CS M151B Video Notes
W 1 M Lec 1-4-16
1_1

- Introduction to Computer Architecture
- Follow the textbook as closely as we can
- Tests will be whatever is on Lecture slides or in class

Computer Revolution

- Computers are ubiquitous
- Devices make use of small integrated circuits
- What has been driving this is Moore's Law
- Doubling of transistor density for every new generation of technology.
- As we add more transistors, we have more capacity to compute resources.
- Moore's Law has been a pretty good indicator of how technology grows.
- Moore's Law have made novel applications in computers feasible.
- Tremendous use of the WWW, cloud computing, search engines, etc.
- Massive amounts of data, augmented reality, etc.
- All these applications are enabled by this tremendous amount of compute. How do we design machines for this?
- Different classes of computers
- What kind of computer should I get?
- Depends on your intent for using that computer
- Handle a wide range of different tasks
- Big difference in type of performance for these personal computers depending on what you buy and how much you spend.
- Personal computers
- The ones we use
- Server computers
- Used for cloud computing, file sharing, etc.
- Have higher capacity in compute resources, storage, and performance
- Throughput
- Work/Time
- Designed for reliability
- Size varies from tiny computers to extremely large computers
- Supercomputers
- Generally more higher end for scientific and engineering computations
- Custom-made generally
- Sometimes made from commodity components i.e. Intel Zeon 5's (?)
- Nvidia GPU clusters
- Large amounts of compute nodes and crunching large computation.
- Relatively small margin of the compute market.
- Example:
- Physics comptuers
- Embedded computers
- Inside other devices i.e. cars, phones, tablets, etc.

- The line between personal computers and embedded computers gets blurry
- A little more stringent power/performance cost
- Fixed battery life and performance target that is easy to reach because it is designed for a certain task.
- Microwave computer
- Does not have to handle Microsoft Word
- Performance can be very predictable
- The PostPC Era
- As we have moved more towards tablets, smartphones, devices that are wearables and away from the desktop….
- Change in market share!
- Sale volumes over a small period of tine for different devices
- Tablets and smartphones have become more popular in recent years compared to PC's
- As a designer of a system, which machines are we likely to see to take precedence.
- Mobile devices that are more energy efficient.
- How to design devices that bridge that gap
- Personal Mobile Device (PMD)
- Has battery life restrictions
- Has to be connected to the Internet to retain full capability
- Has full power when backed up by the cloud
- Cloud computing
- Warehouse Scale Computers (WSC)
- Software as a Service (SaaS)
- Running a compute resource on the server and running it as a client from your machine
- Server is providing all the computation
- You are paying to use the resource on someone else's machine
- Benefit of the cloud is that the server manager can upgrade the software and handle all that cost for a large array of users
- Users don't have to pay much!
- Server admin provides all the upgrades, security, etc.
- Sort of like fiefdom
- What You Will Learn
- How programs can be translated into machine language
- Going into detail on how the hardware implements these different instructions for the machine language.
- How to improve program performance from the hardware perspective.
- Parallel processing
- Mechanism to improve performance

Understanding Performance

- Performance can be broken down into many contributors

- Software writer implements a particular algorithm - how efficiently it can handle a computation.
- Compiler can have a significant impact on performance
- Machine code is the language understood by the processor
- Compilers optimize it and transform it into actual machine code.
- Software tools that are used to create machine code can have a tremendous impact on performance overall.
- Processor and memory system have a direct impact on how fast instructions can be executed.
- How do we take a set of instructions and make them run as fast as possible
- I/O system
- Defines how fast data can be pushed into or out of that memory process system.

Eight Great Ideas

0. Design for Moore's Law
0. Big tool chain of components that can be used
0. Transistor density
0. Provides a lot of silicon to build our chips
0. Voltage does not scale well so it is hard to design
0. Technology in circuit design and manufacturing really drives our innovation
0. By designing for Moore's Law, as we design for increasing transistor budget, how do we spend for it?
0. Use abstraction to simplify design
0. If you can design for something without knowing how it works, we can look at very complex designs
0. Driving cars
0. Don't need to know how engine works or tires use friction
0. Don't need to know every facet
0. Turn wheel, hit gas, hit brake.
0. Interface to abstract beyond many details
0. Go into components of the architecture
0. Make the common case fast
0. If you have something that you are going to be doing frequently i.e. driving your car to work everyday
0. You want to make this as efficient as possible
0. If you have a processor that does integer instruction but not floating point, don't make floating point fast
0. If you want to have an impact on the processor design, you need to know what is being used to cut down your runtime.
0. Put innovation and effort into areas where it can pay off the most
0. Performance via parallelism
0. Extremely crucial to today's performance
0. Instructions take a certain amount of time

0. One of the more easy ones is memory instructions
0. Can take 100's of processor cycles
0. Don't do 1 at a time, it makes more sense to do multiple memory operations simultaneously (in parallel)
0. Performing operations in parallel and if you do something that takes a long amount of latency, think of how to improve efficiency
0. Performance via pipelining
0. A special case of parallelism
0. Instead of having multiple resources that are the same, we have an assembly line
0. Move data from stage to stage
0. Allows stages to overlap in time
0. Think of a car assembly line - different robots putting together a car
0. One puts on the windows
0. Other puts on the wheels, etc.
0. These jobs occur in a sequential order
0. Can be done on different cars at the same time
0. Performance via prediction
0. People think computers are always correct in terms of computation
0. Difficult to know what the software will do ahead of time
0. Not quite at the end of the program yet, you aren't sure what it is going to do.
0. The branch instruction
0. if statement in the code
0. You don't know at some points in time whether the if is taken or not taken
0. Instead of waiting to know if my branch was taken, you can guess and work ahead of time
0. If wrong, you have to recover and waste some time
0. If you are right, you are golden!
0. This class focuses on branch prediction.
0. Hierarchy of memories
0. Difficult to make high capacity memory that is also low-latency
0. A large amount of storage that can be accessed quickly
0. Hard disk storage is huge but slow
0. Orders of magnitude faster but also orders of magnitude smaller
0. Idea that instead of having one type of memory, you have a large array of different types of memory that act like one memory
0. Dependability via redundancy
0. Redundancy is generally bad, but in terms of reliability, this can often only come via redundancy
0. If one component fails, the others will make up for it or correct the error.

Below Your Program

- We mostly think in terms of the software domain
- Take high-level software and translates it down to something the hardware can deal with

- Application software is the most enveloping circle
- What is in the middle is the actual hardware
- An example of this is the application software (Safari or Chrome) running on top of System software (OS X, Linux, Windows)
- Application software is managed by the operating system that is protected by the disk or memory
- Compiler translates it down into machine language - hardware is the main focus of this class
- Know what the hardware is provided with and what optimization can be done.
- What can be done in software and what can be done in hardware.
- Place more burden on the software or place more burden on the hardware.

Levels of Program Code

- High-level language
- Tracing a drop of blood through the body
- Trace it down from a high level language down to a lower level of abstraction.
- In higher-level languages like Java, you have levels of abstraction to hide complexity.
- Gives you abstraction closer to the problem you are dealing with.
- High-level language is supposed to help you write your code and make it reliable.
- High-level languages can be the same across different systems and the language can be consistent.
- Provides a great method of portability across different hardware, operating systems
- Assembly Language
- Ultimately, a compiler will optimize the code and make it as efficient as possible and parses it.
- It provides instructions in assembly language, which the machine will understand.
- Assembly language is NOT portable, it is specific to a certain machine!
- Once you have assembly language, it gets translated down to an assembly language representation.
- Creates a set of 0's and 1's
- Hardware representation
- Hardware will take that code and send it as a set of instructions.
- Binary machine language

1_2

Components of a Computer

- Box that represents the actual hardware of the computer
- Sitting on top of the box is an interface

- Compiler transforms the high-level language into cubes that the computer will understand.
- Computer is responsible for executing whatever the cubes tell it to do.
- Computer corresponds to some input or output.
- Memory is used to store data and a control and data path.
- Control organizes how data flows through the data path. This handles the processing of that particular data.
- User of a calculator would be controller
- Lots of different types of computers out there.
- The big picture is applicable to all types of computers
- Contain some interface, some software running on top, and how the design might be evaluated or optimized
- Opening the Box
- Old-generation iPad components
- Focus on the actual board and the processor itself.
- There is one chip that has an Apple on it (since it is Apple)
- Single chip or piece of silicon and there are integrated a variety of components including a processor
- A more recent SoC
- System on Chip
- Snapdragon 810
- This combines all the components and the different designs into a single piece of silicon and they can communicate more effectively together and they don't have to be constrained
- Attached to a motherboard and there is a limit to how much bandwidth you have.
- SoC: Only limited by metal layers on the actual design.
- Much lower latency and tighter integration of components has a lot of benefits
- You may have more heat problems that are heating the chip differently and you may have larger and larger chips.
- SoC is the tight integration of a variety of components
- Heterogeneous design: a variety of things combined into a single piece of silicon
- Main processing unit of the Snapdragon.
- You can see the Adreno processing unit
- OpenGL - embedded standard component.
- Not as powerful as a board from Nvidia but it is comparable in terms of its design and approach of architecture.
- You can see multimedia processing on the lower right.
- Encoding and decoding media and audio.
- Look at other types of components on here and the general idea is that this is for all major computer resources on a mobile platform.
- Inside the Processor (CPU)
- Consist of Datapath of control - component of processor where data is being processed or manipulated in some way.

- Datapath - flexible in terms of data it can take.
- Can handle a wide array of different instructions.
- Cache memory
- Really just small fast memory that is close to the processor
- Used in the near future
- Used in the top of the memory hierarchy.
- Consists of other components that have larger capacity.
- Two core designs from the Snapdragon A10
- Cortex-A57 and the Cortex-A53
- Has nothing to do with the Apple A5 SoC design.
- A57 and A53 are part of the Big-Little design
- Rather than have one processor design to do everything - we have a more powerful CPU with bigger caches and more capabilities and we have a simpler or little core that has more power efficiency.
- Push the performance envelope that saves energy
- If you need high-performance, use big core
- If you need to save energy, use little core
- Helps you attune to the need of the application.
- Don't get into too much detail, but get into analogous components that make sense.
- See how it relates to the actual cortex design.
- Abstractions
- Deal with complexity
- Really large number of modules that make up an individual processor
- Use abstraction to modularize the design as much as possible.
- Instruction set architecture (ISA)
- Language the compiler will use to speak to the processor
- Guides what the architecture design will need to handle or describe.
- The architecture will have a clear picture of what it needs from the software.
- Design tradeoffs of designing at ISA
- Focus more on the implementation
- Technology Trends
- DRAM capacity over time
- The y-axis shows the size of DRAM: Main memory we have on our motherboard
- Used after caching: if you don't find something in cache, go to DRAM
- If you don't find in DRAM, look on disk
- Over time, we can see that it has gone on an exponential scale in terms of its overall capacity and it's shown on the x-axis.
- Dramatic scaling of DRAM capacity - pretty stable relative to the speed of transistors
- The gap between memory speed and processor speed has been growing.
- This is a great example of how technology trends affect processor design.
- Support lots of data share memory concurrently.
- Faced with the problem that there is a high latency of DRAM access.

- You need to come up with an architecture that hides that latency.
- Relative performance/cost
- The bottom shows the trend that we have had going from Vacuum tubes (50s) to Transistors (60s) to more and more integrated circuits we see nowadays.
- Circuits are made of closer and closer circuits that are smaller
- Growing at a dramatic and ridiculous scale
- Lots of area and technology where we can shrink things to being close to one another.
- Plethora of area in terms of design
- Problems in terms of voltage
- Unable to handle heat well.
- Complicating the matter and pushing towards accelerator rich design.
- What is coming for the future to combat these trends.
- Semiconductor Technology
- Silicon - main focus (conventional materials)
- These semiconductors can be combined to form conductors, insulators, and switches.
- This class is looking at an architectural level
- We assume it is assembled and we have to look at what is put in an actual chip.
- Much lighter abstraction from circuits on a computer.
- This slide shows the chain of how a Silicon ingot gets turned into an actual chip.
- Cylindrical, sausage tube that is manufactured and sliced into circular wafers.
- These wafers go through lithography, which patterns the circuit design.
- Once they have the circuit design, they have a tester to see which patterns work and which don't.
- Potential defects that can occur on circular ingots.
- Wafer where we know which components are functional and which are not.
- Those dies that are malfunctioning are removed.
- Those that work are bonded to a package with a hard encased material.
- This prevents electric problems and this packaging can create errors.
- Any ones that are not working are shipped to the customers.
- Nuance for components that have lots of pores i.e. cells from IBM or Sony.
- SPE's: Synergistic Property Elements
- We can disable one of these particular components
- Improves the overall yield and this turns off components!
- Yield: For a given wafer, how many dies/slices of chips would be functional
- Intel Core i7 Wafer
- The ones on the edges of this wafer are going to be thrown away.
- Of the actual parts that are full working chips, some will have defects and some will not.

