a critical word because of a cache miss first and then process the cache miss as normal.
- Merging write buffer - if writing to a block that's already in the write buffer, just update the write buffer for now.
- Hardware prefetching - get data before it is called on. Although, misses could reduce performance.
- Compiler prefetching - compiler adds prefetching instructions before data is needed.
- Compiler optimizations - things like loop interchanging and blocking can help improve performance. Blocking subdivides a 2D array into a small block improving locality.

Virtual Memory

Only very low level devices directly refer to physical addresses of memory. This approach is fine for fairly simple devices but can be dangerous for complex devices. What if one program accidently writes over the memory of another program.

We need something to protect different processes from one another. Virtual memory is what modern OSes use to help abstract away that problem.

We can even have multiple processes reference and use the same physical memory.

Organization

Instead of seeing physical addresses, processes see a contiguous block of memory that only they have access to.

Each process may have something like 4 GB of virtual memory. Clearly, there is much more virtual memory than physical memory. The solution is that virtual memory uses the HDD. But to talk more about this we need to talk about pages.

Virtual pages are fixed sized blocks of memory. Each virtual page is P bytes where $P = 2^p$. **Physical pages** are the same size as virtual pages but these are stored in physical memory.

Because data is stored on the HDD, the miss penalty is massive. To deal with this, pages are usually fairly large, 4-8 KB. This helps with caching.

Frame is just another name for physical pages.

There is a piece of hardware called the **memory management unit** (**MMU**) that converts virtual addresses to physical addresses using a look-up table stored in memory.

To do the actual translation, the MMU uses **page tables**. These are unique to each process and contain **page table entries** (**PTE**). The page tables are usually cached in L1 cache. But sometimes, the page tables have their own hardware cache in the MMU called the **translation lookaside buffer** (**TLB**). The TLB is faster to access than the L1 cache further speeding up virtual memory.

Each page has a PTE. Each PTE contains a valid bit and an address. Extended page table entries can contain information about permission, read, and write priveliges. If a process tries to do something that isn't allowed, it will throw a general protection fault.

If the valid bit is 1, then the address is of the physical frame in memory. If the valid bit is 0, then the address is of the data on the HDD or the address is null and the page is unallocated.

When the MMU encounters an address that is not cached, it is called a **page fault**. In this case, another page is evicted from memory and replaced with the non-cached page.

Caching

There are three categories that a virtual page can belong to:

- Unallocated - pages contain no data and are not stored anywhere.
- Cached - pages contain data and are kept as physical pages in memory.
- Uncached - pages contain data but are stored on the HDD instead of memory.

The reason virtual memory isn't incredibly slow is because of locality. Generally, the working set is much smaller than the total memory size. If the sum of all the processes' working sets is greater than the main memory size, we get **thrashing**. Thrashing is when performance just dies and pages are just constantly swapped in/out.

Addresses

If virtual memory is separate from physical memory, then their addresses cannot be the same.

An actual virtual address has two parts:

- **VPN** - virtual page number.

- **VPO** - virtual page offset.

As noted earlier, the VPO and the PPO (physical page offset) are identical because the pages are stored as one block.

Process

The basic process of creating virtual pages is as follows:

1. Process requests a page of memory from OS.
2. OS creates a new virtual page.
3. OS finds a free frame (physical page).
4. OS stores page in frame.
5. OS sends confirmation to process.

The process for using virtual pages is as follows:

1. CPU sends virtual address to MMU.
2. MMU converts to physical address.
3. MMU checks if page is unallocated, cached, or uncached. If the page is unallocated, an exception is thrown.
4. If the page is cached, then the MMU return a translated address in memory.
5. Uncached pages are loaded from HDD to an empty or newly evicted frame and the look-up tables are updated.

Note that whenever we evict a page, the page will be updated back in the HDD if it has changed.

Floating Point

Because it's incredibly inefficient to store large binary numbers in their complete form, the floating point system was invented to store large numbers.

Structure

Any number that can be written as $A_2 \times 2^n$ can be written in floating point format.

