CMPE 110 in a Nutshell: The Fundamentals Needed for CMPS 111

Presented by Alexander McCaleb.
Based on notes from Alexander McCaleb, Joel Ferguson, Jared Mednick, Andrea Di Blas, and Jose Renau

What I assume you know

1. CMPE 12 material. Have familiarity with,
    a. Logic Design
        i. At the gate-level
        ii. Can implement basic higher-level devices
            1. Muxes
            2. Decoders
            3. RC Adder
            4. ALU
    b. Assembly Language
        i. Any ISA will due.
            1. I'll mainly be referencing MIPS.
    c. Basic Computer Architecture
        i. Can trace through a datapath
        ii. Stages of instruction processing
2. CMPS 101

Topics we'll discuss (in no order)

- MIPS: Evaluating Performance
- Caches
- Virtual Memory
- Memory Hierarchy
- Processors and Peripherals
- Threading
- Pipelining
- The General Machine Model
- How the different components hook together

Bonuses Included

- Being really gitty about composing all these notes, I will indicate when the notes stray from the absolutely needed for CMPS 111 and are more for the reader's interest.
  - For this, material may be from outside research or covered in other classes that I wouldn't assume everyone has taken.
    - What other classes will be indicated

- Some material from other classes may be covered include:
  - CMPE 202: Graduate Computer Architecture
  - CMPE 100: Logic Design
  - CMPS 146: Game AI
  - CMPS 20: Game Design Experience
  - CMPE 13: Computer Systems & C Programming
  - CMPE 156: Network Programming
  - EE 101: Introduction to Electronic Circuits

MIPS: Evaluating Performance

Defining MIPS

- MIPS stands as 2 things actually:
  - 1. An Instruction Set Architecture (ISA)
    - Further examples will reference this ISA
  - 2. An acronym for "Millions of Instructions Per Second"
- Let's begin with item 2
  - We'll need this to explore item 1 further.

MIPS (Millions of Instructions Per Second)

- Point:
  - We need some way to evaluate performance that accounts for various types of instructions we may execute
    - Ex:
      - add r1, r0, r2 //Can take 1 cycle to execute
      - fdiv f1, f0, f2 //Can take 100 cycles to execute
- Context:
  - To compute MIPS, we need to know something about how many cycles each instruction takes to execute
  - We call this...

CPI (Cycles Per Instruction)

- Point:
  - Given any processor, we need to know how long it takes to complete instructions.
    - This means that the lower our CPI is, the better.
  - Why cycles? Why not actual time?
    - We measure cycles instead of time because we can always change our processor's clock speed.
      - Cycles can only change is we change the datapath of our hardware.
- How to:
  - Many, many, many ways to compute CPI. The equation to use depends on the given data.
  - We'll only be concerned with
    - CPI = CPU time(Clock Rate)/Instruction Count
      - CPU time(Clock Rate) = # of Cycles

MIPS & CPI

- We have two types of MIPS to consider for a given instruction mix.
  - 1. Native MIPS:
    - Based on average CPI
  - 2. Peak MIPS:
    - Based on instruction type that nets the lowest CPI for a given instruction mix

Native MIPS

- Before we can do this, we need average CPI
  - Average $CPI = \sum_{i=1}^{n} \frac{IC\_i}{IC} * CPI\_$
    - Where,
      - We have n instruction types
      - IC_i = instruction count for type i
      - CPI_i = CPI for type i
    - Note:
      - IC_i/IC gives us the frequency of instruction type i in our mix.

- Now, we compute Native MIPS as,
  - MIPS_Native = f_clk/(CPI_avg * 10^6)
    - Where,
      - f_clk is clock frequency
      - We multiply by 10^6 to get million portion.

Peak MIPS

- Why should we do this?
  - Different programs have different CPIs
  - Different ISAs have more powerful, but similar instructions
  - Tells us the maximum MIPS for a given instruction mix
- How to do this?
  - First, find the lowest CPI_i in our instruction mix (by math or by observation)
  - Now we compute Peak MIPS as
    - MIPS_Peak = f_clk/(CPI_lowest * 10^6)

Variations on MIPS

- Looking at CPI_i from Native MIPS, we see that CPI_i's can vary by instruction type.
  - We can give these variations their own classifications.
- Some of these classifications include:
  - MOPS: Millions of ALU Operations Per Second
    - MOPS = f_clk/(CPI_ALU * 10^6)
  - MFLOPS: Millions of FLoating point Operations Per Second
    - MFLOPS = f_clk/(CPI_FP * 10^6)

Problems with MIPS

- Unless we look at specific types of instructions, our MIPS calculation can be skewed by a bad instruction mix.
- For this reason, MIPS is almost never reported by companies making processors.

Exercises

- CMPE 110, Spring 2011
  - HW #1, 1a
  - Quiz #1, 1-4

- Compute Native and Peak MIPS for the following instruction mixes.

Mix 1	FP	ALU
Freq.	95%	5%
CPI	12	3

Mix 2	FP	ALU
Freq.	5%	95%
CPI	12	3

- Can you derive any general characteristics of MIPS from this data?

Caches

Memory: A time consuming process

- When we're communicating with memory, this is a very expensive process.
  - Typical DRAM access is around 200 cycles.
- Logically, this can't be good for performance,
  - Our CPI skyrockets and we can't have our processor be held up on memory all day.

Speeding up memory access

- Our memory access problem is mainly due to distance b/w processor's core and main memory.
  - Electrons take time to travel.
- We reduce this by putting a subset of main memory closer to the processor's core.
  - This memory is a cache.