- One thing to see here is that you should build an intuition for the size of the chip.
- Compose more of the wafer and smaller chips will be able to get more yield out of the wafer since there is less area taken up.
- Integrated Circuit Cost
- Amount of dies you have in the wafer is relative to the perimeter.
- The nature of it being circular, there will be waste
- It is a proportional amount and big dies
- In terms of the yield, it will be relative to the area
- The larger area means you are more likely to have a defect.
- Understand the intuition for what influences the integrated circuit cost.

W 1 W Lec 1_3

- Performance
- There are a lot of aspects that will be important as we continue exploring computer architecture.
- Even if the goal is to not improve performance, it will always be an important factor.
- The book likes use analogies i.e. an airplane.
- Compare a variety of different airplanes on which has the best performance.
- Boeing 747
- y-axis shows 4 different types of planes
- x-axis has different attributes
- Upper left - capacity
- Bottom left - how fast it can get people there
- Upper right shows cruising range
- Bottom right is a combined metric - throughput, how many people you can get at a particular time.
- Response Time and Throughput
- Response time
- How long it takes to do a task
- If I run my compiler and I am compiling a particular program that I wrote (2 minutes) - that is the response time of the compiler.
- Throughput is particularly important for servers
- Web clients can be accessing the server, and the total work done per unit time might be the amount of web transactions per minute
- Faster processor with a faster clock - this would help with response time and potentially throughput. It will definitely help response time though.
- Adding more processors may help if you have parallel applications that can be decomposed into particular tasks.
- If more work can be done in parallel, think about throughput improvement to be an improvement in parallelism.
- Think about response time as a latency reduction. Processor will work faster on a given task.

- Relative Performance
- Think about response time or latency.
- Measure this using relative performance.
- If I have a program running on an architecture, the amount of time it takes would be the performance time
- We describe it as an inverse relation.
- If we compare X and Y, and we say X is N times faster than Y, we divide both X and Y, and the output should be N.
- If we have the program which took 10 seconds to run on computer A and 15 seconds to run on computer B, assuming we have the same high-level program, the execution time would be 15/10 = 1.5
- In this case, A is 1.5x faster than B on that particular program.
- Measuring Execution Time
- Start a stopwatch, record when the computer finishes, see how much time it takes for the program to execute on the computer.
- Anything the OS does during that time.
- This includes everything.
- Look at how much your OS impacts you and all this is important.
- Focus on CPU time
- Amount of time in CPU processing a particular job.
- Doesn't include I/O time wait, OS impact on maintenance tasks
- OS share of servicing CPU requests ONLY.
- user CPU time and system CPU time together
- No overhead of unrelated tasks are included
- CPU Clocking
- How much time we spend on the CPU - the CPU clocking (processor governed by a clock)
- This side says it is constant-rate and there are no processor designs that are quite common in this way to deal with different power performance requirements.
- This clock has ticks and they can be synchronized with respect to one another.
- The cache is stored in the register and this is governed by the clock.
- Asynchronous designs don't use a clock
- Save a lot of power
- Performance suffers since we don't have fast synchronization.
- Counterflow was a great asynchronous design but we won't cover it.
- It is possible to use multiple clocks, but let's assume we use 1
- Particular period of how long it takes for 1 clock cycle and as an example, 250 ps is shown as a clock period.
- Inverse that, you get the clock frequency.
- CPU Time
- The clock allows us to divide the processor view time into cycles
- From the CPU's perspective, the task takes a certain number of clock cycles
- Overall amount of time spent in clock cycle depends on the CPU

- If I multiply 1000 clock cycles but one nanosecond clock, it tells you how much in the real world.
- Number of CPU clock cycles divided by clock rate
- Think about CPU Time in this perspective, you can improve performances as follows:
- Make it more efficient by reducing clock cycles
- Make the clock work faster and you don't have to wait as long for ticks
- Trade-off clock rate against cycle counts
- CPU Time Example
- Computer A has 2GHz clock and 10s CPU time
- We want to make a new design
- Reduce it to 6s CPU time
- How fast do we have to increase Computer B to be?
- If we make the clock faster, we assume that any increase in the clock will cause a 20% increase in the number of clock cycles
- The faster the clock, the more potential clock cycles you have to incur
- Assume it is always 20%
- Put it into equations
- Clock Rate_B = Clock Cycles_B/CPU Time_B
- Clock Cycles_A = CPU Time_A x Clock Rate_A (simple algebra)
- Instruction Count and CPI
- Clock Cycles: How many cycles it takes for the processor to execute a particular application
- Broken down into Instruction Count
- How many instructions the processor actually executes
- The instruction count loops through a for loop 100x
- 5 instructions within that for loop
- Even though the code, only has 5 instructions within it, it looks like 500 because it executes it as a dynamic count
- We want what instructions are executed by the processor
- If some cycles have different amounts, look at the average
- Depends on the instruction makeup of the application we are considering
- CPI depends on the machine and the actual code we are talking about with the machine
- CPI Example
- Computer A has cycle time of 250ps and CPI of 2.0
- Computer B has cycle time of 500 ps and CPI of 1.2
- Same ISA (same instructions and program)
- Let's talk about what this means
- These processors have different cycles - one has twice the clock rate of the other
- CPI is different though.
- Comes from the program being executed (instruction mix of the program) and the efficiency of the instruction
- If it uses the same compiler, the binary used will be identical.
- If binary is identical, the instruction count is exactly the same

- Conditions of running will be exactly the same and CPI difference comes from the implementation, not the instruction mix.
- Which is faster and how much?
- I * 500 ps
- CPU Time of B would be Instruction Count * CPI_B * Cycle Time_B
- Even though A is faster (500 ps * I) and 20% faster
- This larger CPI was outweighed by the clock rate for this particular example
- CPU in More Detail
- Weighted average
- The more diffcult of the 3 to think about - mix of both hardware and software
- Each instruction you encounter has some amount of latency
- The way you form this weighted average is you take the instruction count of a particular instruction and you sum of the total number of clock cycles
- Gives a relative frequency of how often that instruction occurs in the overall scheme of things
- CPI Example
- Alternative compiled code sequences using instructions in classes A, B, C
- Class of instruction might be an arithmetic operation
- Takes some latency for arithmetic, different latency for loading, etc.
- The CPI for a given class in the example will vary based on the instruction
- Different sequences will vary.
- Instruction count of sequence 1 will vary.
- Sequence 2 has 6 instructions
- To get CPI, look at sequence 1 with instruction count = 5, A = 1, B = 2, C = 3
- For sequence 2, you have A = 4, B = 1, C = 1 => 4 * 1 + 1 * 2 + 1 * 3 = 9
- Performance Summary
- 3 components
- 1. Dynamic instruction count - how many are executed
- 2. Clock cycles per instruction
- 3. Cycle time
- Depends on instruction set architecture that is used and the actual program - whatever depends on the algorithm design.
- Clock cycle/instruction can be influenced by anything in your program.
- Changes instruction mix
- Algorithm can affect instruction mix
- Impacted by actual machine organization in the processor.
- Clock cycles are influenced by a lot of things across the spectrum.
- Really doesn’t matter on the compiler and/or the ISA.

1_4

- Power Trends
- Performance is important for the analysis of architecture

- Power is also important and is a first-class constraint
- Very real impact on what is included
- The diagram shows that the clock rate in blue and the power in black for a variety of microprocessors
- Only goes to 2012 but trends are fairly comparable.
- The first y-axis shows clock rate in MHz
- The clock rate steadily increased shown here in what is 82 to 2001
- A lot of performance increase and architectural innovation, but power also grew dramatically
- It grew sharply to around 100W
- Design was cancelled because of power concerns
- This led to a shift in how we handled computing
- Push for large monolithic cores with tremendous complexity
- Multi-core that is relatively simpler with what we have in a monolithic core.
- Run different tasks depending on what the programmer wanted.
- The power trends shows that the power has been steadily dropping as we optimize for many cores.
- Power is proportional to capacitive load on actual circuits * Voltage^2 * Frequency
- Voltage does not scale down
- Leakage current
- There needs to be other alternatives on how to reduce power.
- Capacitive load is impacted by what you integrate onto the actual chip
- To some extent, we need to improve power component and smaller designs
- Using transistors in a smart way to avoid leaking transistors.
- Voltage in this case is talking about a drop in the operating circuit.
- Looked at for dynamic tuning.
- May drop the voltage if the processor is not used to save power
- Frequency
- The clock rate or speed of the chip
- Also a lot of designs that look at dynamic frequency scanning
- Speed Step technology from Intel - DBFS is Dynamic Voltage and Frequency Scaling
- Reducing Power
- Impact we can see for these three components
- P_new for power of the CPU
- Assumption is that there is a drop of 85% of the original CPU for capacitive load
- Power saving design - not necessarily for performance
- Key optimization - how much various reductions on overall design
- Large impact from these drops goes to about 50% of the original.
- We can see the dramatic impact on saving for these various components.
- Power wall
- Mini core revolution we had around 2002-2003

- Without reducing voltage or heat because of increasing power density - we see components move closer together and it is difficult for conventional cooling systems to move out that much heat from an area.
- How do we scale performance in the face of power?
- Uniprocessor Performance
- Shown on a relative scale to Vex and performance increases at a dramatic rate - until 86, it was a 25% year improvement in performance
- 86 to 2003 - 50% year improvement in performance
- As performance continued to grow, we saw 22% decline per year.
- The inability to remove heat, scale voltage, etc.
- Instruction-level parallelism
- Related to finding parallelism in a single-threaded application - deals with out of order execution.
- Hardware tries to find a way to organize the code than the way the compiler could write.
- Instruction level parallelism
- Create more capacitive load and this made things difficult to scale in terms of power.
- DRAM latency is not scaling well relative to transistor latency.
- Memory wall tends to dominate and without a way to attack that, performance couldn't grow as much in this time frame.
- Multiprocessors
- The idea was that with a multicore microprocessor, you have more than one core on a single chip.
- Pushes off some complexity to the compiler writer and there is a need for parallel programming
- Break the program into parallel threads
- Different from instruction level parallelism because hardware takes care of it.
- In this case, the user is responsible for taking care of parallelism
- Finding a way to balance load and synchronize these tasks is a different subject and it's hard!
- CS 133 is an important class for this!
- Prevalent with hardware and forces you to get into that mode of parallelism!
- SPEC CPU Benchmark
- Performance of microprocessor depends on both hardware and software.
- Instruction count makes up execution time
- Instruction makeup and how these instructions are executed by the hardware.
- Both of them have an application dependence.
- There needs to be a standardized way of measuring the applications.
- One benchmark is heavily used in research community and is called SPEC CPU Benchmark
- One of them is a CPU Suite - geared towards exercising the actual processor core.

- Not a lot of use of a GPU or accessory - push the CPU as hard as it can.
- The most recent version is the SPEC CPU 2006 Suite
- Integer portion and Floating Point portion - computed using geometric mean
- Integer Suite - performance from the Core i7 from Intel - you can see some notion of what magnitude of instruction count we are talking about.
- Execution time - benchmarks themselves
- Others, you have to see Microsoft word - HTML browser.
- Some common apps.
- Two games in this particular suite
- Pathfinder
- Video compression
- Some of these may be a bit more obscure
- Pitfall: Amdahl's Law
- Good questions to think about in terms of what it means for tradeoffs and concepts
- This comes back to making the common case fast - if you are trying to optimize or improve an aspect of the computer - you will see a proportional improvement
- See a proportional improvement that is quite high
- Example:
- Imagine your multiplication operations account for 80s and the total execution time is 100s
- You decide to improve multiplication to 5x overall improvement.
- How much can you improve multiplication and get a 5x improvement?
- You would have to reduce 100s to 20s -> the total improvement should be 20s.
- Time taken by multiply is 80s divided by n, which is your improvement factor.
- The unaffected time is 20s and that is what is left over.
- The conclusion is that this cannot be done, you cannot drop 80 by enough to make 80/n = 0.
- Understand what kind of impact will limit the improvement you will see.
- Pitfall: MIPS as a Performance Metric
- Not the ISA we all know and love
- Millions of Instructions Per Second (performance metric)
- Doesn't account for
- Differences in ISA's between computers
- Differences in complexity between instructions
- Take execution time and remove execution time from it.
- If you don't consider instruction count, you are negating several important factors.
- Do not use MIPS, focus on pure execution time.
- Concluding Remarks
- Introduction to the area, many different things that are touched on.
- Getting into detail how things are implemented - it will come out.

