CSC258: Computer Organization

Memory Systems

Schedule

This week …

- First, a few minutes on exceptions to set the stage for the last lab.
- Second, an overview of the memory system.

Next week …

- Wrap-up
- Exam review

Exceptions

Key Questions

- What is the purpose of an exception?
  - What is a system call?

- How are function calls and exceptions similar? How are they different?
  - How is an exception invoked? How do you return?

System Calls

- A system call is just a function call
  - ... that invokes the operating system.

- Printing, getting input, and allocating memory are all system call operations.
- We can't just give user programs control over thse operations because they must be shared across multiple users.

Invoking a System Call

- For security, the operating system must be able to access memory and execute instructions that the user cannot.
  - Otherwise, what would stop the user from just using a resource, rather than asking the OS to do so?

- A system call is invoked by placing arguments in the registers and then causing an *exception*.
  - The exception places the processor in a privileged mode -- one that can act as the OS.

Exceptions in MIPS

- An exception can be caused by a program or by an event.
  - For example, when you divide by zero, that causes an exception. *syscall* also creates an exception.
- The cause of the exception is placed in a special register -- accessed using mfc0.
- Then, an exception handler is invoked.
  - System calls are sent directly to a syscall handler.

Exception Registers

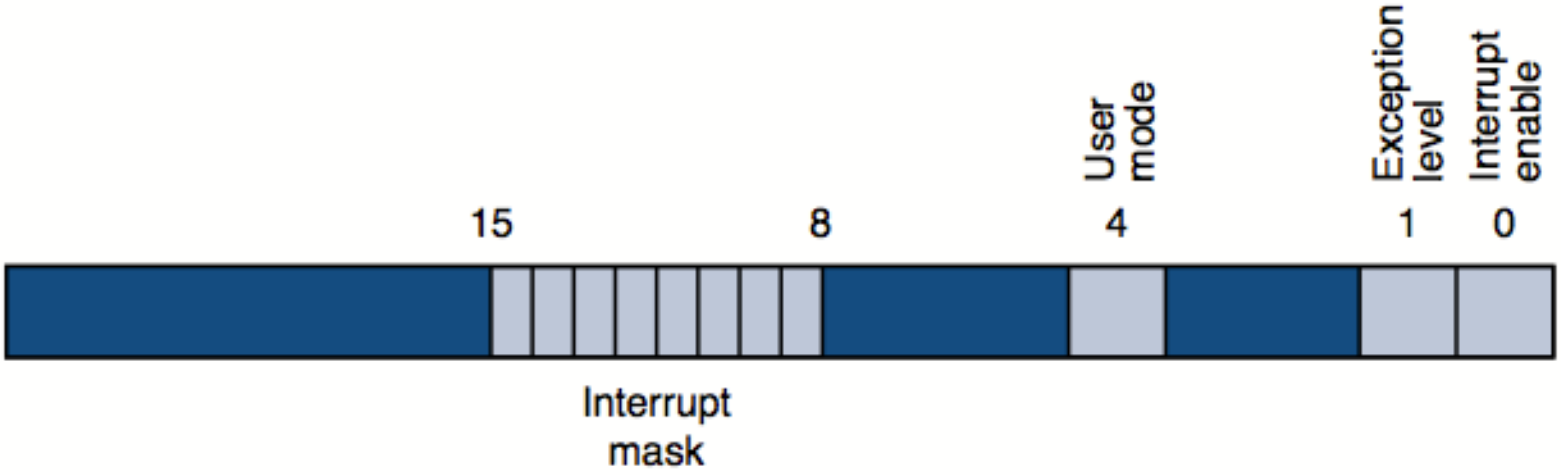

FIGURE A.7.1 The Status register.