Why caches work: Locality

- Two fundamental principles drive the functionality of caches.
  - Spatial Locality:
    - You're more likely to access memory nearby memory you previously accessed.
  - Temporal Locality:
    - You're more likely to access the same address multiple times sequentially than something completely new.
- Ex: (Insert any sorting algorithm here)

How caches work: The Big Picture

- Caches are significantly smaller than main memory.
  - If we could make then as big as main memory, why not just have main memory inside processor's core?
    - Multi-core processor model breaks in this case.
- Caches are indexed by a subset of the bits used to address memory.
- To capitalize on spatial locality, every index of the cache can hold multiple memory addresses.
  - This means we need a way to get to each respective address.
    - To do this, use another subset of bits to address main memory.
- When we have some data in the cache, how do we know where it came from?
  - Use remaining subset of memory address bits to figure this out.

Polling Memory: The Cache Lookup

- When we need to load or store some data, we first look for the needed address in our cache.
- To do this, we decouple the bits of desired address as follows, (for an n-bit memory)

Tag [n:x]	Index [x-1:y]	Byte Offset [y-1:0]

  - Byte Offset:
    - Bits to designate the specific byte we want from the cache.
  - Index:
    - Bits to designate what address of the cache our data will be.
  - Tag:
    - Remaining bits used to properly identify the addresses we have in the particular address of the cache.
- This will get more complicated later on.

The cache as a hardware unit

- The structure of all caches vary on this model
  - Values in this table are arbitrary

Index	Valid?	Tag	Data Bank
0x00	1	0x00	Contents of Addresses: 0x00000 - F
0x01	0	0x20	Contents of Addresses: 0x20010 - F
...	...	...	...
0xFE	1	0x86	Contents of Addresses: 0x86FE0 - F
0xFF	0	0xCD	Contents of Addresses: 0xCDFF0 - F

Cache Decoupling: The Cache Line

- On the model we saw before, what do all those field mean?
  - Index:
    - The particular cache line we're looking at.
      - Comes from the cache look-up.
  - Valid?:
    - A bit to assert whether the data in this cache line is legitimate.
      - Cache contents can be invalid when the computer has just booted, the most recent version of the data is in another cache, etc.
  - Tag:
    - From the cache look-up so we know the entirety of the memory address we have in the cache.
  - Data Bank:
    - The contents of the addresses that fit in our cache line.

Cache Terminology

- Hit:
  - We found what we needed at a particular cache line.
    - The cache line must be valid and the tags must match between the look-up and the tag of this line for this to be true.
- Miss:
  - For the cache line we're looking at, we didn't find the data needed.
    - We now look at the next level of the memory hierarchy for our data.

Types of Cache Misses

- We can miss in a cache for many reasons.
- To get a clear idea of why we missed, we classify our misses as such.
  - Compulsory:
    - This occurs the first time a cache line is accessed as we can't have valid data there.
      - Commonly occurs when system first boots.
  - Conflict:
    - The data we need can't exist in the needed cache line because other valid data is in this cache line.
      - Commonly occurs with Direct-Mapped caches.
  - Capacity:
    - The data we need can't fit in a given cache line because the cache line is filled with valid data already.
      - Commonly occurs with Set Associative caches when we run out of ways.

Variations on Caches

- From our description of cache misses, I've introduced some varying properties of caches.
- We'll explore onl 3 of them.
- For multi-core processors, we'll get many more, but that's a discussion for later.
- Variations we'll cover:
  - Associativity
  - Writing policies
  - Memory Alignment

Variations on Caches: Associativity

Associativity of caches:

- Defining which locations in the cache a particular line can go.

Common types:

- Direct-Mapped (1-way Set Associative)
- n-way Set Associative (SA)
  - n is a power of 2
- Fully Associative

Cache Associativity: Direct-Mapped

- For every memory address, there is only one cache line it can go to.

Pros	Cons
• Very little power consumption • Make index calculation easy	• Introduces many conflict misses • Kills performance unless programmed right

- Requires no modification to model already described to implement.

Cache Associativity: n-way Set Associative (SA)

- For every memory address, it can go within n-slots of the cache index it's mapped to.

Pros	Cons
• Significantly reduces conflict misses • Gives a huge performance boost for just a little more power consumption • Usually the best policy	• More hardware such as ways to handle where data goes • Needs some protocol to handle replacement of data in ways • Introduces capacity misses

- Supporting this:
  - Our cache index calculation uses less bits.
    - Take # of bits for DM cache and divide by Set Associativity.
  - Introduce the sections in the cache index our data can go.
    - These are known as the "ways" of the cache.
  - Each cache line needs a new set of bits to handle replacement policy.
  - For modern processors, need hardware to predict which way will be accessed next.

Cache Associativity: Fully Associative

- For every memory address, it can go anywhere in the cache.

Pros	Cons
• Conflict misses are completely eliminated • Can go for a long time with 100% hit rate • Cache index calculation is eliminated • No replacement policy	• Horribly high power consumption • Once cache is filled with valid data, every access to new data will inflict a capacity miss • Lots of additional hardware to find open slot in cache • Runs extremely slow due to added delay from slot checking hardware

- Supporting this:
  - Tag becomes huge as index is eliminated.
  - Need to implement some support to find the first invalid bit
    - Probably some search algorithm that the OS would most likely handle.
    - Could also do this in hardware, but that'll have its own problems as we know.