- Have a good understanding of performance, what influences instruction count, cycle time, and at a base level where we are in Computer Science and Engineering
- Looking at hardware design, NOT circuit design or transistor design.
- Below level of operating systems but above physical design and circuit design.
- We will start looking at optimization slides and start going into hardware designs as a whole and machine organization.

W 2 M Lec 1-11-16
2_1
ISA - Instruction Set Architecture

- Instruction set: Repertoire of instructions on a computer
- Language spoken between hardware and software.
- Different computers can have different instruction sets
- You can't run a program compiled for ARM on x86
- Old computers had relatively simple instruction sets
- Modern computers also have simple instruction sets
- Complexity of instruction set and how this impacts computer architecture design

Key ISA Decisions

- Bridge design trade-offs and ISA spaces
- This ISA Decision -> what type of thing the ISA designer or writer wants to come up with
- Understand why certain decisions are made and look in-depth why these are made
- Operations that are supported by that instruction set architecture
- How many bits would it take to represent that particular instruction?
- Operands include what data components are operated on.
- a = b + c
- a is the destination operand
- b, c are source operands
- + is the operation
- We could have done this with two distinct operands
- Know how to specify these operands
- How do we compute these operands
- Some complexity in these operands
- How many formats do we support?
- Instructions are in memory
- Stored just like data and have to be fetched from memory by the processor
- Size impacts how much footprint it has in memory and how much of a size it takes up

Main ISA Classes

- Talk about how the line is blurred between the two
- Complex instruction set computers
- Earlier models
- Digital VAX (1977) and Intel x86
- Large # of instructions
- Rich set of instructions and a lot of things you can do
- Very specialized and have to be supported by the hardware
- Makes software job relatively easy
- Works really well in the early days when memory was expensive
- Make it compact and fit into your machine
- Hard to verify and hard to build
- Difficult to make it fast
- Once compilers got better and memory got cheaper, we moved to RISC ISA's
- RISC
- Almost all machines of the 80's and 90's are reduced instruction set computers (RISC)
- Had a lot of headway and there were fewer instructions
- This does not mean lower instruction count, it just didn't have as many choices
- You have more instructions executed since they did simpler things
- Allowed pipelining and parallelism
- Smaller pieces and tasks could be broken down
- Overlapped if necessary than complex instructions
- There may be dependencies that are more difficult to extract
- Why is it that CISC is one of the more popular ISA's?
- Talk about it more in class
- In architecture's like x86, there is a translation done inside the machine
- The micro-ops like RISC, the Intel appears like a CISC architecture but it makes use of all the CISC tricks internally.

MIPS Instruction Set

- We are going to build a machine
- x86 is unreasonably complex
- It makes sense to start with an ISA that is already RISC
- MIPS has better support and we have experience with MIPS from CS 33
- Not most applicable outside of school but it is a good overall RISC ISA

Arithmetic Operations

- Look at add instruction from before
- Example of add instruction at the assembly level
- Three operands listed
- The first operand is the destination operand
- Pseudocode because MIPS does not let you use general memory locations
- You have to specify these in registers

- Talk about it a little more abstractly as pseudocode
- Arithmetic operations follow this same format
- One indication of what function is being performed
- There are some other operations don't make use of one of the sources
- Same form and same formatting
- Makes it easier for architecture to implement it.
- Design Principle 1: Simplicity favors regularity
- Simplifies the hardware's job
- Simplifies data path of an ALU - expects values from same destination and same sources
- Less complexity for the validation of design
- Cost of ensuring correctness does not become burdensome.
- Power management can be done more readily if you have a simpler design
- Arithmetic Example
- f = (g + h) - (i + j)
- This has to get translated down into a number of assembly instructions
- CISC architecture can have one instruction that does all of this
- RISC like MIPS will break this down into multiple instructions
- (g + h)
- (i + j)
- subtract instruction that uses the previous two operand results
- Pseudooperands - have not been stored yet
- Get broken down into these fundamental primitives
- Two or three operands at a time that are being processed
- Register Operands
- Arithmetic instructions use register operands
- Three main places
- 1. As part of the instruction itself - immediate or literal
- Add instruction that always adds 5, 5 is the immediate and would be encoded
- 2. Operands can be placed in a register
- register: fast hardware memory placed on chip
- MIPS has register file where registers is stored
- Has a 32 entry file
- 32 x 32 does not correlate but it is currently what MIPS has
- Tradeoffs of more registers vs less registers
- Idea is that it is much faster to access a register than the actual location
- Give the illusion of a fast memory
- First location where we could go to grab the values
- Manipulated and controlled by the compiler and explicitly handles movement to and from register files
- Registers are orders of magnitude faster than memory
- If you modify it, write it back out to memory
- Registers are numbered from 0 to 31 (5 bits to specify the particular location)

- Those 5 bits give us a unique address into the register file
- One of the hard things is thinking about the 5 bits used to indicate which address you want
- Place to look up the data value
- Not weird magic turning two bits into 5 bits
- Those locations hold 32 bits, and 5 bits is used to hold a location to find where 32 bits are stored.
- Assembler names
- Some other special registers as well
- Everything is numbered from 0 to 31
- Memory and memory structures like SRAM
- Dense mesh of wires
- Larger memory structures don't scale well because of wires
- Having smaller amounts of memory makes things faster
- Access latency is much shorter
- Don't know that aspect, but the reason for this speed is related to physical design
- Register Operand Example
- Same C code as earlier
- In this case, if we assume f, g, h, i, j are located in $s0 to $s4, respectively
- MIPS code adds $s1 and $s2 together and put it into $t0
- add $s3 and s$4 and put it into $t1
- subtract $t0 and $t1 and put it into $s0
- For this coed to actually work, there have to be instructions for what C is used in g and this is called a "load word" instruction
- This code shows some of the benefit of going to a RISC architecture
- Think of these three instructions
- There is some parallelism involved
- First two instructions can be done at the same time since there is no dependency
- Subtraction however depends on both adds
- We will talk about this type of dependency and we need to develop intuition
- Certain instructions can depend on others

- Memory Operands
- Used heavily in CISC
- In RISC, there are operations that handle memory transfer into registers
- We will have explicit instructions to handle data movement
- One thing to think about for x86 is that we have a lot of complexity with how we address memory
- How do we access memory directly
- This made the programmers job a lot more difficult because they have to remember the different formats that are possible
- Deal with emory transfer instructions and see how they deal with that right now

- Memory is a gigantic array
- It is going to be byte-addressed
- Every entry in these arrays will be 8 bits
- There is an address or way of specifying a location that will be in that gigantic array
- Deal with the memory of 4 GB
- This means we have 32 bits to address that memory
- 2^32 = 4 GB
- We have 1 byte per entry even though we have 4 GB of total memory
- Each address to identify this will have to be 32-bits in the example
- In this case, for the MIPS architecture, words will be a 32 bit word
- We have 4 bytes that we are pulling out of memory
- Pull things out at a 4 byte granularity
- We still will make use of byte-addresses
- Ultimately provide a 32 bit address
- When we look at instruction encodings, and we know the result will have to be in our example 32 bits
- We have to make use of compaction for 32 bit addresses
- Keep in mind that ultimately, we have to have a byte address coming out of this request to memory.
- MIPS is Big Endian
- When you look at laying out a multibyte variable
- 4 byte integer
- Put most-significant byte at the least address of a word

- Memory Operand Example 1
- g = h + A[8];
- g will ultimately be placed into $s1, h starts out in $s2, base address is stored in $s3
- $s3 is a pointer to the address
- For the MIPS code
- For A[8]
- Start with the base address and compute offset and load it into a temporary register
- Compiled MIPS code
- To get to index 8, we remember this is a byte-addressed memory
- If A is an array of integers
- Every element of A will be 4 bytes
- To get to the location in A that is A[8], you need to go 32 byte addresses
- If A starts at 500
- 4 byte values make up 32 bits for A[1]
- A[2] would be at 508, 509, 510, 511, etc.
- A[8] would be 532, 533, 534, 535
- The first operand is the destination, the second operand is the source
- source is composed of value of 32 bit offset and the value
- Grabs the value and uses the sum as the address into memory

- This places 32 bit value into $t0
- Has to go to memory and grab the contents from the address
- Once it writes to the register file, the add will be able to take the value and place the result from $s1
- Memory Operand Example 2
- For A[12], this is the destination from the sum of h and A[8]
- Each entry of A will need four entries from memory
- h will be in register $s2
- Compiled code will grab from the offset of 32
- Pull contents into $t0
- Add instruction, add h to $t0
- Take $t0 and put it into memory
- sw (store word) has the same format
- For this sw instruction, $t0 is actually one of the source operands, and the other side is another source operand
- The destination here is memory
- Takes value added in previous instruction and place sit at $s3 + 48
- Load words and store words
- Grab a value from a register and load it into memory
- Registers vs Memory
- Registers are faster to access than memory
- Physical nature of registers
- Larger latency and capacity
- MIPS - need to have loads and stores to operate into memory
- Operate using arithmetic operations
- More instructions that have to be executed
- As opposed to a CISC architecture where you can directly access data values stored in memory
- Compiler has additional burden
- Has to try to use registers as much as possible
- Spilling
- When we run out of registers, and we only have 33 values we have to maintain and only 32 registers
- When we have value not currently in register, we take one register that is not being used or potentially being used
- Throw away current value if memory is already consistent
- Spilling can result in more loads OR more stores
- Makes register optimization VERY important
- Loads and stores are more expensive instructions

2_2

- Immediate Operands
- Two possible sources for operands
- Program itself may change from data inputs via inputs
- Values can change due to a level of indirection
- Immediate operands is a little different

- Embedded in the actual instruction itself
- Do not change, even with data inputs or execution over the instructions
- Treated as a read-only instruction
- Used for literal values that will not change
- addi
- $s3 is both the input and output of this add instruction
- 4 is another input, it is a constant
- It is always adding the constant 4 to the input value of $s3
- No subtract immediate instruction
- With a lot of these MIPS instructions, we want to save values
- In MIPS, the constant is only 16 bits
- Embedded into the instruction, but we will still be dealing with 32 bit numbers
- Because small constants are the most common
- We can embed the smaller value into the instruction to avoid having an additional load instruction
- Constants are often small can be used to reduce instruction count
- This makes the common case fast
- Otherwise, we need another instruction to load those 16 bits
- The Constant Zero
- MIPS register 0 ($zero) is used when you want to convert an add to a move operation
- This sacrifices some register space for the ability to have a simpler instruction set architecture
- Unsigned Binary Integers
- Go through some of the binary integers and talk about how that representation works
- Stick onto the next set of slides
- The idea behind this binary representation is that there is a number of 1's and 0's in sequence
- Represent higher and higher orders of magnitude
- Goes up to 2^n - 1
- Akin to our own number system but instead of base 10, we use base 2
- With an unsigned binary integer, this allows us to represent a range from 0 to 2^n - 1
- 1011_2 -> 11_10
- 2-s Complement Signed Integer
- The most significant bit is given a negative displacement, and this negative displacement is offset by the bits to the right of the most significant bit
- Range goes from -2^(n - 1) to 2^(n-1) - 1
- Bit 31 is the sign bit
- If it is a 1, negative
- If it is a 0, positive
- Mechanism to detect positive or negative value
- Some useful specific numbers
- 0: all 0's

- -1: all 1's
- Most negative: 1 then all 0's
- Most positive: 0 then all 1's
- Signed Negation
- Take a value and negate it
- Flip all the bits (complement it) then add 1
- If we were to add x and x's complement, you should get -1
- If you want to take -x, then you take the complement of x + 1
- Negate one of the input operands and then add 1
- Sign Extension
- Used throughout architecture for smaller values (16 bit, 12 bit, etc)
- You take the most significant bit in whatever representation you have
- Represent it multiple times to pad it to whatever bit level you want
- +2: 0000 0010 => 0000 0000 0000 0010
- -2: 1111 1110 => 1111 1111 1111 1110
- The hardware will do this as we have smaller values extended to 32 bits