There are three components to a floating point number:

1. Sign bit
2. Exponent bits
3. Fraction bits

The first bit in any floating point number is the sign bit.

The k value usually indicated the number of bits used for storing the exponent.

The n value indicates the number of bits used for storing the fraction.

Different Forms

There are three different forms of floating point numbers.

Normalized Form

Normalized form is the first out of three different forms of floating point numbers.

All numbers in normalized form can be written as follows:

$$(-1)^{b_{sign}} \times (1.fraction)_2 \times 2^{k-bias} \tag{19}$$

$$bias = 2^{k-1} - 1 \tag{20}$$

Note that the 1. is always constant in normalized form.

Normalized form is for large numbers.

Denormalized Form

Denormalized form is for small numbers. In denormalized form, the exponent must be **all zeros**.

All numbers in denormalized form are written as:

$$(-1)^{b_{sign}} \times (0.fraction)_2 \times 2^{1-bias} \tag{21}$$

The bias is identical to normalized form.

Special Form

Special Form is used for representing infinity and `NAN` which stands for "Not a Number". In special form, the exponent must be **all ones**.

If the floating point number is in special form, then there are two cases:

1. $fraction = 0$
    - This equals infinity.
    - Note that depending on the sign, we can have positive or negative infinity.
2. $fraction \neq 0$
    - This is the `NAN` case.

Example

For an example, let's try to encode `0`.

We can't use normalized form because it has a `1` sticking at the front of the fraction part. It's not a special case so we don't need special form.

We use denormalized form.

$$0 = (0.0000) \times 2^{1-bias} \tag{22}$$

And we can represent a zero no matter what the bias is.

Note that we can have positive or negative zeros just like we can have positive or negative infinity.

Rounding

Because some numbers cannot be represented in floating point format, we will occasionally round numbers.

The method used most often is the round-to-even method. The idea is the same as regular rounding rules but if we're right in the middle of two numbers such as 2.5 then we round to the even number so 2 in this case.

In binary, ones round up and zeros are even.

The other methods of rounding are:

- Round-up
- Round-down
- Round-toward-zero

Those are all self-explanatory.

Limitations

Remember that floating point can only represent numbers that can be written as $A_2 \times 2^n$.

This means that some numbers cannot be represented accurately.

Another limitation is that floating point operations are not associative, i.e. $a + (b + c) \neq (a + b) + c$. Floating point operations are, however, commutative, i.e. $a + b = b + a$.

Instruction Architecture

Instructions sets are the lowest level of software. All other software gets compiled to the binary instructions which are read by the hardware.

Instruction Structure

Instructions are separated into an opcode and data. The **opcode** is the instruction type and the **data** is the actual data being used for the instruction.

Number Of Operands

Instructions usually have two or three operands.

Generally, instructions with three operands are for instructions like MUL, ADD, SUB, DIV, etc.

Two operands are for STORE, LOAD, etc.

Size Of Instructions

The size of instructions is usually dependent on many factors. Things like the size of opcodes, number of addressing modes, etc. all affect how large an instruction may be.

Encoding instructions is also tricky. Designers must decide between variable and fixed sizes. Variable method allows all addresses to be appended to the instruction whereas fixed sizes always have the same number of operands.

Instruction Set Choices

When designing an instruction set, a few issues need to be considered.

What operations should be implemented and how many different types. The number of operands, if any, an operation takes also needs to be decided.

General Purpose Registers

In modern computers, general purpose registers dominate.

Although registers are very quick to access, we can't have too many of them because a lot of registers would mean longer access times and longer addresses.

Instruction Set Classes

There are a few different types of classes for instruction sets:

1. Stack
2. Accumulator
3. Register memory
4. Register register/load store

All new computers use the register register/load store class. This is because the register register class is often the fastest and uses less memory transactions.

Stack

In this type of instruction set, all temporary results and operands are added and retrieved from the top of a stack.

E.g. C = A + B would be implemented as follows:

1. push A
2. push B
3. add
4. pop C

Accumulator

This is a slightly weird class. It's as if there is only one register you can use so all operations must be performed within that register called the accumulator.