Variations on Caches: Writing Policies

- The writing polocy on the cache dictates when we give the lower levels of the memory hierarchy the modified data in the cache, i.e. a ST instruction was executed.

- Types:
  - Write-through
  - Write-back

Cache Writing Policy: Write-through

- As soon as some cache line is modified with new data, propagate these changes down the hierarchy.

Pros	Cons
• We can propagate our writing downward while our processor handles other tasks (less ST latency) • Keeps hierarchy consistant	• Increased power consumption when we propagate these writes down • May make unnecessary changes to memory (program-dependent)

- No changes to our current model needed to support this.

Cache Writing Policy: Write-back

- Only propagate our changes down the hierarchy once our cache line is about to be replaced.

Pros	Cons
• Only make changes when we have to (less ST latency) • Less power consumption	• More hardware, including Write-back buffer and modified/dirty bit • Memory inconsistencies are more common.

- Supporting this:
  - Our cache line needs to add a bit to indicate if it has been modified by a ST.
    - This is the modified/dirty bit
  - Need to add a buffer to allow our data to be written back when needed.
    - Ideally, this would be configured to write back to each unit lower in the hierarchy in parallel. Wasting time otherwise.

Variations on Caches: Memory Alignment

- Word:
  - Amount of bytes that define a standard amount of memory we access.
    - Usually 4 or 8 bytes (32 or 64 bits), i.e. size of int type.
- Memory Alignment:
  - Accessing memory addresses so that we don't fall in-between words.
    - Can do horrid things by modifying data in-between words.

Memory Alignment: Implementation

- Given any invocation to a memory address, how do we ensure that the first address accessed is a multiple of our word size (in bytes)?
- This is also known as pointer-tagging.
- Solution (assuming 32-bit addressability and 256 byte words):
  - Intuitive solution:

```
value = ((value & 255) == 0) ? (value) : ((value + 256) & -255);
```

  - Solution without conditionals:

```
value = (~(value + ((((value & 255) - 1) & (-2^31) >> 23) & 256;
```

- What does this mean for caches?

Variations on Caches: Memory Alignment Unsupported

- Only ISA I know of that has this is x86
- Allows for loads with unaligned offsets to be executed, i.e.
  - lw 3($3) will work assuming 32-bit words

- Effects on caches:
  - No matter what byte we want to access, the next x-bytes that fit in our cache line are put in the cache.
    - Negatively affects spatial locality as it doesn't allow previous addresses in memory to be in the cache.

Variations on Caches: Memory Alignment Supported

- Vast majority of all ISAs support this
- Now, our load example from before will throw an exception.

- Effects on caches:
  - When trying to access a particular byte from memory, we take the entire word it's associated with with us.
    - To do this:
      - Look for address desired with byte offset hardwired to all 0's
      - Grab every subsequent address from there until cache line is filled.

Exercises

- CMPE 110 Spring 2011, Quiz 6
  - There is some additional concepts needed left to the reader to look up in old lecture slides
- CMPE 110 Spring 2011, HW 6, #1 and #4
- Simplify the memory alignment code such that it uses less operations and still involves only bitwise operations.

Virtual Memory

Reading

- Chapter 3 of *Modern Operating Systems* describes everything needed for interacting with Virtual Memory at the OS level.
- These slides will give a much more detailed background of Chapters 3.1-3.2 of the textbook.
- That said, read those sections of the book first!

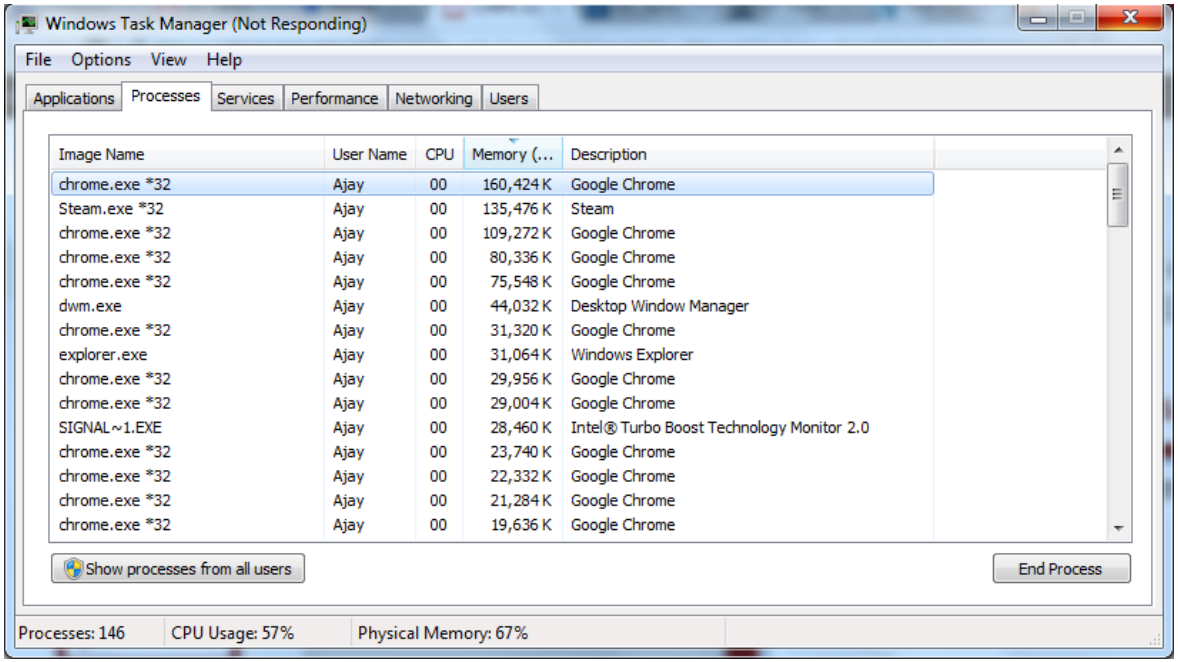
Still More Memory Problems

- Thus far, we've only considered cases of microprocessors.
  - i.e. the processor is running only one program.