2_3

- Representing Instructions
- How instructions are encoded in MIPS in particular
- In MIPS, they are encoded in binary and stored in memory
- AKA machine code
- Every instruction will take up 32-bits
- Fixed length instruction set
- Make things relatively simple
- Relative simplicity in how registers are specified
- In most cases, they make use of registers
- Instructions are regular
- Same size
- Regular number of formats
- Labels are mapped to integer values
- 32 registers
- Those are going to be 0 to 31
- These are going to have labels that match the representation
- 5 bits to represent a particular register (2^5)
- You don't need to know the mapping in particular
- Be aware there are 32 of them.
- MIPS R-format Instructions
- Useful for a variety of reasons
- 32 bit value overall
- Spit up into 6 pieces
- First bit chunk is called an op code
- Indicates it is an R-format Instruction
- This is just a way to determine what format the op code should use
- op - 6 bits
- The rs field is the register source: 5 bits

- rt is second register source: 5 bits
- rd is destination register: 5 bits
- shamt indicates how much we shift (00000 for now): 5 bits
- funct is the function code: 6 bits
- The way to think about is that the op code specifies what kind of instruction this is
- There is an additional field called the function field that tells you whether to add or subtract, etc.
- add instruction example
- Two source operands and one destination
- All three are registers
- $t0 is the destination
- $s1 and $s2 are the source
- op code indicates this is an R-format (000000)
- $s1 (17) is placed in rs field
- $s2 (18) is encoded in rt field
- $t0 (8) is mapped to rd field
- Indicate we are going to perform an add using the funct field
- Understood by control unit to perform the add
- All of these are needed to perform the ISA
- This is not an arbitrary thing
- Perform the ISA itself
- The instruction itself is just based on the R-format encoding
- This processor would use the opcode and decode the rest of the instruction appropriately.
- Hexadecimal
- More compact representation that is a bit easier to read
- Akin to binary in that there is a base
- Base 16 in this case
- Increase by orders of magnitude of 16
- Mapping from 0 to 15 in both hex and binary
- f represent 1111 (15)
- We can reduce a binary value into a sequence of hex characters
- MIPS I-format Instructions
- We will see there are some exceptions to that rule with formats we come up with
- They follow the same kind of pattern
- 1 input, 1 output, and a pattern that tells you what to do
- When we look at loads, stores, and branches, we will know what to get into.
- Immediate is a constant that is embedded to the actual instruction itself.
- Breakdown of 32 bits
- 6 bits for the op code
- Consistent across all formats
- Here the rt is not always a source
- Sometimes it is a destination

- Last 16 bits are part of an embedded constant in this I-format instruction
- How those 16 bits are used depends on the instruction we are dealing with
- Add this constant or address as an offset or base
- If we want to store a 32 bit immediate, we would still need an op code
- There is tradeoff with just using 16 bits.
- Fundamental tradeoff to the amount we allow in our instruction
- Stored Program Computers
- Instructions exist in memory itself.
- The manipulation of what goes in memory is often program-related
- How do we extract it out in a fast, efficient way?
- Logical Operations
- Another concept we have covered in previous classes
- Java has a shift right (>>>) that is different from C
- Perform manipulation on individual bits
- Performing very simple discrete sets
- Shows an example of operations in C and Java
- AND, OR, NOT
- The OR immediate and AND immediate are in the ALU's
- Shift Operations
- Can be done in left or right logical
- sll is the shift left logical
- Takes in register input and shifts it left, filling it with 0's on portions that are shifted
- Equivalent to multiplying by 2^i
- srl is the shift right logical
- Equivalent to dividing by 2^i (unsigned only)
- AND operations
- Very useful for masking
- Take two inputs rs and rt
- Take the values containing those registers and apply AND to each individual pair of bits
- Written to t0 and part of the R-format instruction
- OR operations
- Used to build up or construct an address
- Specify source operands
- rs, rt, rd fields
- OR would have a unique function field
- The key difference between distinguishing OR, AND, and ADD is from the last six bits
- NOT Operations
- Not present explicitly in MIPS
- NOR is used instead
- Allows you to double the use of NOR as a NOT if you have a $zero register

2_4

- Conditional Operations
- AKA branches
- Instructions that result from an if statement, while loop, etc.
- Any condition is dependent on the data
- Three conditions:
- beq: branch if equal
- Checks register contents and branch to L1
- Encoding is a little tricky and if we talk about how this label actually works, we can think of it as a particular address in the text segment
- bne: branch not equal
- j: jump condition
- This is an unconditional jump no matter what
- Equivalent to a goto statement that exists in certain languages
- Program counter that keeps track of what instruction is on your processor
- PC will always be incremented by any instruction
- Let's you change anything besides essential construction
- Compiling If Statements
- Check if i == j
- Assign value of f
- The way to exit the same sequence of code signifies a divergence of control
- Pick one of the exits and think about this in the context of the code
- bne to check if things are not equal
- If you have i and j stored in $s3 and $s4, you branch to else
- Instructions are located in contiguous memory
- If you have an instruction at 500, the next instruction should be 504
- Instructions need 32 bits to be stored
- If you start at 500, you add 4 to the address to get to the next 32 bit instruction
- If your bne is at 500, add would be 504, j is 508, and else (sub) is 512
- This assumes that memory is byte-addressable
- Follow the branch to the else, PC would change to 512
- If they were equal (because of bne), I would not take the branch
- Jump will take you to the Exit at 516
- Exit would be some other instruction
- Linker will figure out which address to link up to a particular instruction
- Key point: if statement has two paths that come
- Take the branch
- Go to else (taken branch) vs a not taken branch
- The not taken branch is what instruction comes next
- That would be the instruction after bne
- Cover some of the problems that come from a processor pipeline
- Compiling Loop Statements
- Branch is a means of exiting the loop
- Assume i starts out in $s3, k starts out in $s5, save is in $s6

- Start by shifting value of i by 2
- This assumes that we have an array of 32 values (integer)
- Each integer needs to be stored in 4 memory locations
- Increment by 4 each time, so that's why we left shift by 2 (2^2 = 4)
- Add to base address
- Load word goes to value of address contained at $t1
- Grab value and place it into $t0
- bne: if not equal, branch to Exit, otherwise, you would add 1 to $s3
- Jump to loop to continue executing
- Basic Blocks
- Sequence of instructions with no other branches except at the end
- The guarantee that you have is that your entry point is at the beginning of the basic block
- Everything else in between would be executed regardless if you have an exception or not
- To manipulate code for purposes of optimization, you would treat it as a coarser granularity
- Treat as basic block by basic block
- You can describe the control flow
- You can look at predication or ways of combining things together
- Focus on single entry point at the beginning
- Single exit point at the end
- More Conditional Operations
- slt: R type instruction
- Does NOT really change to a PC, writes to a register value
- Keep things in sync with the book
- slt means set less than
- Compare if rs is less than rt
- If that is true, set rd to 1, otherwise set rd to 0
- There are only two possible outpost of the slt
- slti compares an input to a constant
- Use in combination with beq and bne to compare equal and not equal to
- slt does not result in a branch, only a comparison
- Branch Instruction Design
- The hardware that is required for that functionality would be slower than for simple equivalents
- Don't need to impact the pipeline
- More instructions
- RISC philosophy is more clear
- Don't have hardware where critical path latency penalizes every instruction
- Fundamental design compromise
- Pseudo instructions
- Eventually translated down to real instructions
- Exist at the high level of assembly
- Signed vs Unsigned

- Variation between how they compare bits using signed and unsigned representation.
- The ISA and the architecture will have to distinguish between the types of architecture
- Focus on the slt first off.

2_5

- This section is not emphasized as much, keep it at a high-level
- Review of CS 33 and not used much in creating a data path
- Procedure Calling
- Particular type of branch
- Unconditional
- Won't be a condition where the branch is always taken
- Special case for returning
- Do what's called jumping and linking
- Saved current location
- Parameters can be placed in registers and we may have to place it somewhere else in memory
- Pointer to text segment of memory can be changed
- Require storage for the procedure
- Add another stack frame and increase storage allocation
- May involve additional procedure calling, but eventually, we will place it in memory for procedure calling
- The part we want to make clear is the jumping and linking portion
- Related to instructions on data path
- Register Usage
- The actual number index we use not he architecture level
- Look at that last register
- $ra
- This is dedicated to holding a return address from a procedure call
- Subtlety for procedure invocation instructions
- Procedure Call Instructions
- jal has two functions
- Changes PC to ProcedureLabel with a little bit of extra encoding
- Takes address of the next instruction
- Each instruction is 32 bits and we want the next memory location to give you part of that same instruction
- Go 4 addresses down to go to next available instruction
- Place that address into $ra
- $ra holds the address until we return back to that location
- jal and jr differ
- jal has one procedure label
- jr has a label it is jumping to
- jr has to be told which register to use
- If you want to use jr with jail, it is usually used with the register $ra
- Copies $ra to the program counter

- Continues execution with whatever was following the jal instruction
- Leaf Procedure Example
- Calling and returning procedures
- callee and save registers on the stack
- Important to understand (CS 33 material)
- Not a primary focus
- C code that is an example of a function with 4 parameters
- These 4 parameters fit in $a0, …, $a3
- f is computed in callee-save register
- Restore it after it is completed
- Place result in $v0
- MIPS code
- The basic idea is you want the procedure call where first instruction is addi
- Increment stack pointer and make more space
- Stack grows in upside down fashion
- Procedure body happens with the two add instructions and the subtract instruction
- Move is done by adding with a $zero register
- lw increments the stack pointer
- pop is equivalen to the lw and add instructions
- jr returns
- Benefits of procedure calls is that you can invoke it from multiple places and reuse the same code
- Have to use stack manipulations
- Some overhead of procedure calls
- Non-Leaf Procedures
- If you make multiple procedure calls, you can lose the return address
- Any arguments or callee save registers would have to be saved on the stack as well
- Non-Leaf Procedure Example
- Recursive factorial function
- MIPS code:
- Saves return address onto the stack and we can make further procedure invocations
- Test for n < 1
- combines two instructions and takes a set of add instructions
- They then increment the stack pointer to take those items off the stack
- Take a jr relative to $ra
- Take it back to the procedure that invoked that non-leaf procedure
- Begin the same set of instructions for another procedure frame
- Eventually it will hit the code and come back from that factorial
- It will be completely different from what we had and this should bring back our original value of n
- Add stack pointer to get rid of allocated amount of storage
- We would be able to return to the calling frame

- Local Data on the Stack
- Frame pointer holds the base of the frame
- Stack pointer holds the end as the stack continues to grow
- Part a shows the original frame
- Higher address at the top, lower address at the bottom
- This holds any argument registers that were saved
- If there are any saved registers that would be placed next, eventually we will get to the next stack pointer.
- Eventually when the frame is popped, the stack pointer will be restored as part of c.

