Introduction to Computing and Systems Architecture

1. Computability

A task is computable if a sequence of instructions can be described which, when followed, will complete such a task.

This says little really. Therefore, we use Turing machines to allow us to more formally describe computability.

Turing Machine

The machine can attain a finite number of states. A number of transition rules exist that help govern the actions of the machine. A tape of infinite length exists which contains symbols (laid out in a linear manner). The tape can be read from (one symbol at a time) or written to (one symbol at a time) by the machine's read/write head. The state of the machine together with the symbol under scrutiny (position of the tape), and the transition rules present dictate what the machine will do next. For example, if we are in state S1, and we read symbol M1, then move to state S2 and move read/write head one to the left.

By definition a Turing machine (as just described) is deterministic in nature. There also exists non-deterministic Turing machines. However, these are more for the theorist at this moment in time. They are very interesting though.

A machine, which can be fed data and act upon such data by following rules to produce outcomes that are deterministic are all the ingredients we need to start programming.

Computer

Going from a Turing machine to a computer is a big engineering leap, but not a great theoretical one. One change, theoretically speaking, is a limitation on resources (finite).

Consider a very simple computer made up of a CPU that has a list of instructions encoded within it (instruction set) and has access to memory which can hold data. The memory can be accessed by the CPU at any place (random access). The CPU reads memory locations in sequence, and decodes the data using its instruction set and does something (either write data back to memory or read data from memory). Now, depending on what the CPU is doing (its state) the CPU may decode the memory as instructions (as mentioned) or values. For example, add memory location 1 to memory location 2 and put the answer in memory location 3 has a few instructions and a few values.

Now, this seems easy. What could possibly go wrong? Well......

What if the CPU reads values and thinks its instructions? What if the CPU writes values into memory locations that already contain something (possibly instructions)? What if the person who wrote the program doesn't know what they are doing and writes values over the instructions that are responsible for the correct operation of the computer's basic systems (operating system)?

2. Engineering speed (and complications)

Geography and magnitude:

It is much quicker to reach out to the table and find out your friends number on your iphone, but takes a lot longer to access the global directory of phone numbers from the Internet.....

Getting data from memory is very slow compared to the CPU so let's get a chunk of memory and store it close by to speed things up (cache).

Direct access

Sometimes it is much quicker to simply google than to ask the teacher.......

The CPU is not the only piece of hardware around in a modern computer, however, traditionally it is the only thing that knows how to handle memory. Let's break this tradition and allow other things to access memory directly for reading and writing. Disk drives, graphic cards and many other devices need direct memory access (DMA). - (Multi-core use it as well)

Parallel processing

Two hands are better than one.....

Multi-processor and multi-core allow more than one processing element to simultaneously operate on the memory (either their own or shared). Sometimes the computer can automate the parallel process, but sometimes the programmer has to do it.

Single Instruction Multiple Data (SIMD) is when an instruction can be applied to many pieces of data at the same time. If you have 50 cars and you want to apply the same velocity function to all of them this may be for you (note. check PS3 cell architecture).

Multiple Instruction Multiple Data (MIMD) is when many instructions can be applied to many pieces of data at the same time.

Yes, you may have SISD and MISD.

3. Caches

In general a processor has a small amount of memory attached to it, with very fast access. This is known as a *cache*. The processor stores a copy of the most recent data it has accessed in the cache. This means that, if the processor needs to read that data again, it is accessed considerably faster than if it is accessed from main memory again.

The time it takes a processor to read a piece of data from memory is known as *latency*. In effect, the processor has to wait until the data becomes available to it. Obviously the smaller the latency, the faster the processor can do its job. Reading from a cache has a much lower latency than reading from main memory, as the cache is a piece of physical memory attached to the processor specifically designed for fast access.

If a requested piece of data has been used recently then it is already in the cache, giving a much faster latency when accessing it. If the data is not in the cache then what is known as a *cache miss* occurs – ie the data has not been found in the cache, so it has to be accessed from main menu, and will be copied into the cache in case it is needed again. Code which needs to run as optimally as possible is designed so that it reuses data in manageable chunks. A thoroughly optimised piece of code will work on one section of data, with no cache misses (ie no dependencies on data from other parts of memory), before moving on to another chunk of data which will, in turn, fill up the cache for reuse.

A cache consists of a set of rapid access memory locations, known as *cache lines* – each line contains:

- The address in main memory where the data has been copied from (or the *tag*).
- The data itself.