- This begs a question
  - How do the processors in our personal computers work?

Booting Your Personal Computer

- Bringing up the processes tab in our task manager reveals...



  - We're running multiple programs at the same time
- As we know, once we actually start using the computer, that's even more parallel tasks we're handling.

How We Run Multiple Tasks

- We could easily argue that each process we run must exist in its own software thread, but that still leaves the question of how does the hardware behave?

- After all, we can't arbitrarily spawn hardware

How We Store Multiple Tasks

- Our main problem is that we have an arbitrary amount of programs running, but they all need to share the memory built-into our computers.

- How can we resolve this?
  - Require every program running to know everything about the memory usage of every other program?
    - This would make some form of a dependency graph
      - Unless this graph was a DAG by some miracle, we have circular dependencies, so this approach doesn't work.
    - Furthermore, if this could actually work, would it even be a good idea?
      - Think about good OO-design
        - We make things more abstract for clarity and modularity
          - How can we apply this to hardware?

OO-Design of Hardware: Virtual Memory

- In software, let each program have its own notion of memory and have hardware handle how this programs sits in physical memory.
- To do this, we give a dedicated virtual address space to each program.
  - Typically >= the physical address space
    - If we made it < physical address space, can only use a subset of memory.
- Modern processors do a similar process with registers, but that's a discussion for later.
- So this sounds like a good idea, but how do we go from virtual to physical?

Paging: Going From Virtual To Physical

- Page:
  - A block of memory.
    - For our purposes, these blocks contain mappings of virtual addresses to a physical ones.
- Paging:
  - The process of creating pages of a distinct size.
    - That size being large enough to store a mapping as well as some other information needed by externals that operate on pages.
- Page Table:
  - A table where pages are accessed and stored.
    - The size of the table is determined by the Virtual Page Number (VPN) size.
      - Our VPN size is smaller than our physical memory size, thus our page table can't hold all possible pages.

- This leaves the question, where are all of my pages?

Finding All The Pages

- Unless you have some ridiculously configured OS, whenever you install a program, all of its contents exist on the hard disk.
- When we run that program, it goes to instruction memory from the hard disk.
- The page table itself obviously cannot contain any of the pages from this newly started program.
  - That said, the first time this program needs to load something from memory, a page fault occurs.

Page Faults: Definition

- Page Fault:
    - When an instruction requiring data memory interaction was started, we couldn't locate the page in physical memory that corresponds to the address in data memory we need to access.
        - Basically the same idea as a cache miss

Boils down to:

- "A page in physical memory must be swapped for content on disk"

Page Faults: Resolution

- This is done as an interrupt service routine (ISR) which behaves as follows:

1. CPU saves state of process and transfers control to OS.

2. OS looks up the page table on disk to find the missing page.

3. OS must choose page to swap out of memory to make room for new page.
    a. The rest of Chapter 3 focuses on algorithms to do this.

Page Faults: Resolution (cont.)

4. If page is dirty, must write back page we're swapping.
   a. Like we saw with write-back caches

5. Read new page on disk and put in physical memory.

6. Update the page table in memory and the TLB.
   a. We'll get to TLBs momentarily

7. Restart the instruction that caused the page fault.

8. OS hands control back to CPU.

Paging: Still Have Some Unresolved Issues

- In solving the problem of multiple program interacting with memory, we've reintroduced a previous problem.
  - Every time we need to get a page, we have to go to memory which is a slow and horrid process.
- We can solve this problem in a similar way as last time.

Translation Lookaside Buffer (TLB): A Cache of Pages

- This is exactly what is sounds like.
-
- Only differences between a TLB and a cache:
  - Data held in TLB is just one virtual to physical mapping versus the contents of multiple addresses that a cache can have.
  - TLB lines can have unique control bits that make no sense for caches.
    - Ex: Process ID (PID):
      - Some TLBs have a limited number of processes they can handle concurrently and it need to know which process a page belongs to.
  - A miss in the TLB forces us to look at memory page

Putting Virtual Memory Together: Virtual Address (VA)

Virtual Page Number (VPN)	Page Offset (PO)

- PO = lg(Page Size)
  - Similar to the byte offset of our caches
- VPN = lg(VA Space) - PO
  - Similar to the tag bank of our caches, but is very different.
    - Ex:
      - Page Table indexed by VPN
- Translates to TLB lookup

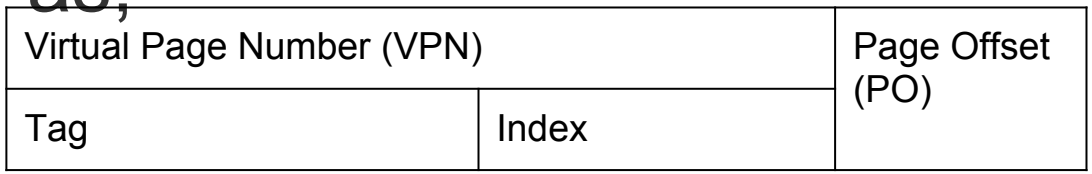
Putting Virtual Memory Together: Physical Address (PA)

