Disclaimer: The contents of this document are scribe notes for The University of Texas at Austin EE360N Fall 2008, Computer Architecture. The notes capture the class discussion and may contain erroneous and unverified information and comments.*

Interleaving and Intro to Virtual Memory

Lecture #8: September 24, 2008
Lecturer: Derek Chiou
Scribe: Vijay Kanumuri and Todd Riffell

1 Recap and Outline

In Lecture 7 we talked about cache hits and misses, and discussed the types of misses such as compulsory, conflict and capacity misses. We then went on to discuss replacement of data in the cache along with writing to the cache.

In this lecture, we covered interleaving and introduced virtual memory. The first part of the lecture discussed how to improve bandwidth/throughput through interleaving. The Cray-1 supercomputer was used as an example to illustrate interleaving. The next topic was endian-ness. We then discussed alignment and how to deal with unaligned accesses. We concluded with an introduction to virtual memory.

2 Final Words on caches

Cache size refers to the size of the data component of the cache only. For example, a 32KByte cache means 32KBytes of data. Unlike disk memory where manufacturers mention a GB as 10^9 bytes, 1 GB of cache memory means 2^{30} bytes. This size does not include storage for address tags, permission tags, and replacement information.

All address bits must be used to access data in the cache. If any bit is omitted, ambiguity is introduced since more than one address can map to the same location. Each bit must be used either as an address tag, set selection (sometimes known as index) or offset. But the bits need not be contiguous. For example, both mappings, shown in the Figure 1, are legal mappings.

If we use the same address bits more than once to access a cache, we have redundant information in accessing the cache, potentially causing waste. For example, if we use a bit in set selection, using that bit in the address tag simply means that address tag bit will be a constant in that set. Thus, we are wasting the storage for that address tag bit.

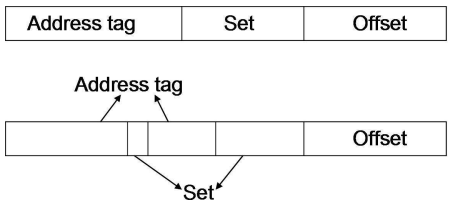

Figure 1: Address Bits

3 Improving Memory Performance using Interleaving

3.1 Latency

Latency is defined as the amount of time from when the request is initiated to when the request completes. For example the latency of a read request is the time from when the request is made to when the data returns. The latency of a write request is the time from when the request is made to when the data has been written.

As an example, if we want to buy a laptop, the latency would include the time to drive to the store, the time to buy the laptop and time to get back from the store.

3.2 Throughput/Bandwidth

Throughput, otherwise known as bandwidth, is defined as the number of things (generally bytes or bits in a computer or network) that can be processed in a unit of time. Take traffic on the freeway as an example. If we want to measure the bandwidth on a freeway, we would measure the number of cars that pass a line drawn across the freeway in one unit of time. Hence if we have 4 lanes instead of 2 lanes, we have twice the bandwidth, assuming that all cars in the 4 lanes travel at the same speed and are equally spaced.

3.3 Interleaving

Memory has long access latency, often taking multiple processor cycles to access. In addition, early memory was asynchronous, meaning that accesses could not be pipelined. Thus, when one memory operation is outstanding, others must wait until it is complete before they can start.

One way to increase throughput is by interleaving/banking. In an interleaved memory system, there are multiple *banks* of memory, each of which can process requests independently of the others. One can issue an access to one bank and then issue another access to another bank that is not currently busy servicing an earlier request. Data is *interleaved* or spread across those banks so that any particular byte of data lives in just one of those

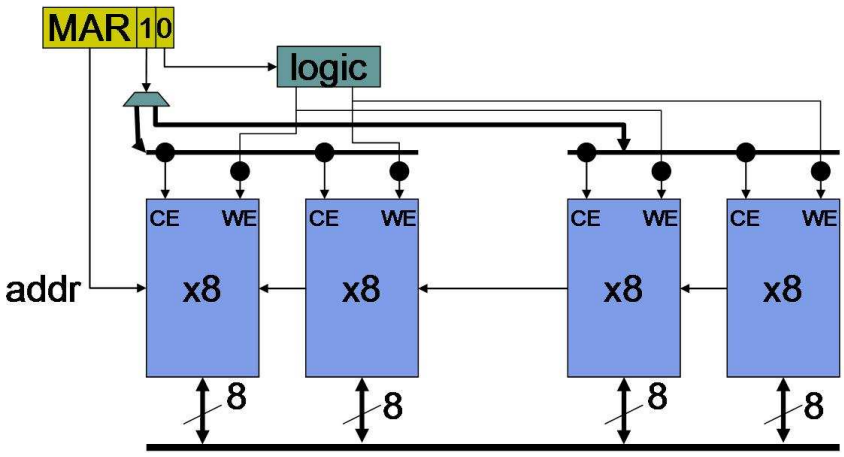

Figure 2: Interleaving of Memories

banks. Ideally, data is interleaved the same way memory is accessed since then each bank will have the maximum time to satisfy the current request before its next request arrives.