E.g. C = A + B:

1. load A
2. add B
3. store C

Register Memory

This is when operands use one register and a memory location.

E.g. C = A + B:

1. load R1, A
2. add R3, R1, B
3. store R3, C

Generally, this class allows fairly dense instructions but encoding instructions is more complex than register register.

Register Register

This is the class that's is, by far, the most common today. It uses registers for operands.

E.g. C = A + B.

1. load R1, A
2. load R2, B
3. add R3, R1, R2
4. store R3, C

This class has fairly simple encoding with a fixed size but it does take more instructions that some other classes.

Memory Memory

This class can operate directly on memory locations.

E.g. C = A + B

1. add C, A, B

This is the most compact class. But, the instruction size and work per instruction can vary quite a bit.

RISC v. CISC

RISC stands for reduced instruction set computer. **CISC** stands for complex instruction set computer.

CISC allows for many different types of instructions but it is more complicated to implement than RISC. CISC is also harder to optimize.

Luckily, nowadays, designers need not choose between RISC or CISC because of an idea Intel came up with some time ago. Intel implemented a converter for converting CISC instructions to RISC inside their processors.

This meant that RISC v. CISC made no difference to the actual performance. Both got roughly the same performance.

Memory

Another big part of designing instruction sets is deciding how the instructions can refer to memory and how memory is stored on a given architecture.

Ordering

In physical memory, multi-byte data can be stored in one of two ways:

1. Little-Endian
2. Big-Endian

Most of the time, ordering is irrelevant to programmers. The only time it really becomes relevant is when computers with different ordering interact with one another.

Little-Endian The LSByte of the multi-byte data is stored in the lower address.

Big-Endian The MSByte of the multi-byte data is stored in the lower address.

Alignment

In most architectures, you cannot place a byte at any address. There are often restrictions on what can be placed at what address.

Generally speaking, data is aligned if $Mmods = 0$ where M is the memory address of the data and s is the size of the data in bytes.

To make data aligned, the compiler with often add gaps in the memory.

Structures can be aligned as well. They are just treated as large data blocks but they follow the same rules as the small data blocks.

Data inside structures also has to be aligned. This means there are often gaps littered throughout a structure. This can be troublesome for storing things like an array of structures.

We can try to manually increase the compactness of structs by placing the biggest data first.

Addressing Modes

Addressing modes are about accessing memory in different ways.

The whole purpose of different addressing modes is to reduce the number of instructions necessary to achieve a task. Allowing many different instruction modes allows denser instructions.

The most popular instructions are:

- Displacement is just accessing memory at a certain displacement x from a given address in a register.
- Immediate is just hardcoding a value in the instruction.
- Register indirect is used to get an address from inside a register. Displacement is often used together with register indirect.

Pipelining

The essence of pipelining is to separate a task into stages that require different resources. Then, we try to complete any stage we can at any given time.

A simple example is doing laundry. The **sequential** version is to wait until the entire first load is done to start the second load. The **pipelined** verison is to load the second load in the washing machine when the first has moved to the dryer.

Pipelining reduces overall time spent by doing things simultaneously. But, pipelines are limited by the slowest stage.

There is usually **overhead** in pipelining that comes from storing the results of a prior stage so that the input to a later stage doesn't change halfway through a stage.

Note that instruction level parallelism (ILP) has not been the focus since 2005. It became too complex. The goal is now to work on thread-level parallelism with multicores.

Formulas

There are some useful formulas for dealing with pipelining.

Note that N is the number of stages.

$$T_{cycle} = T_{slowest\ stage} + T_{overhead} \tag{23}$$

$$T_{total\ instruction\ time} = T + N \times T_{overhead} \tag{24}$$

$$ideal\ speedup = \frac{T_{unpipelined}}{T_{overhead}} \tag{25}$$

Note that the $CPI_{avg} = 1$ because the clock is based on the slowest stage and the pipeline is limited by the slowest stage.

Sometimes, a stall factor can affect the CPI.