Physical Page Number (PPN)	Page Offset (PO)

- PO
  - Comes from VA
- PPN = lg (PA Space) - PO
- Translates to cache lookup

Putting Virtual Memory Together: TLB Lookup

- Since this comes from the VA, we can compare them as,
  - VA:
  - TLB Lookup:



- Computing new fields:
  - Index = lg(#sets)
    - #sets(TLB) = lg((#entries * #lines)/Set Associativity)
  - Tag = VPN - Index

Putting Virtual Memory Together: TLB Line

Valid	Bits for Writing Policy	Bits for Replacement Policy	Bits for Process Support	Tag	Physical Page Number (PPN)

- Valid and Bits for writing and replacement
  - Just as with cache lines
- Bits for process support
  - Only needed when TLB can support finite processes
- Tag
  - Comes from our TLB lookup
- PPN
  - This is the data in the TLB and a page table entry
    - Page table entry probably also needs some data related to page table's replacement policy

Implementing Virtual Memory

- Having knowledge of how virtual memory works, how do we build it?
- Some universals:
  - The memory that the page table uses as well as all the TLBs in our processor live in the Memory Management Unit (MMU).
  - All Virtual Memory implementations involve some interaction with the OS to process interrupts.

Implementing Virtual Memory: Early MMU Designs

From: "Virtual Memory In Contemporary Processors"

- A hardware state machine performed page table search on a TLB miss.
  - The state machine walked the page table
  - Then the mapping was loaded
  - TLB was then refilled
  - Lastly, our computation was restarted

- TLBs themselves
  - Fully-associative, which holds same problems as fully-associative caches.
  - "To provide increased translation bandwidth, designers often split TLB designs".

Implementing Virtual Memory: Early MMU Designs (cont.)

From: "Virtual Memory In Contemporary Processors"

- OS interaction
  - Some would freeze pipeline when TLB miss occured to service the miss
    - So everything would have to wait for the TLB miss to resolve
  - Others would allow the potential exception from TLB miss to propagate through
    - If the exception isn't handled by the time the instruction that triggered the TLB miss gets to the end of the pipeline, need to dump everything and restart from offending instruction.
      - Is this really economical?
    - Ex: Intel Pentium Pro

Implementing Virtual Memory: Software-Managed

From: "Virtual Memory In Contemporary Processors"

- OS:
  - Handles page faults as previously mentioned => handles page table organization
    - ISR to assess page faults and TLB misses that works like any other code.
      - I.E. it's run from I$

"If the handler code is not in the instruction cache at the time of the TLB miss [or page fault], the time to handle the [problem] can be much longer than in the hardware-walked

Implementing Virtual Memory: Software-Managed

From: "Virtual Memory In Contemporary Processors"

- OS:
  - Designed to hide hardware particulars from the user.
    - This typically prevents programmers from fully optimizing the hardware.
      - Unfortunately, this is a general trend between software and hardware
        - Ex: Xbox 360
          - Arithmetic units designed for ultimate floating-point performance
          - .NET Compact Framework severely limits the floating-point operations programmers can execute.
            - Luckily, only games on the Indie Marketplace use this framework.
  - Contains its own TLB instructions
    - Ex: TLBWR, TLBWI
      - These give flexibility to the replacement policies the OS can define.

Implementing Virtual Memory: Hardware-Managed

From: "Virtual Memory In Contemporary Processors"

- Mainly follow trends we saw with early MMU designs.
- Typically will use an inverted page table.
- TLB miss mechanism is completely in hardware.
- INSERT NOTES FROM EXERCISE HERE

Implementing Virtual Memory: Location in Datapath

From: "Virtual Memory In Contemporary Processors"

- What's intuitive
  - Translate from virtual to physical memory ASAP and operate mainly on physical memory
- What can be done
  - We could reserve memory translation for just when we go to memory
    - Less power consumption from saved operations
    - More hardware required for virtual cache implementation
  - Could put memory translation anywhere between caches and memory
    - Will change values of aforementioned tradeoffs
- What's done today
  - Do memory translation as a component of the caches
    - This gives Virtually-Indexed, Physically-Tagged caches.

Exercises

- How do you suppose the OS handles the multiple processes it runs? Give some logical algorithms.
- In LC-3, write a trap routine to handle a page fault.
- CMPE 110, Spring 2011, Quiz #7
- For those familiar with State Machine Design
  - Construct a state diagram to snoop our page table on a TLB miss.
  - Implement the state diagram in either hardware or software.
- Research more into hardware-managed Virtual Memory and add findings to these notes.

Memory Hierarchy

Pipelining

Before We Begin

- Review instruction processing in LC-3
  - Whatever chapter in the CMPE 12 textbook

What We'll Cover (in brief)

- Single Cycle Processors
- Multi-Cycle Processors
- Pipelined Processors
- Scoreboard Processors
- Tomasulo's Algorithm
- Modern Out-of-Order (OoO) Execution
- Superscalars

Considerations When Evaluating Processor Performance

- Latency:
  - How long it takes for one operation to complete.
    - Ex: Execution Time
- Throughput:
  - Rate at which we're finishing operations.
    - Ex: CPI
- When evaluating processor performance, our primary concern is throughput
  - This is because we can easily manipulate latency by changing our clock speed.
  - Modifying throughput requires adding/deleting hardware.
- We'll be analyzing multiple techniques for improving throughput.