2_6

- Memory Layout
- Ignore the “Reserved” segment
- Text is right above “Reserved"
- The next piece above that is global data (static data)
- Heap grows in the upward direction (malloc)
- Stack is used to store frames and grows downwards
- Important registers
- PC (Program counter) - pointing out address
- GP (global pointer) - point at something in the data side
- Stack pointer - point at something in the stack side
- Instructions are loaded into memory
- 32-bit Constants
- If we need to deal with 32-bit constants, it is a little more sophisticated
- Make use of the lui (load upper immediate) instruction
- Does NOT pull anything from memory
- Takes a constant (16 bit constant) and places it in the upper 16 bits of a register that is specified
- You have lui of the immediate 61
- $s0 is a 32-bit register
- Place the 61 in the upper portion of the register
- They do an ori (or immediate)
- OR $s0 to itself
- They are OR’ing something that has all 0’s in the lower immediate bits
- They are placing that value there
- This creates both halves of the 32-bit constant
- For any arithmetic operation, it is ready to go for that register
- Branch Addressing
- Conditional branch addressing
- Uses the I-format
- PC is a 32 bit register
- We only have 16 bits to specify things for bne or beq
- Most branches don’t go very far in the text segment of memory
- Can be forward or backward
- The way it is accomplished is the following equation

- Taken address would be the program counter of the branch + 4 since we have incremented the PC to the next instruction + whatever the offset is * 4
- All relative addressing would have considered the fact we are branching to the next instruction
- Do PC + 4, then add the offset * 4
- We know that with branch addressing, it will be 32-bit granularity
- Take advantage of that to compact the address base
- 18-bit address that would be added to a PC will have the lower two bits as zeros
- Shift left by two and think about the offset as stored as a word address.
- Do PC + 4 + offset * 4
- Jump Addressing
- Uses j-type format
- Has an opcode that has 6 bits like every other format
- Has a 26 bit immediate
- Much larger!
- With jump and link instructions, we don't need to specify registers
- Simpler to pack it into the actual instruction itself
- PC is 32 bits
- Loads immediate into the PC
- We have some missing bits
- One thing we do is the same trick for branching
- Going to be a word address
- Two lower bits of the address will always be 0
- We still have to take the upper 4 bits from somewhere
- Take it from the current PC and concatenate it to the lower 28 bits
- This is pseudo direct jump addressing
- Target Addressing Example
- See that some computation occurs
- Two branching instructions
- bne and jump
- The bne of $t0 and $s5 with label Exit
- Encoded as 16 bit immediate
- Displacement of PC + 4
- You see where things map into instruction memory
- addi is at 80016
- To get to 80024, add 8 to get to that location
- Stored as a word-address, not byte-address
- 2 left shift by 2 will give you the value of 8
- Jump has to store the entire value of the loop header (80000 in this case)
- If you are wondering how the assignment of load or exit occurs, it depends on the layout of memory
- Not implicit in actual assembly
- Branching Far Away
- Instead of doing beq, convert it into a bne
- Very close (2 instructions away)

- If you wanted to jump further, then you need to load it into a register first
- Addressing Mode Summary
- Immediate addressing
- Value is part of the instruction itself
- Level of indirection
- Look at 5 bit register specifier
- There is a base addressing where we use the location from a register base + an address stored
- Combine together to form an index into memory
- Instead of using a base address, you can use the program counter (PC)
- Pseudo-direct addressing can be used to get knowledge that we are accessing a word of memory at a time
- Concatenation - pseudo-direct address

3_1
Arithmetic for Computers

- ALU of the microprocessor: One of the major units for doing design space exploration
- Implementation alternatives and timing analysis of this particular unit.
- Operations on integers
- Add, subtract, multiply, divide, AND, OR, XOR, etc.
- Floating point real numbers
- Storing the values

Integer Addition

- Adding two positive numbers together: possible for overflow if result is out of range
- Adding +ve and -ve operands, no chance of overflow
- Overflow detection

Integer Subtraction

- Add negation of 2nd operand
- Overflow if result is out of range

Dealing with Overflow

- Language dependent on a handler
- C uses an unsigned instruction for overflow raising
- Java does NOT use this, so they make use of MIPS to handle overflow
- Ada and Fortran are other languages that require raising an exception

ALU Design

- We have an instruction fetch, where instructions are pulled out of memory
- Instruction decode figures out what kind of instruction it is and the operands required
- We are grabbing operands from register file
- Execution performs a computation

- Result store
- Next Instruction
- The outputs are the ultimate carry-out bits
- We want branch instructions to make use of the 0 output
- To a large extent, we will ignore this until we get to exceptions
- Result should be the same # of bits as a and b
- Some operations like a and b have more bits, but we will make the assumption that all operations will have the same # of bits for simplicity

One Bit ALU

- A single bit ALU performs a variety of operations on single-bit operands.
- Components
- AND gate
- OR gate
- 1-bit adder
- Multiplexer
- Output that is propagated to the multiplexer on the right-hand side
- All the operations are performed at the same time and they are available to the input of the multiplexer.
- All these operations are performing regardless of what is actually output
- There is a result on the right-hand side.
- Come back on how to put these things together to form the larger ALU
- Adder is covered in some earlier classes

One Bit Full Adder

- Also known as (3, 2) adder
- Has a carryin
- Half Adder has no carrying
- We have two outputs that carry out
- The table shows the inputs for a, b, and CarryIn as well as the output for CarryOut and Sum

CarryOut Logic Equation

- Reduces to sum of products form with three AND gates
- Calculate these in parallel and meets in a three-input OR gate
- Be careful of reducing the logic in the Boolean form
- CarryOut is a sum of products form with three AND gates

Sum Logic Equation

- All cases will be defined by an X input OR gate
- Think of this as a cascaded series of XOR's

3_2

- Construct a 32-bit ALU as a starting point made out of 1-bit ALU's
- Look at it through different configurations and having a Ripple Carry ALU

- Try to look at the initial binary addition and ripple the carry from each bit to the next.
- Each 1-bit ALU handles one of the places of the computation.
- ALU 0 handles the zeroth place: a_0 + b_)
- CarryOut of ALU 0 is connected to CarryIn of ALU 1
- This works in such a way that the latency of the overall set of ripple carry computations starts at the CarryIn and propagates down to the ALU 31
- 32 results on the right and they are all 1-bit results that make up the output.
- CarryOut coming out is NOT shown
- Operation belongs to each operation input and selects from the multiplexer for which particular output should be used.

Subtraction?

- Each of ALU needs an inverter for subtraction
- Take input b (value to be subtracted) and negate it (invert every single bit, then add 1)
- If we select 0, take original form of b, else if select 1, take the negative form of b
- We still have to add 1 and we do this by setting the CarryIn of 1 whenever we want to subtract

Overflow

- We need to check the most significant ALU
- Check the CarryIn[N-1] and compare the CarryOut[N-1] using an XOR gate

Zero Detection

- Determine whether or not the result coming out happened to be 0 or NOT
- One big NOR gate with 32 bit inputs
- Conceptually, this is the simplest result
- If any input is NOT 0, the result of the 0 bit would be 0

NOR

- Ainvert will flip the bit received by the a path and the same principle applies to Binvert with b

Set-On-Less-Than (SLT)

- Produces a 1 if rs < rt
- Defined as r-type instruction
- Only writes to register file
- Does NOT change PC
- Only two possible outputs
- 1
- 0
- Provide a large number of 0's then a 1 or a 0

- We are going to use subtraction
- rs - rt < 0
- This is a register indirect addressing: contents of register file at rs and rt
- Add input and output
- LESS
- SET

SLT Implementation

- The 3 input is received by the path Less
- If 3 is selected, Less will go straight through to result
- You won't be doing anything with the other outputs.
- You will have an additional output called Set on the Most Significant Bit
- Keep in mind that you perform the subtraction and the Most Significant Bit output comes from this result
- Set will be whether the value of the subtraction was positive or negative
- Sign bit in the two's complement form
- Assume we don't have overflow
- Take the sign bit and reroute it into the least significant bit's less
- This design is confusing
- Spans all the ALU's
- Uses SLT which students are less familiar with
- Set output comes out of result of the Adder, NOT the MUX
- The result of the Adder is subtracting two values
- Does NOT matter we are selecting two operations out of the result
- We are using the Adder as a way of determining if we have a positive or negative value
- Set of MSB is connected to Less of LSB
- Each bit position goes as input to the ALU's and we have all the results of before
- Operations propagated in the others
- We need a selector to see if the CarryIn of the entire chain is 1 or 0
- Shows that the set input which is coming out of the MSB is connected to the Less input of ALU0
- This should output when you select SLT
- Outputs 0 from every ALU result from either 1 or 0

Final ALU

- Bnegate has replaced Binvert
- Not only changes the MUX Binvert but is also tied to the CarryIn
- Determine if we use subtraction for the CarryIn
- Zero detector (NOR gate) determines whether we have 0 output
- Hardwired zeroes for the non-significant bits
- SLT is probably the most confusing, so understand the ripple carry addition
- SLT should take some time to trace the signals through
- Key thing to try to understand is thirty-one 0's and either 1 or 0

3_3

Can We Make a Faster Adder?

- Looking into making the delay for N-bit Ripple Carry smaller
- If you look at the right of this slide, you see a representation of 1-bit of an ALU
- Focus on the CarryOut portion
- Formed by sum of products representation of gates
- Inputs to AND gate are CarryIn, a, and b
- They are ready at time 0 because that is delivered from the register
- Computed by less significant bits
- For one of this set of this computation, it has to go through two gate delays
- Go through the delay of the AND gate and the delay of an OR gate to go from CarryIn and CarryOut
- 2N gate delays
- 2 gates per CarryOut
- Grows considerably as we look at a larger N value
- Look at Carry Lookahead Adders

Carry Look Ahead

- Try to generate a CarryIn ahead of time
- Use some logic on the side to trade area for performance
- See if it ripples through
- Start with a small Carry Look Ahead adder
- Larger adders will be a bit more clear
- Look at the table in the upper right hand side
- 3rd column is C-out based on values of A and B
- If A and B were both 0, regardless of what you get as C-in, you will always get a C-out of 0
- This is called a “kill”, which kills a carry
- If A and B are 1, you can ignore Cin, and you will “generate” a carry FOR SURE
- Cin is the delay that is difficult to handle in this case
- In that case, if that is the critical delay, we already know what the CarryOut will be.
- If A and B are not equal, we have to see the value of C-in
- C-in must be 1 in order to “propagate”
- We are going to augment each adder with two new signals
- G = A and B
- P = A xor B
- Generating means we will definitely have a carry
- Look at the larger figure on the left
- In coming in and because it is a 4 bit adder, it will be the initial carry.
- Smaller squares are all adders
- {A_0, B_0}, {A_1, B_1}, …

- We don't have any connection with the carry-out signal
- Signal on right does all of the carry-in signals.
- We see G and P outputs going from adder to carry lookahead.
- All GMP's are done in parallel -> no latency/dependency that exists
- The GMP's and carry-in's arrive in that box and they form the carry-outs.
- A_0 and B_0 should be one and we should "propagate" at 0.
- Propagate at 0th adder in this chain.
- If we generated at 0, we would NOT have been able to see it.
- C4 can be done on our own and figure out how it will propagate.
- Generate something at bit position 2 or generate at 1 at propagate at 1 and 2.
- We will do an analysis of the timing in class to think about how Carry Look Ahead works.

Partial Carry Lookahead Adder

- Make an even bigger adder
- 8-bit carry lookahead adder
- You would have all these C values being generated
- This is going to be output to the next carry lookahead adder
- Ripple carry of carry lookahead adders
- Do a time analysis of this in class
- Two alternatives of implementing the carry chain

Generate and Propagate

- Take the 4-bit CLA and add another lookahead to it
- The left part of the figure consists of four 1-bit adders with AND gates and XOR gates
- They are used to generate and propagate alpha
- Take four 4-bit CLA's and have another level of CLA logic
- Generate propagates here and figure out how to generate something with these four CLA's
- Generate one of the 4-bit adders and if you have the first one, you need to generate at 1, 2, and 3
- If you generate on the 2nd adder from the left, you propagate at 2 and 3 in order to see that generate
- Watch video for this! Hard to follow for this part.
- Propagation is a little more straightforward
- All four 1-bit adders have to propagate (AND them together)

Hierarchical CLA

- Each of these four carry-lookahead adders has an alpha, beta, gamma, and delta
- All equivalent designs and annotate the generates and propagates
- Each of these will have four 1-bit adders that contain carry lookahead logic.
- The overall 4-bit CLA will have this larger generate and propagate

- In terms of timing, all the generates and propagates would be done in parallel
- All of those would be created after the generates and propagates are ready.
- We will have a similar CLA and C4 which is the input would be either generated at alpha OR we would have a carry at 0 and a propagate for the 4-bit CLA.
- The form of these lookahead logics is similar to the 4-bit CLA
- Different generate and propagate are focused on hierarchical properties

Generate and Propagate

- The logic that is provided is in sum of products form
- We want to go through the timing of ripple carry vs partial carry
- Time permitting, we will do one more design which Reinmann will show on the next slide

Carry Select Adder

- Does actions in parallel to speed things up
- Takes this idea further and trades even more area for more parallelism
- On the left side, the adder is fixed with a CarryIn of 1, and on the right side, the adder is fixed with a CarryIn of 0
- Both are added to the 1-bit ALU and the computation is done in parallel
- Try both computations which can have only one of two values
- Take the output of both in terms of sum and CarryOut
- MUX at the bottom will let us select which output we will have overall as our Carry Select Adders.