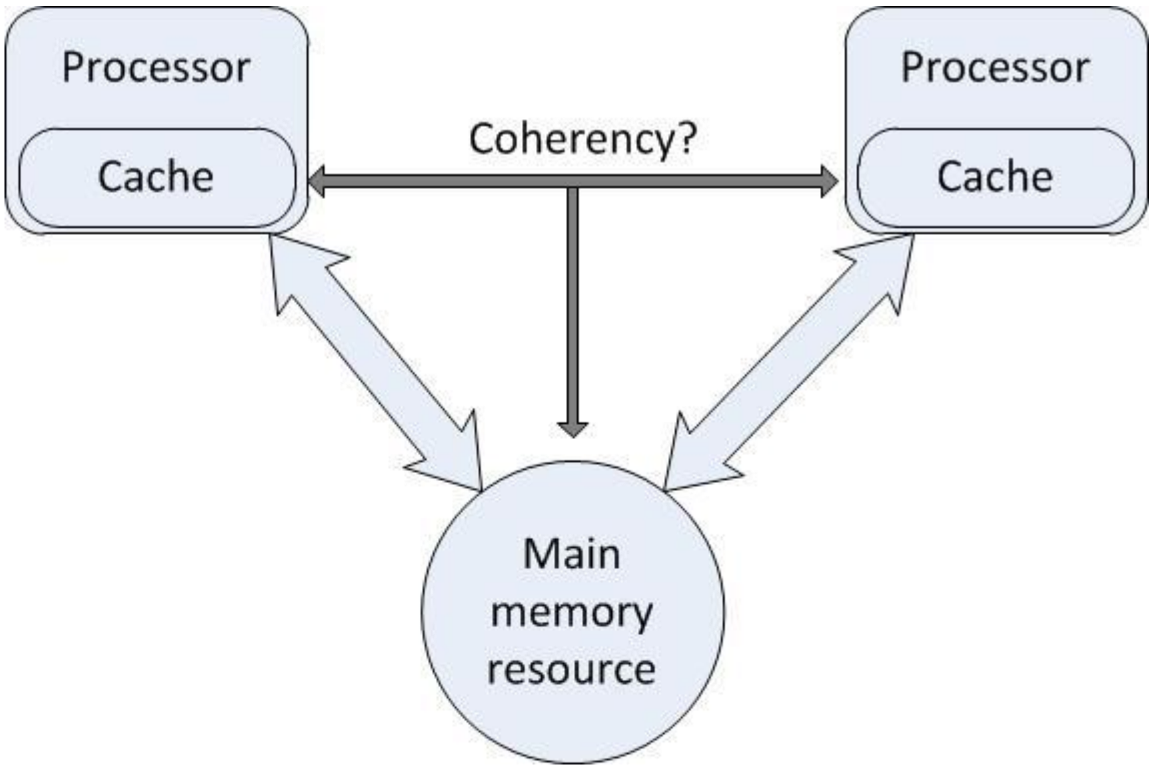
Cache sizes vary from processor to processor, which means that highly optimised code for one processor may result in cache misses, or the opportunity for further cached data, on another processor.

It is important to note that the cache is not physically part of main memory, and that it contains copies of the data held in main memory. This means that, if the data in main memory is changed, the cached value will not reflect that change – i.e. the cache is not a set of pointers into main memory, it is a recent copy of the data in main memory.

Cache Coherence

This approach works nicely in a single processor system. However if there is more than one processor at work in the system (for example a CPU and a GPU), then an issue arises. We need to consider what happens if the data which has been copied from main memory into the cache of one processor is changed in main memory by another processor. Further to this, it is possible that two processors could have copied the same data from main memory into their respective caches – again there is an issue if that data has been copied at different times and each processor is working with different values of the data.

There is therefore a need to ensure that cached data is consistent with the data in main memory that it is copied from, and also that the caches of each processor can handle inconsistencies in matching data. This is known as *cache coherence*.



In order to resolve any inconsistencies between cached memory accesses, a *coherency protocol* is required. Various protocols exist to maintain coherency between distributed caches in a multi-processor system.

Sequential Consistency

This is an early type of coherency protocol. The protocol requires that the results of the instructions carried out on multiple processors are identical to the results if the instructions had been carried out sequentially by a single processor. To achieve this, the protocol must ensure that:

- All main memory requests (both read and write) are carried out in the order specified by the control program
- Each cache has a single FIFO queue for memory requests (ie first in, first out).

It can be seen that this ensures that each cache contains the most recent version of data as it is being processed – another processor requesting that data can't read it until the first processor has written its changes back to main memory.

Release Consistency

This coherency protocol is based on the idea of a processor identifying data as being subject to change. It uses two synchronisation operations known as *acquire* and *release* - when the processor is ready to write to main memory it acquires that data, which prevents any other processor from accessing it. When it has finished writing, it releases the data, freeing up that data for other processors to acquire. It is similar to the idea of checking out a sourcecode file from a repository using SVN.