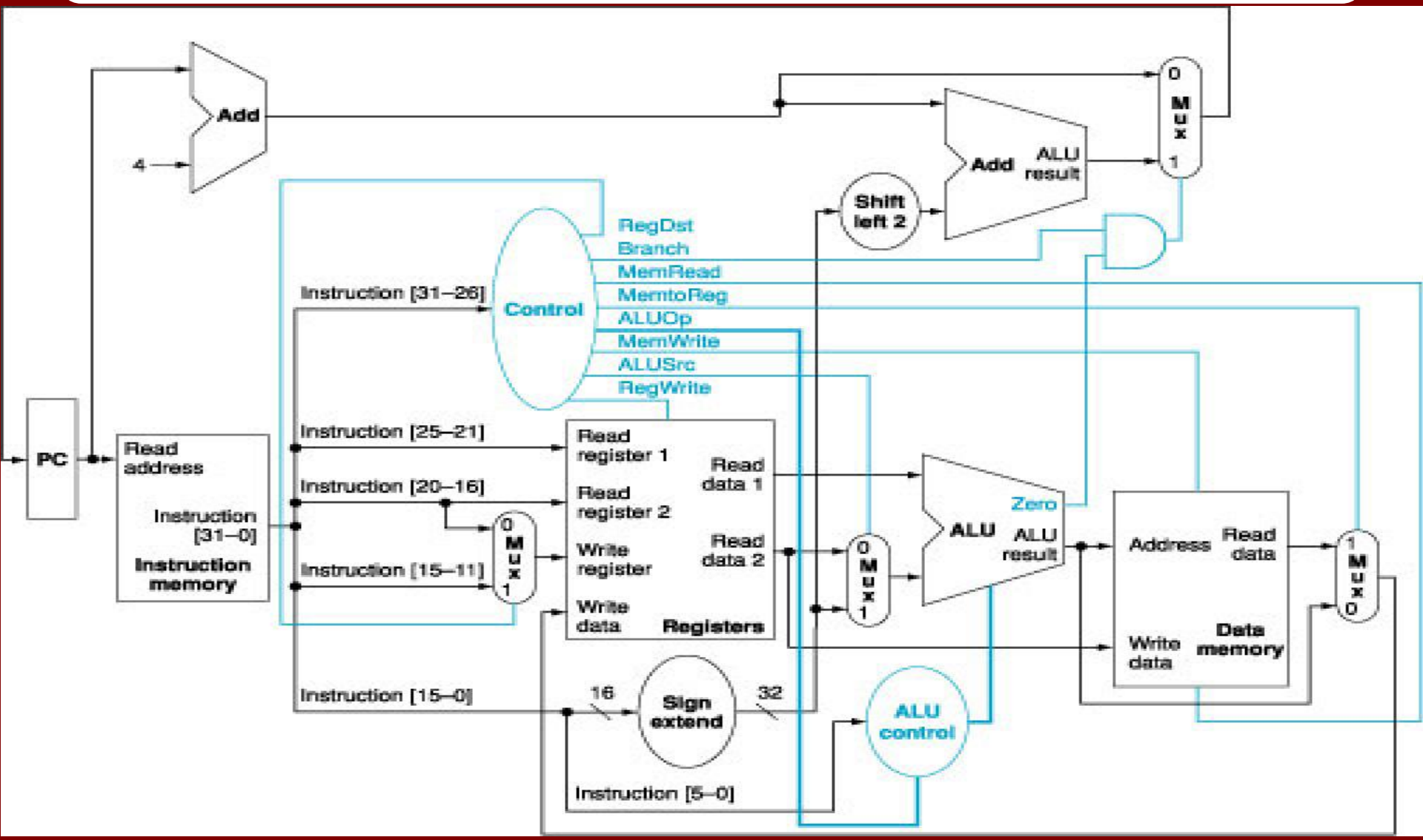
Processing Our Insturctions

Now that you've reviewed the stages that the LC-3 uses, you've got a general idea of what needs to happen to each instruction.

- So the issue now is how do we implement this process?
- In this explanation, we'll begin with adding components to the MIPS datapath.
- Towards the end, we won't even have one complete datapath to reference.
- KEEP THESE DATAPATHS HANDY!

Easiest Solution: Single Cycle

Image from: http://inst.eecs.berkeley.edu/~cs61c/su10/assignments/hw8/datapath.jpg



Single Cycle Processor

- After every clock cycle, an instruction is complete.
  - So our CPI = 1
    - Cool! How could we possibly beat this? Well...