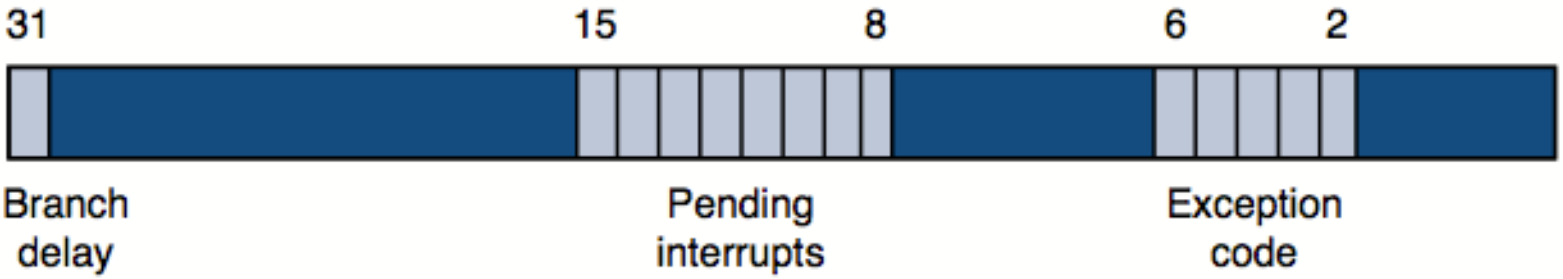

FIGURE A.7.2 The Cause register.

Source: Larus, *Assemblers, Linkers, and the SPIM Simulator*. http://pages.cs.wisc.edu/~larus/HP_AppA.pdf

Things to Remember

- The exception handler is just assembly code.
  - It's like any other function!
  - **... but it must not cause an error.**

- It should leave the machine just as it was -- except for fixing whatever condition caused the exception.
- It can't change the stack. It shouldn't junk any registers.
  - The exception handler you'll use in the last lab saves two registers in the data segment. Those are the only registers you may use.

The Memory System

Quiz-like Questions

Last week, we discussed *spatial and temporal parallelism*. This week, it's *spatial and temporal locality*. They aren't particularly related, so be careful!

1. Generate examples of both types of locality outside of a computing context.
2. Define *spatial locality* and *temporal locality*.

Key Computer Systems

- So far, we've focused on the processor. That's where execution happens.

- There are two other key systems: *memory* and *I/O*.
  - *Memory* controls the storage of data.
  - *I/O* controls the flow of information into and out of the system.

Connections to OS

- We'll focus on the memory system and not talk much about I/O this term.

- At a high level, the memory system includes RAM, the hard drive, and the communication infrastructure that connects those devices to the processor.
- We won't discuss the hard drive much.
  - That's the realm of virtual memory, and that's where this material connects with CSC369 (OS).

Why is the Memory System Important?

- Most processor spend most of their time waiting.
  - ... often for memory. This is the "*memory wall*".
  - As processors get faster, more processor cycles can be executed before a *load* completes.

- As a result, *Amdahl's Law* tells us this is an aspect of performance that is becoming increasingly important.
- Caches are one way to reduce waiting time.

Quiz-like Questions

1. State *Amdahl's law*.

2. *For discussion:* What does Amdahl's law tell us about optimizing performance of software?

A Memory System

- A whole memory system includes main memory (RAM) and a series of caches.
  - The L1 (level-one) is very close to the processor, and it is quite small.
  - The L2 is a bit further away and a bit bigger.
  - The L3, if there is one, is often off-chip and even larger.
- The further a cache is from the processor, the more it stores.
- The size of the cache line shrinks as you get closer to the processor.

Caching

- *Caches* are a solution that makes it appear that memory is closer than it is.
- Every load to memory fetches more than just the value that is loaded.
  - In fact, a lot of values -- a *block (or line)* -- is brought from the memory to a location close to the processor.
  - That location is called a *cache*. It stores the value that was loaded and the values near it, in case they are needed soon.

Locality

- *Locality* is one of the big ideas in computer science.
- In essence, locality is the assumption that history will repeat itself. That:
  - *spatial locality:* If one object in memory is accessed, objects close to it will also be accessed.
  - *temporal locality:* If one object is accessed, it (and objects around it) will be accessed again soon.
- Locality is required for caches to function well.

Examples

- "Iterating over an array" exhibits both temporal and spatial locality.
- "Executing code" often exhibits temporal and spatial locality.
- "Accessing items from a dictionary" does not: the items in the dictionary may not be close to each other in memory.
- Linked lists and other dynamically allocated structures can also cause locality problems.

Caches

Key Cache Terms

- Address
- Tag
- Block
- Set
- Associativity
- Hit rate (and miss rate)
- Average Memory Access Time (AMAT)

Addresses and Caches

- Each *load* fetches an entire *cache block* -- not just a single value.
  - The size of a cache "block" is dependent on the cache.
  - A "block" is a set of words with closely related addresses.