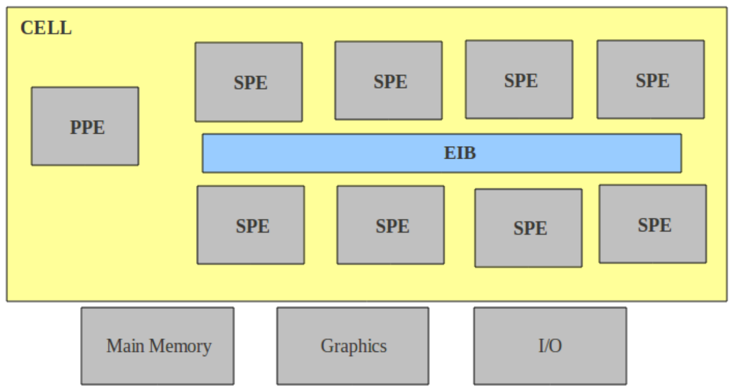
Weak Consistency

This coherency protocol allows for more processes to occur concurrently, by relaxing the rules of Release Consistency slightly. Rather than locking out other processors during the whole period from the *acquire* to the *release*, this protocol ensures that the order of acquires and releases is seen to be consistent across all processors, but the order of reads and writes between those events can be seen to vary. The sum of the reads and writes between each synchronisation event must also be the same.

4. The Cell and Playstation3

The Cell Engine Broadband Architecture is a microprocessor developed by IBM, Sony and Toshiba.

The Cell architecture is shown in the diagram below:



Note: In the PS3 one of the SPEs is "offline/locked out/lanced" and one is dedicated for use by the operating system. This leaves six left for game developers.

Overview

The PowerPC Processing Element (PPE) may be considered capable of running alone, like any regular CPU. However, in this architecture the PPE may be aided in satisfying its computational requirements by the Synergistic Processing Elements (SPEs). The Element Interconnect Bus (EIB) accommodates communication between PPE, SPEs and external components.

The PPE may control execution on the SPEs. This can take the form of scheduling processes to run and interacting with such processes (e.g., start, stop, interrupt).

Memory Access

Standard load/store commands are used throughout the system for memory access by PPE and SPEs (just like a regular CPU would use them). However, SPEs may only access their own local store (256 KB) whereas the PPE can access main memory and the memory of the PPEs. The local store of the SPEs is not cache (it is actually considered local memory), but it is high performance Static Random Access Memory (SRAM), higher performance than Dynamic Random Access Memory (DRAM) and is commonly found in cache architectures.

PPE main memory access is via cache (the PPE has L1 and L2) that utilises Direct Memory Access (DMA) for cache updates. All processing elements (PPE and SPEs) have DMA utilisation capabilities, making DMA the main focus of memory movement throughout the system. This allows a PPE to also access main memory (and the memory of other SPEs) via its local memory (using a virtual memory address to indicate the source of the “get” and a variable to indicate the size of memory to retrieve).

All caches are cache coherent (access of “stale” memory locations in main memory are avoided). However, as the SPEs' local memory is not considered cache then this may prove problematic to some programmers. This really makes programmers consider the SPEs as distinct programming elements and should be treated as such.

As an added complication (or bonus, depending on how you look at it) each SPE has a small amount of cache (512B). Cache coherence is avoided as requests from other processing units for a memory location that is actually in the SPE cache are satisfied from the SPE cache (not the local memory of the SPE). This gives a very nice, very fast, shared memory system that exhibits consistency. How you use this effectively is not clear, but it may have its uses.........

5. XBox360

The Xbox360 also has a multiprocessor architecture, consisting of:

- 3 CPU cores with a shared 1Mb cache.
- 1 GPU with 10Mb embedded cache, and 48 parallel unified shaders.

The CPU

The central processing unit is a 64-bit triple-core processor. Each core contains multiple floating point processors, and SIMD vector processing units, so the focus of the hardware design is on floating point calculations. Each core is capable of simultaneous multithreading, although they utilise in-order execution, rather than the more recent out-of-order execution. The cores share a comparatively large cache (I Mb), which is accessible at half the CPU clock speed (i.e. significantly faster than main memory).