MIPS Example

In the MIPS architecture, instructions have five stages:

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execute (EX)
4. Memory Access (MEM)
5. Write Back (WB)

Hazards

Pipelining isn't necessarily straight-forward. There are three main hazards to pipelining:

- Structural hazards
- Data hazards
- Control hazards

Structural Hazards

Structural hazards occur when two or more instructions require the same resource. In this case, the best approach is to stall an instruction for a cycle.

E.g. let's say data references are in 40% of all the instructions. The CPI without any hazards is 1.

The CPI with the hazards is $1 + (0.4 * 1) = 1.4$. To clarify, it is $CPI_{no\ hazard} + (Percent_{hazard} * CPI_{no\ hazard}$.

To fix structural hazards, we can add more hardware to allow more instructions to use the same type of resource at the same time but this is a costly approach. Plus, structural hazards are relatively rare.

Data Hazard

Data hazards occur when a later instruction needs data that is updated in a previous instruction.

RAW Hazard RAW stands for **read after write** hazard. An example might an instruction that stores a result in R1. The next instruction uses R1 as an operand. In this case, R1 might be read in the second instruction before it was updated in the first instruction.

WAR Hazard WAR stands for **write after read** hazard. We could also have a situation where the previous instruction is reading a register that the next instruction is changing.

WAW Hazard WAW stands for **write after write** hazard. We could have another situation where we write to a register in two consecutive instructions.

To deal with these data hazards we can have something called **forwarding**. Forwarding allows forwarding of updated values directly to other instructions' stages that need it instead of waiting to be written to a register.

Note that even with forwarding, we can have hazards. Sometimes, we still need to have some stalls in between instructions.

To improve the CPI with data hazards, we can also do scheduling and hardware speculation.

Control Hazard

Control hazards occur when the program flow gets to branches which may or may not be taken.

There are a few solutions to this problem.

We could redo the fetch for the instruction after the branch. Note that this is possible because the next instruction will be fetched when the branch is decoded. But this approach is very detrimental performance wise.

Another solution is to predict that the branch will not be taken and proceed as normal. If the branch is not taken, the pipeline will be working well but even if it is taken, we didn't waste any time.

We could also predict that the branch is taken and if we had the target address of the branch before execution, we could start pipelining destination instructions.

Another solution is to put a useful instruction that needs to be executed regardless of whether the branch is taken after the branch instruction. This approach is similar to predict not taken.

Another means of improving performance is to get better at predicting whether a branch will be taken or not. The key to doing this is noticing that individual branches will heavily favour being taken/not taken.

We can also do loop unrolling to help reduce the CPI and increase performance.

Scheduling

We can use scheduling and pipelining together to improve performance.

The idea behind scheduling is to reorder instructions such that the performance is improving with no change in the results.

Static Scheduling

Static scheduling is done by the compiler.

Dynamic Scheduling

Dynamic scheduling is done by hardware.

Dynamic scheduling does not need to know about the architecture and it works for cases where dependencies are not known at compile time.

The disadvantages are that it is fairly complex to implement and it complicates exceptions.

Another complication is that exceptions must occur in order even if the instructions are not in order.

Tomasulo's Algorithm

Tomasulo's algorithm is a way to implement dynamic scheduling.

There are three steps to every instruction:

1. Issue
2. Execute
3. Write

First, we issue the instruction to an empty reservation station for that type of instruction. If the operands are in registers, we copy the value, if possible, or we just refer to another reservation station that will produce the required value. If no reservation station is available, we need to stall.

Second, we execute the instruction if all the operands are available. If any operands are not available, we monitor the common data bus which will place the required value into the reservation station when it is available.

Third, the result is ready so we write it to the common data bus so that it can update all the registers, buffers, and stations that require that result.

Note that Tomasulu's algorithm finishes instructions out of order.

Hardware Speculation

We can improve Tomasulo's algorithm but make it handle control hazards better if we can execute instructions out of order but commit them in order.

When an instruction executes, it is kept in a **reorder buffer** instead of writing to a station. This means that we never update any registers with the wrong values because the results are never committed.