3_4

- You have multiplicand that is multiplied by the multiplier.
- Each bit of the multiplier is analyzed from the least significant bit to the most significant bit
- When you will shift to the left when we write out the multiplicand
- It is a sequence of adds and shifts
- Perform an add on each multiplier digit
- Shift one to the left each time you perform a new add
- Summation that comes from each added value is the product
- In terms of multiplication hardware, it is the set of hardware that carries out this sequence of steps
- Multiplicand is capable of being shifted left
- Each individual bit will add a higher and higher form of the multiplicand
- Shifting left is the same as lifting it to another power of 2 (multiplying by 2)
- Over the course of multiplying by a 32-bit multiplier, we will have 32 possible shifts that can occur.
- Possible to occupy all 64-bits assuming both the multiplier and multiplicand are 32-bits
- Running sum will accumulate as we go and we will have to shift off the least significant bit using this multiplication approach.

- Multiplicand holds the actual multiplicand itself. The product will develop over the course of many summations.

Multiplication Hardware

- You are supposed to place the result into the product register that an addition should be performed and the ALU writes its result into the product register
- Control test throttles whether the product register writes.
- There is a control test to see if the control test is equal to 0.
- There are two steps that occur regardless of whether the LSB is 1 or 0
- Shift the multiplicand left by 1 bit
- Same idea as the example of 8 * 9 where the potential multiplicand would have been shifted left for each cycle
- Go in order from least significant bit to most significant bit.

Optimized Multiplier

- Think about ways of potentially optimizing things
- 64-bit ALU
- Increases in orders of magnitude by left shifting
- Instead of using a 64-bit ALU computation, we want to sue a 32-bit ALU.
- The multiplicand in this example is NOT shifted, but this is going to make an equivalent functional result, but it will be done in a mores sophisticated way.
- The product register remains at 64 bits.
- We can optimize by adding into the upper bits in the product, and shift that to the right.
- The multiplicand increases in order of magnitude and we can add to the most significant bits of the product and shift those back down
- Multiplier is loaded into the lower 32 bits and the upper 32 bits are zeroed out
- As we shift to the right, we are stripping off the individual bits of the multiplier and we are using those for the control test.
- Control test only has to throttle if we write the product register or not.
- We would always shift right after each check of the least significant bit.

MIPS Multiplication

- Two 32-bit registers for the product
- HI: Holds the most-significant 32 bits
- LO: Holds the least-significant 32-bits
- Instructions
- There are some like the mul, which have an explicit destination (rd)
- Write the least-significant 32 bit of product
- Not as precise but good for modulo operations
- HI, LO can be a bottleneck if you are performing a lot of different multiplies for one register

3_5
Signed Multiplication?

- Make both positive
- Even if you have -5 * 5, assume both are positive
- Complement the product when you are finished
- The other approach is Booth's Algorithm
- The idea is to replace a sequence of additions with a subtraction and an addition
- ALU with add or subtract gets same result in more than one way
- 14 = 2 + 4 + 8
- Booth's Algorithm reduces the sequence of adds
- Multiplier would be used to know how many times to add the multiplicand to the running sum

Booth's Algorithm

- When we see 10, this is the beginning of a run of 1's
- We do this regardless of whether or not it is a single 1 or multiple 1's
- If we see 01, this is the end of the run of 1's
- Get four 1's in a row and the current bit is a 0
- Multiplicand is one higher than it was in the original implementation
- We are at the 0 instead of a 1 in the multiplier
- Effectively gives us one order higher and come up with the same result
- For 11, we aren't doing anything in terms of operations except shifting
- Show a few examples and see how it works for signed multiplication

4_1
The Processor

- Two main implementations of the data path i.e. single-cycle implementation
- Have every cycle take a single cycle
- Relatively low CPI
- We will have a cycle time that is determined by the longest latency instruction.
- CPI will potentially be positive
- Assume that we implement these versions of a single cycle version and a pipelined version
- Assume a simple subset of MIPS overall
- Memory reference: lw, sw
- Arithmetic/logical: add, sub, and, or, slt
- Control transfer: beq, j
- Question: Add a 10th instruction without breaking remaining 9 instructions
- That's what he can ask on a test
- Instruction Execution
- We have to provide a sequence of instructions by the actual hardware
- There is a program counter that is a pointer to the next instruction to execute

- Fetch an instruction, and once we have the instruction, let's figure out what it is and read out components in the register file
- Depending on the instruction class, we will do different things
- ALU can be used to calculate the following
- Arithmetic result
- Memory address for load/store
- Branch target address
- Arithmetic results will be written back to the register file
- This depends on the instruction that is given
- CPU Overview
- The connections here are not correct
- Broad overview of what is happening
- Combining wires without any types of multiplexers
- Get the big stages
- From the left of the figure, look at the PC (32 bit register)
- Holds the address of the current instruction
- Square on the left that has an address port and the address is used to specify what word from memory that we want to grab.
- Don't worry about what the Instruction memory actually is, it will just dump things out from that address
- Usually cached as opposed to porting from memory
- We will have a port to access it and instructions will come out.
- Register files are embedded and we can read them off.
- Most instructions have a maximum of three registers
- These specifiers indicate which we will use from the register file.
- There is an ALU where we get things from the register file.
- Line that comes straight out of the instruction called an immediate.
- ALU performs a calculation that goes two places.
- Into the register port and the address port of Data memory
- Data memory is used for loads and stores
- Address port specifies what address we are accessing in Data memory
- In a load, data will flow out of the right-hand side of data memory
- A store will grab something from the register file and write it into memory
- The first adder is for incrementing PC + 4 (this is the one with the 4)
- The other adder is for effective adder calculation and adds this to the PC + 4
- What is missing from all of this is the Control
- We need to get a feel for the big pieces altogether.
- These 5 components together make up a large part of the individual pipeline.

Multiplexers

- The last figure has some serious problems
- You cannot just have them meet and it will come miraculously into 32-bits.
- Make multiplexers here.

Control

- Controlled by a blue oval at the bottom
- Sets up what happens across the data path.
- The components shown in black are the data path
- Data flowing through lots of different choices
- Blue part decides what gets done at each different level
- Lots of MUX's that are being added here.
- This allows us to select what operations are being performed.

Logic Design Basics

- Not a lot of low-level stuff in M51A, but talk about what is of use to understand how the data path actually works.
- The logic design basics focus on low voltage = 0, high voltage = 1
- Individual wires sustain a single bit
- Multi-wire busses will be supported with a slash and they propagate on that particular wire
- We won't draw multiple wires unless we have to.
- The overall data path will be composed of combinational elements
- State (sequential) elements store information in boundaries

Combinational Elements

- AND-gate
- Y = A & B
- Multiplexer
- Y = S ? I1 : I0
- No storage inside, output changes as input changes

Sequential Elements

- Store data in the circuit
- Triggered typically by a clock
- Flip-flops
- Input D is stored in the actual device until the input changes internally
- Tiny diagram shown at the right, and the clock is delineated by dotted lines
- Arrives by some value that stabilizes and when the edge triggers, it propagates into Q as the output
- This is used as our boundary
- Single-cycle data path
- All the processing has to occur before the next clock tick occurs.
- Latch results in the register file and we have to slow our clock considerably to sustain a single cycle data path.
- The amount of work we have to do has to stabilize at the output ports.
- Write enabled port
- Additional input called Write
- Throttles when we do or do NOT write
- With a write enabled
- Q will only output when Write Enabled is 1 (High)

- Write enabled guides whether or not the register will be written at the end.
- The program counter for now will be unconditionally (always) written for simplicity

Clocking Methodology

- State elements bounding combinational logic and the combinational logic feeds back to the state elements
- Overall scheme is the same and we have to figure out the worst case delay.
- The state element holds data before and after that combinational logic.

Datapath Notes

- PC: 32-bit register that holds the address of the current instruction.
- Goes into instruction memory block that has an address inport and instruction outport.
- Register: register specifiers may be read here.
- Most of our instructions that we deal with have a max of 3 registers
- R-type
- rs, rt, rd
- Indicate which registers we will use for the register file
- ALU: two inputs from register file
- One from R[rs] and the other from R[rt]
- Immediate that short cuts coming from the bottom (from instruction memory)
- Performs calculations to two places
- Data import of register file
- Address port of data memory
- The address port on data memory specifies which address we will access in data memory
- Data memory: Has an address and data inport
- Load: data will come into input port of register file.
- Store grabs it from register file and puts it in the data port
- 4 is hardwired to be used for PC + 4
- Control: sets up what happens across the data path.
- This is data flowing through lots of different choices through lots of different connections.
- Lots of control and MUX's being added to here.
- Keep in mind you have a data path and a control path that lets you select what data you want to perform

4_2 Building a Datapath

- Set of elements that process data and addresses and perform heavy lifting and the work.
- See what operations are actually performed
- Large and cumbersome
- Understand it piece-by-piece.

- Instruction Fetch
- There is a 32-bit register called the PC, which provides an address into Instruction Memory
- Has a port for Read address
- Has a port for the Instruction coming out
- As soon as it has a Read address, it will provide the port on the output.
- Not a clocked component where this is a read at the end of the cycle.
- Everything happening here is from the same cycle.
- There is a line coming to the adder used as one of the input operands where PC is incremented by 4.
- PC + 4 will come out of he Adder and placed back into PC
- Instruction address comes out of PC
- PC is used to increment by 4 to get the next address

R-Format Instructions

- Provide two register operands as inputs
- Come from rs and rt fields
- Register file components -> improved resolutions from previous figures.
- Those take 5-bit inputs that correspond to pick one of the 32-bit registers.
- Place 5 bit value in Read register 1 and it is driving out the Read data 1 stuff.
- Assumption is you put data out on the ports and it is driven out.
- Address of the stuff that comes out, and Read register 2 comes out and there is no signal here for reading.
- Put data out.
- RegWrite indicates if you should write a register or not.
- There is no such thing as an empty line
- This data represents a value.
- All of these are values
- An additional piece of control logic indicates whether or not a write should go through
- Tells if the instruction should write to the register file or not.
- 4 bit ALU control line: There is a large MUX that can select what should be coming out of that MUX

Load/Store Instructions

- We need some additional components
- Grab register operands
- Store will also require RT
- Take the immediate and sign-extend it to 32 bits.
- The oval that says sign extend will take the sign extension unit and replicates 16 copies of that input to make 32 bit output.
- Data memory will have ports and an address port where we will do reading or writing.
- Two control lines like a MemWrite and MemRead

- Why do we need a control line for MemRead?
- What is it about data memory that would require us to have a MemRead control line?

Branch Instructions

- Dump out two register files
- rs
- rt
- Compare them using the ALU and check the Zero output to determine if two register operands are equal
- Use that Zero output to conditionally set the PC
- Set it to PC = PC + 4 or create a new address using SES(I) to PC + 4
- Have an additional adder that takes in as input bits for the instruction, and those 16 bits would go into a sign extended logic at the bottom
- Put this into the shifter near the adder.
- This provides a SES(I) that we can use
- The adder will have a sign extended and shifted value along with something from the fetch data path.
- The output from the adder should be the branch target.
- The output we are concerned with will be the 0

Composing the Elements

- The key thing is how should we combine the components together and we shouldn't be reading or writing at the same time.
- Register file can only read from two.
- One cycle can only do one function at a time.
- Read or write something from data memory
- Different data sources will be used for a single port.
- We also need an immediate in this case.

R-Type/Load/Store Datapath

- We have instruction coming from left and this would have had the instruction fetch already done.
- This goes into the register file for rs, rt, and rd inputs
- Immediate portion gets sign extended and the instruction itself will have bits which for the R-type would be the shift amount and the sign extension unit.
- There will be a 32 bit value sitting on the MUX that we wouldn't use.
- There will be something that goes to that port, and the control has to indicate whether or not it is useful.
- Registers drive out Read data 1 and Read data 2
- Read data 1 always goes into ALU
- Read data 2 and sign extend go into a MUX determined by ALUSrc, which indicates if we use it as an input to ALU
- We will have MemWrite or MemRead asserted.
- The Read data that comes out of the memory will appear at the MemtoReg MUX, and only cases where we write to register files will it be enabled.

4_3

- Get to the full data path and start looking at CTRL
- Extending it with the jump instruction
- Major modules are the PC which will be incremented by 4
- Bits of the instruction itself are used as inputs to the register file
- Registers themselves come out and are used as inputs to the ALU
- SE output is a potential input to the ALU and is part of the shifter logic at the top
- This result is put into another MUX to determine if we want to use PC + 4 or PC + 4 + SES(I)
- This can be one of the potential inputs to the data file.
- We are going to go through and see how we can control this.