One possible implementation for interleaving memory capacity is shown in Figure 2.

In Figure 2, there are 4 memories that are each 8bits wide. Two memories are concatenated together to form a 16bit-wide *bank*. Obviously, there are two banks in the system. If the memory access pattern is sequential on 16b words, we would alternate between banks for each successive word. Thus, we would use the second least significant bit of the address to select the bank, which includes ensuring that only one bank drives the bus in the case of a read. The Chip Enable(CE) control signal can be used to avoid bus conflict since, when it is deasserted, the memory chip goes into high-Z state. Of course, tristates can also be used to ensure that only one bank drives the bus at any time. In general, with more than two banks, a decoder can be used to select which memory is allowed to drive the bus.

Interleaving using either the MSB or the LSB of an address is shown in Figure 3. Assuming word-addressed memory and a three bit address, if we use the MSB to interleave, a memory access pattern that alternates the MSB minimizes conflict. If we use LSB to interleave, a sequential memory access pattern minimizes conflict.

3.4 Byte Stores

In the LC-3b, memory can be accessed in word granularity or in byte granularity. When performing a byte store, the least significant bit of the address is used to select whether the Least Significant Byte(LSB) or the Most Significant Byte(MSB) should be written. Obviously, when performing a word store, all 16b are written to the bus. Since the LC-3b ISA requires alignment, when a word store occurs, the least significant bit of the address is simply ignored (another possibility is to cause an error when the least significant bit is not zero in the case of a word load or store.)

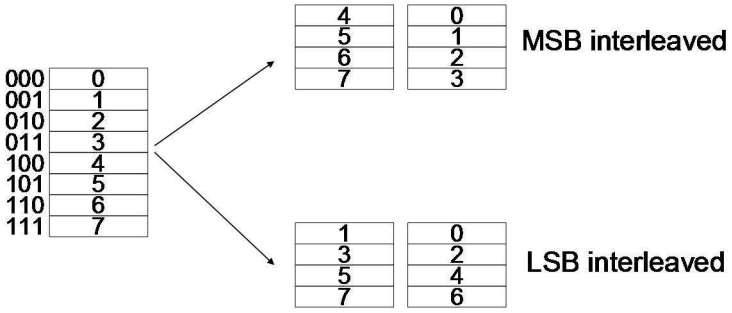

Figure 3: MSB and LSB Interleaved Memories

In a byte-addressable machine with a 16b instruction word, when incrementing the address, we must increment by 2.

3.5 Banks versus Channels

DRAMs are partitioned internally into banks as well, that can be accessed independently like the memory banks we described in the previous sections. Today, multiple DRAMs that are aggregated together into what was formally called a bank are now called a *channel* to eliminate the potential confusion with the internal banks of a single DRAM. Oftentimes, each channel consists of a single DIMM (see Lecture 5.)

3.5.1 Example: Banking in Cray-1

If there was a memory with a single bank, one outstanding request to memory would block any subsequent requests, forcing all remaining operations to wait for the present access to be completed. To reduce the chance of such blocking, the Cray-1 contains a memory system with 16 banks (in the classic meaning of the word bank.)

Cray-1 banking is shown in Figure 4.

In the Cray-1, the CPU cycle time was 12.5 ns while the memory cycle time is 50 ns. Thus, to avoid stalls due to memory, a minimum of four banks are required. Why, then, did Cray use 16 banks? One reason is to compensate for I/O using memory, requiring more banks. Another reason is to reduce the chance of memory bank conflicts when the memory access pattern does not match the interleaving.

For example, if the data is accessed sequentially, we would require only 4 banks. In this case, data would be accessed from bank 0,1,2,3,0,1,2,3..and so on. If data is read from every other address, data would be read from bank 0,2,0,2.. and so on. This access pattern would cause stalls if we only had four banks, thus requiring 8 banks for that access pattern. 16 banks enables more access patterns that won't cause stalls.