Problems with Single Cycle Processor

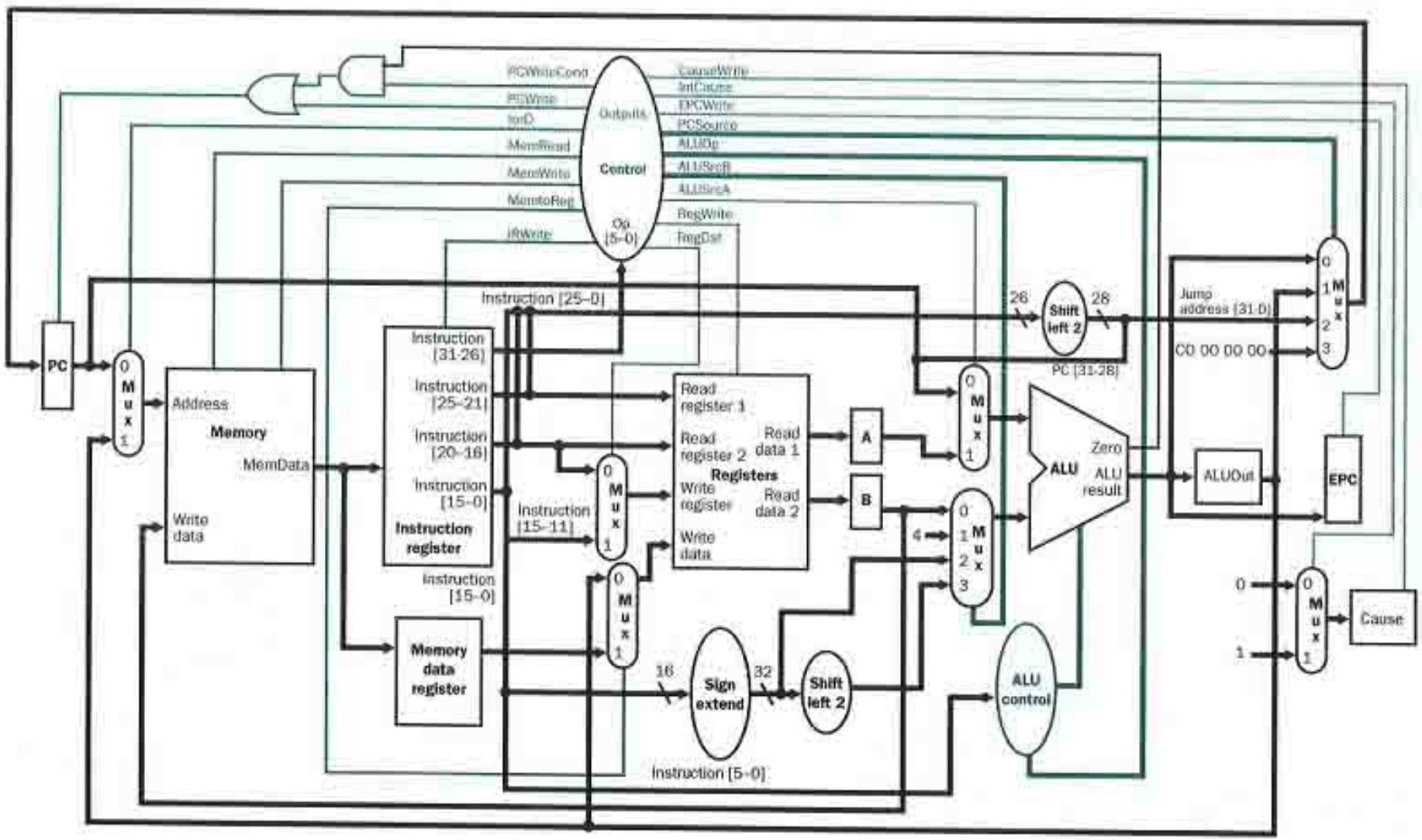
- While our throughput is excellent, the latency here is atrocious.
- As our processor runs in just one cycle and with no sequential logic elements (except for the PC), our processor is almost purely combinatorial.
  - This means that noise or outside interference can affect values easier.

Problems with Single Cycle Processor (cont.)

- In order to ensure our processor can operate properly, we must ensure that we can complete every possible type of instruction before the next clock edge.
  - In other words, the clock period must be >= the time for a signal to propagate through the critical path of our processor
    - Usually executing a DIV will trigger the critical path.
  - This implies that all instructions will take the same amount of time to complete.
    - So a NOT will be as long as a DIV where a DIV can take milliseconds to complete
      - THIS IS ABSOLUTELY UNACCEPTABLE!

Multi-Cycle Processors

Image From: http://milk.sjtu.edu.cn/HanchengWANG/img/multi.jpg



Multi-Cycle Processor

- Now, instead of completing all stages of execution in one shot, we dedicate one cycle to each stage.
- This allows instructions to be of differing lengths.
  - Each instruction can have varying stages which take differing times to complete.
- Our clock period can be much faster as we only need to consider the critical path in the longest stage to complete.
- Introduces sequential (synchronous) logic elements, increasing the stability of the processor.
- This all gives significant improvement on latency, but...

Problems with Multi-Cycle Processors

- Our throughput chokes:
  - CPI = # of stages to complete
    - 3ish for a JMP
    - 40ish for a DIV
  - Thermal issues also choke our performance as each section of hardware is only used "1/# of stages to complete"th of the time.
    - Think of trying to warm something in microwave in 10 second intervals vs a straight 2 minutes.
  - Using each section only "1/# of stages to complete" th of the time is not economical.

Implementing this is actually more complicated than the solution to the multi-cycle processor.