ALU Control

- Used as a separate controller used to select what ALU does in the context of one instruction
- Various functions that the ALU performs
- Load/store will require the function to be an add
- We have to do an effective address calculation
- Branch instructions like beq or bne
- Compare to register inputs using subtraction
- R-type
- F depends on the funct field that are possible out of the ALU
- Based on the instruction type, we want to be able to add or subtract
- 6 bits will have the ALU operation depend on that
- Build up an ALU controller by looking at potential outputs first
- ALU will have 6 functions it can perform
- AND
- OR
- add
- subtract
- set-on-less-than
- NOR
- In our simplified set, the ALU Control will be a pretty sparse table.
- Control will have two inputs: type of instruction we are performing
- ALUOp will have a 2-bit input and have an opcode that goes into the main controller
- This decides which ALUOp we are going to output.
- Indicative of a load, and we always want the function to be an ADD.
- If it is a beq instruction and we want to performa subtraction, we will use an ALU Op of 0110
- In the case of the ALU Op being 00 or 01, we will be performing an dadd or subtract in that case for the first two bits
- Bottom will show the function field that shows add, subtract, AND, OR, or slt

- Why do we have two add's?
- Talked about in class
- The Main Control Unit
- Figure out what to do for these two data paths.
- They all have an opcode first in the upper 6 bit
- All R-types have the same opcode
- 4 opcodes we can possibly get
- The bits 25:21 are always used to specify rs
- We are always reading out rs register as our read register 1
- In the case of load and store, it will be the written output.
- In terms of destinations, the R-type will use rd to specify its destination register.

Datapath With Controller

- There are a lot of wires coming out of main control and these individual control lines will indicate what we will change depending on a particular instruction.
- We take this slide and without all the values for the control table we are going to build up is to figure out what the outputs of the controllers will be.
- Try this on my own later (9:20)
- R-type doesn't make use of memory at all so it neither reads nor writes memory
- Use rd as destination register and set RegDst as the value 1
- This would be representing rd
- The branch one would be used to throttle whether we look at the 0 bit as a means of setting the PC
- Use PC + 4 and to affect that, the MUX at the upper right should be used with 0.
- Use the AND gate as an input to the MUX and set that to 0
- Even if the ALU represented in a 0 output, we would still not change the PC to some bogus value.
- MemRead is going to be a 0 in this case
- MemtoReg would be set to 0 since we take the ALU input
- ALUOp will be set to 10 since we want the ALU control to use the function field.
- MemWrite is set to 0 since we are NOT writing to memory
- ALUSrc will be set to 0 since we don't want to make use of the immediate
- RegWrite will be set to 1 because we are going to write to the register file

Load Instructions

- RegDst set to rt in which case those instruction bits are 20:16.
- Write to register rt at the end
- Branch set to 0 since we don't want to change it from PC + 4 + SES(I)
- MemRead set to 1 because we are reading from memory
- MemtoReg set to 1 because we take value from memory
- ALUOp set to 00 since we are forcing an add
- MemWrite set to 0 because we aren't writing

- ALUSrc set to 1 because the source comes from SE(I)
- RegWrite will be 1 since we are writing to the register file

Branch-on-Equal Instructions

- We want to subtract rs and rt, and use Zero bit to indicate whether or not to branch
- RegDst is don't care
- Branch is set to 1 since we want it to be conditioned by 0 bit
- MemRead is set to 0
- MemtoRead is don't care because we don't write to register file
- ALUOp is 01 to force a subtraction
- MemWrite is 0 because we don't write to memory
- ALUSrc is set to 0 because we want to make sure we are subtracting two registers and not an immediate
- RegWrite is set to 0

General observations

- When we have RegWrite set to 0 and we aren't writing to a register file, it doesn't matter which register we are using.
- When we don't use MemRead or MemWrite, it wouldn't matter
- We have data flowing from sign extension unit to shift left unit
- Only instruction that makes uses of PC + 4 SES(I)

Controller

- Outputs of main control unit
- Columns show 4 different instructions we can have
- These slides didn't go through store words (go over this on your own!)
- All the control bits are shown here
- The Opcodes uniquely specify instructions
- Left hand side
- Physical implementation (not needed to know)
- We need to know table level implementation
- Extend table to right with additional instruction

Implementing Jumps

- Uses J-type field that uses an op-code with 26 bit field
- Jump has pseudo direct addressing
- Two 00's at the bottom
- We are going to augment the data path without breaking it for the other instruction

Data path With Jumps Added

- New line that goes straight up to a new left shifter
- Carries instruction bits from 25:0
- They are appended to another MUX that has been added to the right hand side.

- Cascaded from the original MUX and we are going to add additional control inputs and we are fine with this cascading notion.
- When it comes to extending the data path, make it more clear and easy to read.
- Show clarity for extending the data path.
- Needs to perform well and make it easy to read.
- We need to have RegDst set to Don't Care
- MemWrite set to 0
- ALUSrc set to Don't Care
- RegWrite set to 0
- Cycle time needs to be long enough and we will see a larger disparity between what individual instructions want to take.
- All of this is for a processor where we don't see a single cycle processor with longer operations

4_4

- Pipelining is the next session we cover

Performance Issues

- Longest delay through the circuit for any instruction through that data path.
- For the conventional set of instructions, the critical path is derived from the load instruction.
- We can access gate memory and write to the register file in this manner
- Not feasible to vary clock period for different instructions.
- We only have a simple setup so far but we want to make the common case fast.
- Instead of going through a single cycle, we will use pipelining
- Reduce CPI by as much as possible

Pipelining Analogy

- Laundry analogy
- Imagine you have a # of loads you want to use and you have four or more loads of laundry.
- You use a washing machine, a dryer, then doing folding, and then putting away dry clothes.
- There are 4 steps you have to perform, and pipelining describes these as discrete steps that you have to perform.
- Stack of folded clothes is the process of folding and you'll notice on the vertical axis that there is a task order: A, B, C, D
- The horizontal axis has a time from 6 PM to 2 AM
- Naive approach: Put load A first, and assuming each takes 30 minutes, you would put them back to back. 6 to 6:30 for A, 6:30 to 7 for B, 7 to 7:30 for C, 7:30 to 8 for D
- Each load would take 2 hours with NO overlap
- We can overlap if we only look at the structure!
- Each stage takes on different resources

- Different kind of parallelism than parallel programming
- You overlap different resources working on different tasks
- Every stage of the pipeline is doing some work.
- This is called the steady state.
- A new piece of laundry will be finished every 30 minutes
- Every stage delay costs 1 unit of work
- Asymptotic speedup is 4 in this case.

MIPS Pipeline

- Five stages, one step per stage

0. IF: Instruction fetch grabs the instruction from memory
0. ID: Instruction decode where register read occurs
0. EX: Calculates effective address
0. MEM: Access memory for load and store
0. WB: Write back to a register file

- There is a difference on which part is used for different resources.
- Slice each data path for each individual resource
- Book has a nice way of sharing this information a bit later.
- Instruction decode is the register file and memory is the data memory.

MIPS Pipelined Datapath

- Blue line circling back indicates write back
- Actual write of the PC is done in memory
- Memory stage is where the PC gets exchanged for bne
- There are individual slices and we break it up into 5 discrete stages.
- We will have different instructions at different stages
- The steady state that we want to be in will involve every stage having this instruction

Execution in a Pipelined Datapath

- Load instructions will all be independent
- Very idealistic set of instructions
- This kind of diagram is a pipelined diagram
- Horizontal axis has CC1, CC2, …
- Represents Clock Cycle
- Vertical axis has 5 load word instructions
- lw, etc.
- Goes through all 5 stages of the pipeline.
- Label by acronyms on previous slide OR use larger structures that compose that stage
- Reinman prefers IF, ID, EX, MEM, and WB (used in class)
- The important thing is to be able to assign what stage the instruction is at any point in time.
- Later on, we can find what hazards we might encounter.
- No hazard == no problem!
- No type of delay besides going through individual stages.

- In CC4, Instruction 1 will be in the memory stage
- The 2nd load instruction starts in CC2 and overlaps with the original instruction, which would be in Instruction Decode (ID)
- Figure out which instructions are in what stages of the pipeline
- By CC3, there are three instructions all the way up to CC5.
- Reach steady state by CC5. This is done in an N-stage pipeline after N-1 cycles.
- The main idea is that the more instructions we have, the more latency would overlap and the more we can drive CPI down closer to 1.
- Goal is to try to get CPI as close to 1 as we possibly can

4_5

Pipeline Performance

- Goal of a pipeline is to reduce Clock Cycle Time
- Assuming we don't have hazards, let's look at our performance in cycle time.
- Assume for any stage, the latency will be either 100 ps or 200 ps.
- We will have to go through 200 ps of Instr fetch, 100 ps of Register read, etc.
- This is a total of 800 ps for latency of lw
- The tables below show that the instructions are faster because they either don't access memory or don't write
- Single-cycle:
- Our cycle time would need to be 800 ps for this to work.
- Pipelined:
- The stages themselves would set the cycle time.
- Looking at the figure at the bottom, the first lw would enter its instruction fetch and we would need 200 ps for the next Instruction fetch to come along.
- These different load words are going to move separately from one another.
- The maximal latency will set the clock cycle time.
- The longest latency of any stage is 200 ps
- Maximal stage latency to complete to ensure that we can move to the next stage
- The cycle time here is 200 ps, a quarter of what we have in the single cycle case.
- Pay for that with complexity in the following slides

Pipeline Speedup

- If we want to try to figure out pipeline speedup, we can take a single cycle data path and see more speedup if all stages are balanced.
- To get the full speedup, we should have all the stages as balanced as possible
- There was not completely even balancing across all the different stages.
- The overall time we saw dropped was only by a factor of 4 instead of the 5 stages we saw.

- As we look at deeper pipelines, it will become more difficult to speed things up into different stages.

Mixed Instructions in the Pipeline

- Add instruction doesn't need to go through data memory.
- Put instructions only in the stages it needs to go in.
- The problem is that if you look at this data path and the pipeline diagram on this slide, the load and add instructions reach the same stage in CC5
- Both finish at the same time and are ready to write
- Collision!
- In Clock Cycle 5, both the load and the add want to use it, so it's not going to work.
- Need to regularize the path
- Add instructions cannot skip data memory, and this might lead to having two instructions at once.
- We can't do this without increasing our resources.

Pipelining and ISA Design

- MIPS is specialized for pipelining, all instructions are the same length (fixed length)
- Not that man instruction formats, so we can decode and read registers in one single step
- Other machines like ARM will take many more stages because you increase the depth of the pipeline
- Load/store addressing becomes similar
- Alignment is relatively simple to force memory accesses to take a certain cycle

Pipeline Principles

- We can force any instructions that share a pipeline to have the same stages in the same order.
- It cannot do anything in the MEM stage but it has to be a placeholder for the add.
- We need some way to ensure that the instructions that are one after the other in the pipeline don't interfere with each other's signal
- Pipelining impedes functional blocker use.
- We don't want to have a situation where 2 instructions try to access the same resource at the same time.

Pipeline registers

- Add latches in between the stages of a pipeline
- This is the single-cycle data path we have been looking at in many times and divide it up into rectangular blocks
- Banks of registers or flip-flops where data is latched or stored.
- There is a bank of registers after the instruction fetch stage.
- The first rectangular box is greyed in with IF/ID

- Register latch that separates Instruction Fetch (IF) from Instruction Decode (ID)
- Once you access PC to get access in the front end of the pipeline, you can use this to drive out the instruction and send it into that latch.
- You will have latched that instruction in the ID/IF latch
- In the 2nd cycle, you can read out PC + 4 and latch it into ID/IX
- Then you will read out register values that will be latched out.
- We will also be looking at the next instruction and grabbing it from instruction memory.
- You have instructions that are moving together and saving intermediate results in latches that exist in the pipe stages.