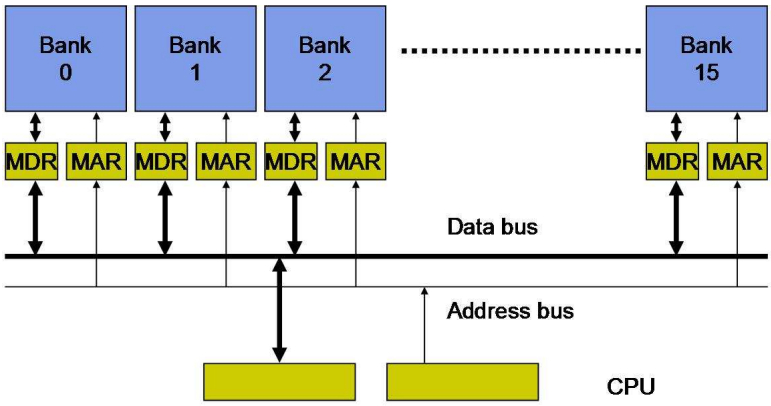

Figure 4: Cray-1 Computer

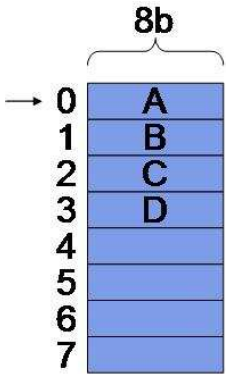

Figure 5: Endian-ness

3.6 Reordering Memory Operations

We may allow reordering of memory accesses, but we need to tag the out-of-order requests so that they can be matched with their requests when they return with data.

4 Endian-ness

Endian-ness refers to a method of storage in memory. There is an issue when a word is bigger than the addressability of a machine. For example, LC-3B is a byte-addressable machine while the registers are 16bit wide. There are two ways to resolve this issue.

4.1 Big-Endian

A big-endian machine stores the most significant byte (MSB) of the word in the first location. Hence the pointer points to the MSB of the word. For example, given the memory shown in Figure 5, $*((word\_t*)0)$ is AB and $*((word\_t*)1)$ is BC.

4.2 Little-Endian

A little-endian machine stores the least significant byte first. Hence the pointer points to the LSB of the word. For example, given the memory shown in Figure 5, $*((lc3b\_word\_t*)0)$ is BA and $*((lc3b\_word\_t*)1)$ is CB.

4.3 Issues

One must be careful about endian-ness to ensure that one's code is portable to machines that are a different endian than the development machine. For example, if one creates a word by writing individual bytes of that word, one must be careful to write those bytes in the right order for the machine that one is running on.

X86 and VAX are little-endian machines while IBM mainframe and Sparc machines are big-endian. PowerPC has a configuration bit that specifies whether it should support big-endian or little-endian. LC-3b is little-endian and word aligned. **Thus, on an LC-3b, if we want to load a word, the address must be aligned otherwise an illegal operand address exception occurs *(Chiou: corrected from lecture)***.

Chiou: Extra information One common way to make your code unportable between machines of different endian is to create a union type of an integer and a bit structure and create an integer using the bit structure. Bit structures are extraordinarily convenient constructs in C that look like a structure but with a number indicating the number of bits on any or all of the structure variables. However, structure variables are stored in successive positions, starting at the base address of the structure. So, in our example in Figure 6, a is stored in the lowest address, b is stored in the second lowest address (assuming a byte addressible machine), c is stored in the third lowest address and d is stored in the fourth lowest address. Thus, when coercing that bit vector to an integer on an x86 machine, we get the printout 0x44332211 (since we always print out with the most significant byte in the leftmost position.) However, on a Sparc machine (sunfire) we get the printout 0x11223344.

We can fix this problem by using shifts instead of bit vectors. In this example, the `shift` variable prints out 0x11223344 on both an x86 machine as well as a Sparc machine.

4.4 Unaligned Words

Not all machines require address alignment when accessing a word. Given a byte-addressable machine with 16b words that supports unaligned access and a single bank

```
#include <stdio.h>

typedef struct {
  unsigned int a : 8;
  unsigned int b : 8;
  unsigned int c : 8;
  unsigned int d : 8;
} bitvector_t;

main() {
  bitvector_t bv;
  unsigned int shift = 0;

  bv.a = 0x11;
  bv.b = 0x22;
  bv.c = 0x33;
  bv.d = 0x44;

  shift = (0x11 << 24) | (0x22 << 16) | (0x33 << 8) | 0x44;

  printf("   bv: %x\n", *((int *)&bv));
  printf("shift: %x\n", shift);
}
```

Figure 6: An example of how to get in trouble with different endians and how to stay out of trouble

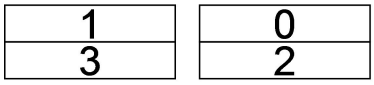

Figure 7: Unaligned Words

of 16b wide memory, providing an odd address to a word load would require loading one byte from one memory address and one byte from the next memory address.