The ROB, reorder buffer, stores four fields:

- Instruction type
- Destination
- Value
- Ready

For an instruction to be committed, it must be at the top of the ROB because the ROB behaves like a queue. This makes it easy to discard speculative results and incorrect branch predictions.

The four steps of the speculative version of Tomasulo's algorithm are:

1. Issue
2. Execution
3. Write result

4. Commit

The differences between the speculative version and the non-speculative version are that write result doesn't change values in registers but does send an updated value to the CDB.

This version of Tomasulo's also avoids imprecise exceptions. All exceptions occur in order.

Loop Unrolling

We can improve things even more after scheduling for loops.

The bottleneck after scheduling in a loop becomes the overhead instructions. The idea behind loop unrolling is to take many iterations of a loop and combine them together in one longer iteration to reduce the overall overhead.

The downsides to loop unrolling are that the code is larger and we need more registers for unrolling more iterations.

Dynamic Branch Prediction

This section is important enough to deserve its own section. As said above, if we can predict whether a branch will be taken or not taken, we can improve pipelining significantly.

One simple way to do it is to remember whether a branch was taken last time. We can keep an array of prediction where the correct index is based on the lower bits of a branch address.

1-Bit Prediction Scheme

Let's keep a one bit record of the last time we got to that branch. We invert the bit any time the prediction is wrong.

We predict branch taken if the prediction bit is 1 and branch not taken if the prediction bit is a 0.

This works very well but if there is only one branch not taken in a series of taken branches, this approach will predict incorrectly twice.

Once when the branch is not taken and it predicted taken but again on the next one because it will predict not taken since the bit would be flipped.

2-Bit Prediction Scheme

This approach is very similar to the 1-bit approach but with one advantage. The two bit approach needs to predict incorrectly twice in a row to change its prediction.

Let's take the previous example with a series of taken branches with one not taken. With a one bit approach, we guessed incorrectly twice.

But with a two bit approach, just one incorrect prediction won't change our prediction so the branch right after the not taken will still be predicted to be taken.

Correlated Prediction Scheme

A more complicated type of prediction scheme is to consider whether other branches were taken or not.

Let's look at the last two branches. So we could have the following combinations:

- taken, taken
- taken, untaken
- untaken, taken
- untaken, untaken

For each of these combinations, we have a different predictor.

For a (m, n) predictor, we choose from 2^m branch predictors (combinations) and there is an n-bit predictor for a single branch. Note that m is the number of previous branches we look at and n is the number of bits we use as a prediction scheme.

Basically, we keep 2^m branch combinations and each combination has a certain number of prediction entries. These prediction entries work the same way as the prediction entries for the simpler schemes.

We select the correct branch combination using a global branch history register.

Tournament Predicton Scheme

There is an even more complicated prediction scheme that we can use.

So far, we've only talked about local and global prediction schemes separately. The one and two bit prediction were local prediction schemes and the correlated branch prediction was a global prediction scheme.

We can have a new more complicated prediction scheme that chooses between the global and local prediction scheme depending on which one is more accurate.

A two bit tracker is used similarly to the two bit prediction scheme to track whether to use the global or local prediction scheme.

Branch Target Buffers

BTB are buffers that are like a cache for branch targets for cases when the branch is predicted taken.

This helps pipeline the instructions at the destination PC to continue seamless pipelining.

We can also do a variation of this by storing the target instruction instead of the address. This way, it's even faster.

Exceptions

Exceptions change the flow of the program for special cases.

Asynchronous Exceptions

These exceptions, also called interrupts, are caused by things outside of the processor.

These may be caused by things like the reset button, or IO actions.

Synchronous Exceptions

These are caused by instructions from inside the processor.

These are things like traps, faults, and aborts.

Traps These are intentional exceptions such as breakpoint traps, etc. They return control back to the next instruction.

Faults These are unintentional but sometimes recoverable. These are cause by divide by zero, page faults, etc. These exceptions may re-execute the faulty instruction or stop execution.

Aborts These are unintentional and unrecoverable. These abort the program and stop execution.