The GPU

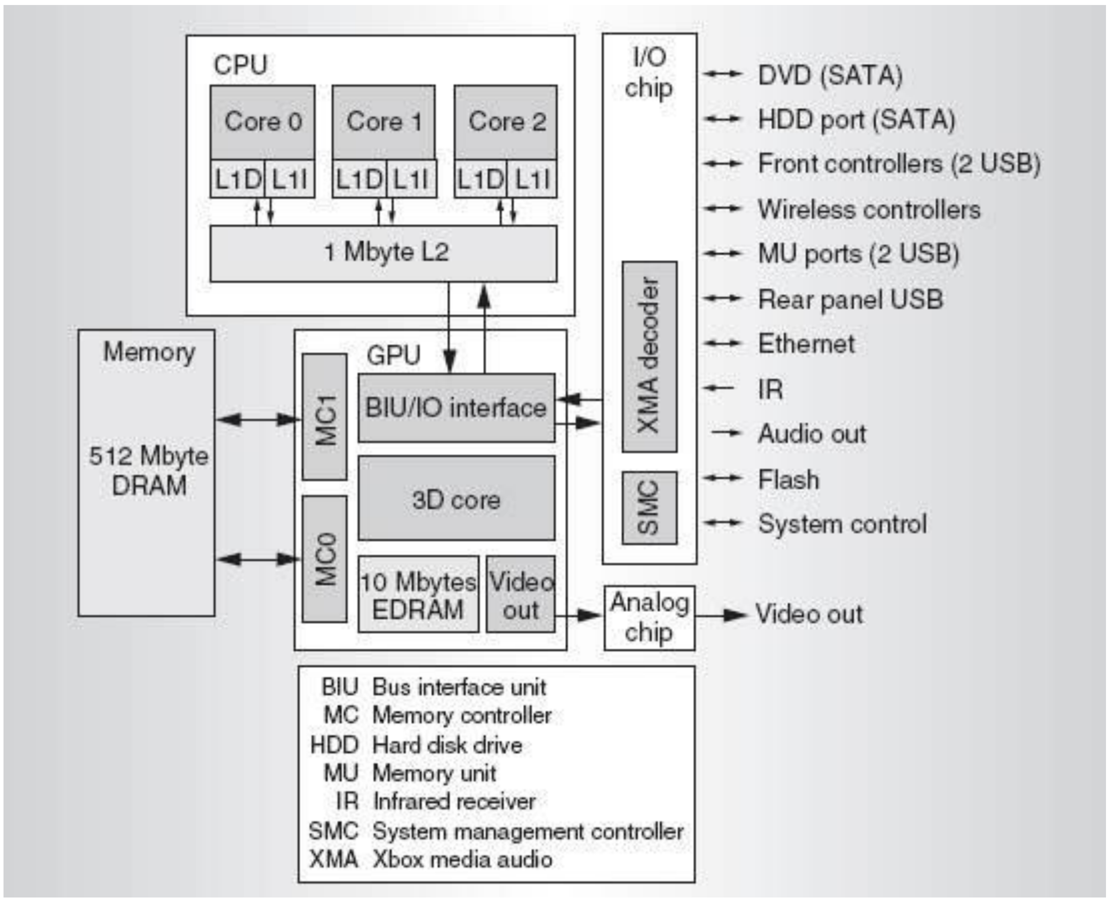
The key to the GPU’s power in the Xbox360 is the 10Mb cache, which is attached to the GPU on a daughter die. This cache is sufficiently large that graphical processes such as alpha-blending, z-buffering, and full-screen anti-aliasing can be implemented with virtually no performance cost to the GPU.

Each of the 48 shader processors consists of a 5-wide vector unit, capable of executing two instructions per cycle – i.e. each shader processor can execute 10 floating-point operations every cycle. The shader processors are arranged into three groups of 16 processors each. All processors in a SIMD group execute the same instruction, so in total up to three instruction threads can be simultaneously under execution.

Main Memory

The Xbox360 has 512 Mb of main memory. This memory is shared between the CPU and GPU, via a unified memory architecture. For development purposes, this means that there is no distinction between “graphics memory” and “main memory”, so the overall memory footprint can be designed with the needs of each game in mind.

The GPU also acts as the memory controller, although this is largely invisible to the developer.



6. Cross-platform development

Most modern games are released on multiple platforms, unless they are developed by the hardware developer’s in-house and first-party teams. There are two approaches which can be taken here:

- Develop on one platform, and port to the other platforms once the lead platform development is complete.
- Develop on multiple platforms at once, using shared technology as much as possible.

Both approaches have their advantages and disadvantages. Developing on multiple platforms can lead to a better engineered codebase (as much of it has to compile and run in more than one development environment), but can also lead to taking the “common ground” of multiple platforms, so features which are designed to fully utilise one platform’s abilities are prioritised below cross-platform functionality.

In general, the skills and approaches required to run code on one multi-processor platform are the same as for another (setting up a code architecture for multi-threading will benefit both Xbox360 and PS3 development). However getting down to the specifics of what tasks are assigned to the processors must be platform-specific. As can be seen in this document, developing code to take full advantage of the PS3’s SPU architecture with their minimal caches, is a different challenge to using the triple-core processor on the Xbox360 with its much larger cache.

Identifying which parts of a game engine need to be targeted for platform-specific development is an important skill. The key decisions are based on how often a piece of code runs and how much data it requires to run.