For example, in Figure 7, if the machine is word aligned, the first two aligned words are from (memory 1, memory 0) and (memory 3, memory 2). The first unaligned word, however, is from (memory 2, memory 1).

There are a few ways to deal with alignment:

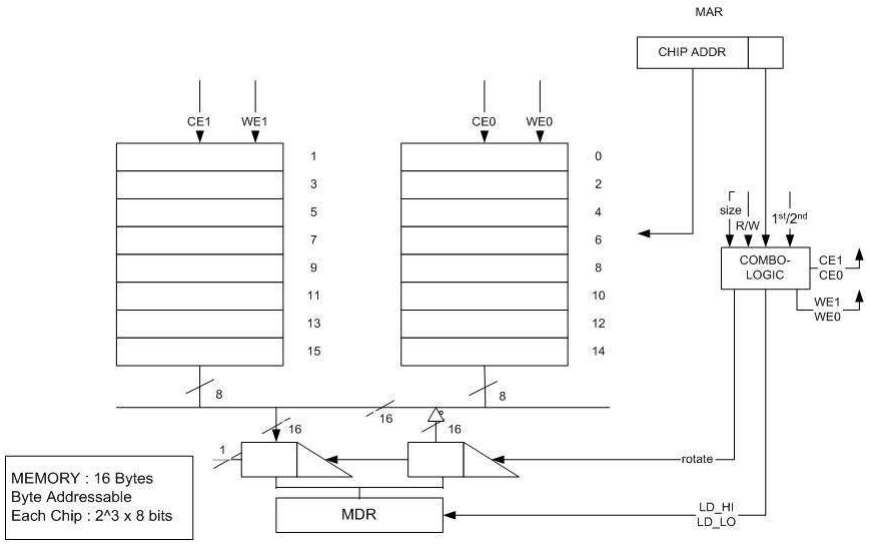

Figure 8: Hardware Alignment

4.4.1 ISA

One way to deal with aligment issues is to explicitly disallow unaligned addresses in the ISA. If unaligned address are not allowed, software must deal with it by masking and shifting parts of the word and ORing them together to assemble them together. Of course, doing so requires more instructions and thus may impact performance.

4.4.2 Hardware

1. $Processor - centric$: Creating a word from an unaligned address is done within the processor. For example, in a byte-addressible, 16b word machine, the processor reads the first word, shifts and masks it to extract one byte and does the same to the second word to extract the other byte and then concatenating them together to form the unaligned word.

2. $Memorysub - systemcentric$: The memory system provides a word from an unaligned address. The processor gives the unaligned word to the memory and gets the correct data. One memory-centric hardware alignment solution is shown in Figure 8.

5 Introduction to Virtual Memory

An analogy that demonstrates why virtual memory is useful is cooking Thanksgiving dinner. The cooks are constantly moving things around to make room for other things because they have only so much room on the counters. To avoid these problems, cooks

want many, infinitely large, infinitely fast countertops, where each countertop corresponds to one recipe. Having such a set of countertops eliminates the need to move things around to make sufficient room for the current recipe. Also, it would eliminate the need to remember where things were moved.

Likewise, programmers want infinitely large, infinitely fast memory for each of their programs. However, resources are always limited, making such a desire impossible. In the old days, computers only had a few hundred words of memory. There is a tradeoff between perceived resources and programmer productivity. If resources are limited, the programmer has to put in more effort to do same task compared to a machine with more resources.

If the computer supports multi-tasking, there would be more than one program running at the same time. There might be viruses or buggy code running along with important code. Clearly, we would want each program to be protected from the bad behavior of other programs. In some sense, each program should be run on its own computer, but that is an expensive and inconvenient solution (how much space would you need for all those computers?). Alternatively, the user could power cycle the machine every time a new program needs to be run, but this wastes energy and time. Virtual memory helps the architect solve these problems at a much lower cost by providing automatic resource sharing and the illusion of a large, contiguous, fast memory space to each process.

Virtual memory is similar to caching in the following three ways.

- Naming: Computers use addresses to name data or, more precisely, to name locations where data can be stored. However, both virtual memory and caches do not always store data with a specific name in the same place. Thus, both need a mechanism for finding data given a name.

- Placement: If there is infinitely large space, everything would have its place. But if it is a smaller space, we need to look at where we would like to put things. If data is small and the memory is small enough, we can place it anywhere. But if the memory is large, we would like to place it in specific locations so that it is easier to access later.

- Replacement: Once you are out of space, we need to determine what we would like to move and where we would like to move it. For example in cache, if we have to write to a cache line, we need to use a replacement technique like LRU to determine which cache line we need to move and how to move the data present in cache line (write through cache or write back cache).