- The easiest way to define a block is to look at its *mask*.

Bit Masking

- A *bit vector* is an integer that should be interpreted as a sequence of bits.
  - We can think of an address as a bit vector.
- A *mask* is a value that can be used to turn specific bits in a bit vector on or off.

- For example, let's set a mod-16 mask.

```
value = ....
mod_16 = 15                  # 0x000000F
print (value & mod_16)       # Only the bottom 4 bits
                             # "&" is "bitwise and"
```

Masking Addresses

- We can create masks to define different parts of an address.

- Suppose we have 32-bit (4-byte) words.
- Suppose an array starts at *base*. Then the offset is:
  address = ...
  base = ...
  offset = address & base
- Similarly, suppose a cache block contains 8 words. Then every block contains the least significant 32-bytes (5 bits)
  cacheblockoffset = 31 # 0x0000001F
  cacheblock = maxint - 31 # 0xFFFFFFE0

Quiz-like Question

Given a 32-byte address space, identify the tag, set, and block offset for the following cache configurations:

1. A direct mapped cache that stores 16 32-byte blocks.
2. A 2-way set associative cache that stores 32 16-byte blocks.
3. A fully associative cache that stores 8 16-byte blocks.

Discussion

What is the effect of increasing:

(a) block size
(b) associativity
(c) cache size

Discussion

What is the effect of increasing:

(a) block size
(b) associativity
(c) cache size

What is the impact of having a two-processor system with:

(a) a shared cache
(b) separate caches but shared memory

Cache Loading and Evicting

- Back to caches: every load brings in a block.
- Each cache has a finite size.
  - It can store some maximum number of blocks.
  - Based on associativity, it can store a set number of blocks with a specific hash.

- Every time a load is performed from memory, the block must be stored.
  - This means that another block might need to be *evicted*.

Associativity

- Most caches use some form of hashing.
  - The caches are smaller than the memory they are caching from, so they can't store everything!
- If two blocks hash to the same value, they can't both be stored. To avoid that, caches are often *associative*.
  - A 2-way set associative cache can store two blocks that hash to the same value.
  - A fully associative cache doesn't have to worry about hash collisions at all.

Eviction Heuristic

- How do we choose which cache block to remove (evict)?

- The most common heuristic is *"least recently used" (LRU)*.
  - The cache block that was accessed the longest time ago is dropped.
- Other heuristics include "*first in first out" (FIFO)*, *"least frequently used"*, and others.

Quiz-like Question

Simulate the performance of a cache on the following (hex) address *loads*.

40 48 4c 40 50 58 5c 40 60 48 4c 44 40 60 58 5c

The cache is direct-mapped and stores 4 words. It uses a FIFO eviction policy.

Quiz-like Question

Simulate the performance of a cache on the following (hex) address *loads*.

40 48 4c 40 50 58 5c 40 60 48 4c 44 40 60 58 5c

The cache is direct-mapped and stores 2 blocks of two words. It uses a FIFO eviction policy.

Quiz-like Question

Simulate the performance of a cache on the following (hex) address *loads*.

40 48 4c 40 50 58 5c 40 60 48 4c 44 40 60 58 5c

The cache is 2-way set associative and stores 4 words. It uses a FIFO eviction policy.

Quiz-like Question

Simulate the performance of a cache on the following (hex) address *loads*.

40 48 4c 40 50 58 5c 40 60 48 4c 44 40 60 58 5c

The cache is 2-way set associative and stores 4 words. It uses an LRU eviction policy.

Quiz-like Question

Simulate the performance of a cache on the following (hex) address *loads*.

40 48 4c 40 50 58 5c 40 60 48 4c 44 40 60 58 5c

The cache is fully associative and stores 4 words. It uses a FIFO eviction policy.

Cache Consistency

- Consistency is a huge problem -- and we won't discuss it.
- Imagine a multi-processor system: each processor has its own L1.
  - What happens when one of the processors issues a store? Is it written back to memory? To the other processor's L1?
  - How quickly does that write-back happen? What will occur if the second processor accesses that memory address before the write-back?