Pipelined Processors

Image From: http://cseweb.ucsd.edu/classes/wi08/cse141/Media/mystery-pipeline.png

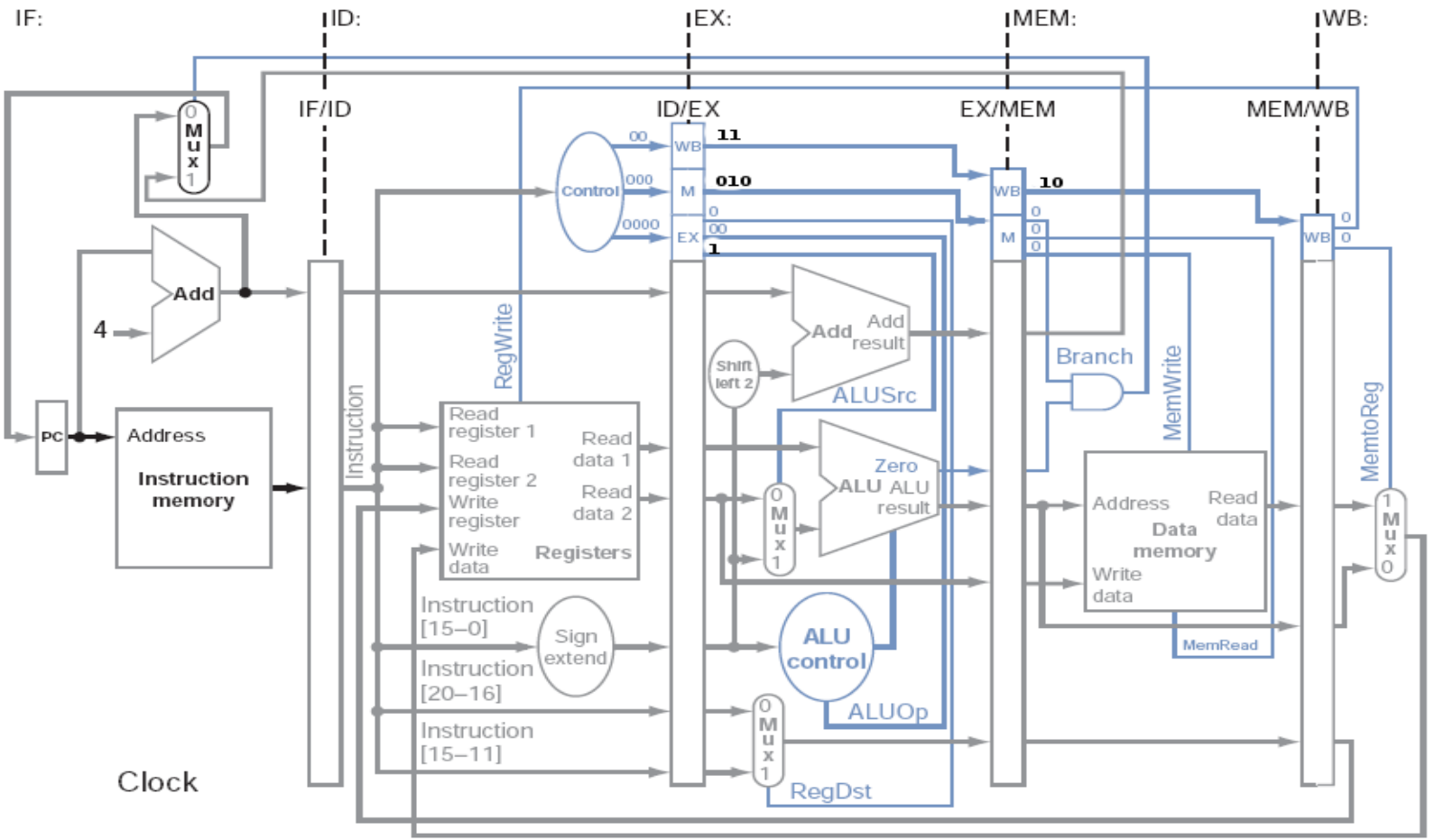


FIGURE 6.14.10 A blank single-clock-cycle pipeline diagram with control. (With Corrections)

Pipelined Processors

- Combine all the positives of the single and multi-cycle machines.
- Behaves exactly like an assembly line.
- At every clock cycle:
  - An instruction complete one stage AND
  - A new instruction is issued
- Latency is instruction dependent (and reasonable)
- Throughput when pipeline is filled: CPI = 1
- BUT...

Pipelining Hazards: Overview

- These are defined as situations preventing execution of the next instruction
- Three main types:
  - Structural Hazards
  - Data Hazards
    - This has 3 types itself
  - Control Hazards

Where to find more

As I won't be updating these slides as frequently during the quarter, more info can be found on course webpages of previous offerings of CMPE 110. In particular, quarters taught by Andrea Di Blas

Mention all pipelining hazards and break them down one @ a time with the solutions to them

Then mention problem of executing instructions in-order & solution of scoreboarding & then scoreboardings problems

To solve scoreboarding problems, introduce Tomasulo's algorithm & its problems

To solve Tomasulo's problems, introduce the ROB and its problems

Then mention problem of issuing instructions one @ a time & introduce VLIW & Superscalars

Processors & Peripherals

What We'll Cover

- Various Processors and their unique properties. These include:
  - MIPS trivial in-order processor
    - MIPS R3000
  - First Scoreboard Processor
    - CDC 6600
  - First processor to implement Tomasulo
    - IBM 360/91
  - ARM Cortex-A8
    - Used in iPhone 4s
  - Intel Core i7
    - What most of your PCs are using nowadays

- Various Peripherals our processor interacts with. These include:
  - Disk
  - Keyboard
  - Mouse
  - More to come

What We'll Cover (cont.)

- Other components of the processor we haven't mentioned yet (very briefly). These include:
  - PC
  - Control Logic (The State Machine)
  - Arbiters
  - Sign-Extenders
  - Shifters
  - Shift Registers (primarily used for WB)
  - Multipliers
  - Dividers
  - Faster Adders
  - Branch Predictors
  - LD/ST Queue
  - Issue Logic
  - Comparators (Mainly for branch evaluation)
  - Buses

Threads

The General Machine Model

How the Different Components Hook Together