4_6
Pipeline Operation

- Operation of the pipeline on set of instructions independent of one another
- Use the pipelined diagram from the single cycle data path and try to see how these would work for load and store instructions

IF for Load, Store, …

- Load word goes through instruction fetch in the first cycle of execution
- We will be talking about the load word across many cycles
- Reads out the PC of the load instruction, grabs the instruction itself from memory, and place it in the IF/ID latch
- Increment PC by 4 and place PC + 4 into IF/ID latch
- Value of PC will be written into the actual PC itself
- This can later cause a problem in instruction MEM
- Let's assume there are NO instructions that can change PC for now
- 2nd cycle
- Goes through ID and reads out of IF/ID latches to grab register data from the register file and latch them.
- The load doesn't use RT
- The 16-bit immediate will be sign-extended and we will propagate PC + 4 into the ID stage as well.
- We need PC + 4 and the register file value at register RS and register RT
- The 32-bit sign extended immediate will come from the load instruction itself.

EX for Load

- Adds it to a left-shifted form of the immediate and placing it into the EX MEM
- The Load word instructions grabs the register value and this value is going to be propagated from the ALU
- The MUX is going to be grabbing the sign extended immediate in ID/EX
- Since this is a load instruction, we will add both operands into the ALU together.

MEM for Load

- Place it at data memory into the address port
- Take the data and place it into the MEM WB latch
- Notice the value we computed for the ALU is also being latched at the MEM WB latch

WB for Load

- Load takes the value from memory and uses that as the input to the write data port of the register file
- On the write register port, there is an error (look for it)
- It has a wrong register number!
- The intermediate data is stored in the IF/ID latch
- The load word is currently in the MEM WB latch
- We didn't propagate the right register specifier
- Anything you want to use in later stages has to be propagated in those instructions

Corrected Datapath for Load

- We can use register specifier bits and indicate which register we want to write to.
- Those 5 bits are going to be propagated into the ID/EX stage
- These will be propagated through the ALU stage and latched into EX MEM latch
- Propagate through data memory stage and eventually they make their way into the right register port.
- The data needs to reach the write data port, and these need to be paired together down the pipeline.
- Think about why this loop would have to occur and these instructions go through one stage at a time.

EX for Store

- Things are the same for the IF/ID stage as they are for loads
- After execution, the slides show Execution, but there is no difference for the Execution stage.
- Should have been stored in the EX/MEM latch

MEM for Store

- Things actually change here!
- Write data into memory
- Change R[RT] and place this into the right data port.
- Those together along with the MEMWRITE signal should let us write to memory
- The key point is in the next cycle, we aren't going to be making use of those values at all.

4_7

Multi-Cycle Pipeline Diagram

- This form shows resource usage so it's easier to see things internally
- ALU is NOT the easiest figure to draw and it shows on the horizontal axis that we have time
- Vertical axis is program execution order
- The load word would go through the instruction memory in cycle 1
- ALU access in cycle 3 and data memory access in cycle 5
- When we have 1 instruction per stage with all instructions being occupied.
- This diagram is useful when we start encountering hazards because we can see when things are no longer dependent on the other.
- Traditional form
- Use abbreviations like IF/ID/EX, etc.
- Shows the name of the individual stages and the horizontal and vertical names.
- Make sure to use ones that make the most sense to discern what is going on in the pipeline.

Single-Cycle Pipeline Diagram

- Show the full pipeline diagram along with instructions on the horizontal axis
- The oldest instruction would be the one in write back and this is useful for trying to reason where instructions are in the pipe.
- Cumbersome when trying to draw the interaction of a lot of different instructions

Pipelined Control (Simplified)

- Identical to what we had for single-cycle data path
- We still have RegDst and so forth.
- We have multiple instructions for the pipeline so there isn't one set of control signals at any point in time.
- Separate those so we don't confuse instructions
- Control signals are read in the IF stage
- IF/ID goes through the same main control unit which dumps out all the control signals
- Save the signals inside latches
- Notice they separate them out into EX/M/WB
- The control signals will be used within the execution stage.
- Then, the control signals for memory and write back will be propagated straight to the EX/MEM latch.
- Full diagram that shows the control signals and latches integrated
- Useful for instructions that are currently in the execution stage and will be stored in the ID/EX latch.
- By insulating the instructions from each other, it is similar to how we were insulating them from one another in terms of their data.

Pipelined Control Signals

- We just cluster the by whether they are execution stage control lines, memory stage control lines, etc.
- Different from the original way of organizing the table but essentially the same thing.

4_8

Pipeline Hazards

- We are starting to see complications that pipelines can have

Hazards

- Performance will drop and we will see CPI > 1.0 in the ideal case because of conditions where you either have incorrect data, incorrect control, or resource utilization that won't work out.
- Start with just data hazards
- In this sequence of instructions, there are initial conditions for any register i.e. #'s for these registers
- For any arbitrary register #, this is just an arbitrary initialization for the purposes of this example.
- Just setting up initial conditions.
- Takes the register values from $10 and $11, and place it in register $3
- lw will use $3 as it's base address, and it should load memory into location 92 (42 + 50)
- add will sum 20 and 22 in this case, and this result 42 goes into register $3
- sub will remove 14 which is in $7

The Pipeline in Execution

- add will make its way into the ID stage and it does read out in blue the values 20 and 22 just like it should.
- We have gotten the correct values and load word has been pulled out of memory
- subtraction gets fetched from memory, and each instruction depends on the next.
- lw is reading its registers in this cycle, and add has not even completed the calculation because it is finding the value.
- Register $3 has not been written
- It is reading an incorrect data value because the value has NOT yet been written

Hazards

- Situations that prevent starting the next instruction in the next cycle
- The three hazards are the following:
- Structural
- Two instructions need to use the same resource

- If we had only a single port to data memory, they might have to wait
- Data
- Wait for a previous instruction to finish its data read/write
- Control
- Wait on a branch to resolve (either conditional or unconditional)
- You need to know what the PC should be

Structure Hazards

- Each instruction is in one and only one stage
- Loads and adds go into one stage, so we might have a hazard where two hazards might end up in the same pipeline at the same time.
- When the load or store wants to access data memory, you will either have a stall or an instruction fetch.
- Pipelines should move from one stage to the next
- If you have to stall the pipeline stage, the other thing would continue to move, so there might be a gap where there is no instruction.
- The further away we will get from a CPI of 1.

Data Hazards

- More complicated hazard
- Have a # of different solutions
- Try to better understand the problem first
- Slightly more simple example
- We need in the worst case to stall
- We would inject bubbles so the add would have to proceed through so yo u can perform the next instruction.
- The implication from add and subtract dependency on the next slide.

More sophisticated example

- This will illustrate the full potential of the problem
- subtract instruction that writes to $2
- The subsequent instructions all use register $2
- The instructions are shown on the Y-axis and the oldest is the sub instruction and the youngest is the sw
- The X-axis has the clock cycles
- Subtract goes through its stages and the register is written at the beginning of cycle 5 and this is shown in the greyed in region at CC5
- This is called a **transparent latch**
- Allows us to read the same register value in the 2nd half
- The implication is that in the 2nd half of a cycle, you can read, and you can write on the 1st half of the cycle
- More efficient
- $2 is available in CC5, but we need it in CC3 for the AND instruction
- sw is the last instruction, and it gets the register file in the following cycle
- Transparent latch
- add instruction can be able to write in the 1st half of CC5 and read it in the 2nd half of CC5

- The “and” and the “or” have problems
- We will have to do something else to satisfy the hazards for those instructions

4_9
How to cope with data hazards

Dealing with Data Hazards
- Software
- Rearrange instructions and insert no-ops if necessary
- Hardware
- Stall the pipeline
- Perform data forwarding

Dealing with Data Hazards in Software
- Expect that there are no data hazards because they will be handled in software
- Reduces the portability (backwards compatibility) of the software
- In the instruction pipeline, we have to cycles we have to be careful about.
- The software would have to be aware of that and it should be a favorable design point.
- We don’t have to add any complexity to the hardware, and it is much easier to add software complexity
- Less validation in software efforts, but it is much easier to patch software than it is hardware.
- Subtract followed by an add
- Same example but we would either find separate instructions or place no-ops so subtract and add can work correctly.
- We can inject no-ops in this case
- No-ops basically do nothing
- Shifting left by 0 to register 0 for example
- Doesn’t do anything in terms of functionality but it takes up space when necessary
- Set things up so that add reads its registers at the same stage that subtract uses its registers
- This lets us keep our CPI close to 1 because we pump out something for each cycle
- The CPI will hopefully remain close to 1, CT won’t be affected, but Instruction Count goes up!
- There is a tradeoff in performance as well as portability.
- Key tradeoffs
- **Instruction count goes up**
- **Portability is limited**

Code Scheduling to Avoid Stalls
- Fill the gap that we place with no-ops with other instructions instead.

- Compiler can do a fairly good job at this for a smaller scale
- Hard to move loads above stores
- Difficult to move instructions around branches
- In general there is some optimization that can be done
- Takes 13 cycles because we have to add stalls for the dependencies that occur
- Register communication using $t2 and $t4 respectively.
- Instead of placing instructions one after the other, we can rearrange it to reduce dependencies.
- Still will have one stall in the new piece of code, but we wouldn't have the value for the add, so we need some form of stall to fix that.

Where are No-ops needed?

- In this case, we have a subtract writing to register $2, an "and" that consumes it, an "or" that consumes it, etc.
- Do a pipeline diagram and figure out where you need to inject no-ops.
- You would have to out an additional no-op and have to have the no-ops added if we did no-op injections.
- Are our no-ops really necessary here?
- We could rewrite this code and try to avoid the no-ops entirely.
- There is no way of avoiding at least one no-op here.
- Instead of having that first no-op, we have an additional slt.

Handling Data Hazards in Hardware

- Stalling the pipeline is like injecting no-ops, but you just do it in the hardware rather than doing it in the software.
- When the add reaches a stage where it cannot get the register value, it will wait in that stage.
- We can independently control these different pipe stages
- The pipe stages that are after it may continue executing and NOT allow the execution in the pipe stage to move anymore.
- Avoid pileup where multiple instructions reach the pipeline together.
- The Bubble keeps anything else from moving forward in the pipeline, and the subtract will continue to execute until CC5
- We will talk about the hardware in a bit, but we want to inject the two bubbles which is analogous to stalling the add in two cycles.

Compare stalling in hardware to no-ops in software

- To be discussed in class notes

Handling Data Hazards in Hardware

- Add instruction that doesn't use the register, and a dependency in the or
- The other dependency that exists is the 2nd add instruction ($14) which places it into memory
- The adds to register $12 has a dependency on the add in register $14
- Dependencies are shown in the colors

- Bubbles are stalls
- The bubbles carry downward to subsequent instructions
- Stalling instruction 1 will subsequently stall instruction 2 all the way to instruction N

Pipeline Stalls

- How we actually affect the stall
- Detect that the hazard exists
- Prevents it from being correctly handled and let's make the stall happen
- Keep the instruction fetch and instruction decode stages from making forward progress
- ID stage is the last place we can read the register value.

The Pipeline

- The first thing to think about is how do we know we have to stall?
- The places the instructions can be:
- Some instruction A that is in the execution stage
- Computing a value A and it is consuming that value
- We want the instruction in instruction encode stage to stall and how do we know this situation even happened
- Instruction A has in its execution stage the register specifier that it is going to write to.
- The MUX that is in the execution stage that is controlled by REG DST will provide a register specifier to determine where A is going to write.
- We need to compare this against the src operands for register B
- In addition, we have to make sure register A is even writing to register B
- Check what time of instruction that register B is

Stalling the Pipeline

- In terms of affecting a stall, we have to make sure we don't write the PC
- Add an additional control signal so we don't write to PC
- Choose not to write to the PC an don't rewrite to IF/ID register
- Instead of passing control signals from the decoded instruction, we want to pass control signals that are effectively all zero.
- nops will have zero effect on the pipeline at all.
- Think about outputting all 0's to the control.

4_10

Forwarding (aka Bypassing)

- Last solution
- Intuition:
- Even though we may not have written to the register file, it might be ready at an earlier time
- Add instruction: add the register value to a 3rd register, and it should be ready at the execution stage
- Ready a full two cycles earlier

- There is an add instruction and a subtract instruction that reads it as the next instruction
- No other mechanisms for handling the hazard, we would have to have a mechanism to handle two style stalls.
- The register value that we are writing to $s0 is ready and can be communicated to the following cycle.
- It can be communicated to the subtract by the beginning of the 4th cycle
- We don't have to wait at all if we can allow this communication to occur.

Data Hazards in ALU Instructions

- Followed by 4 dependent instructions
- In this same sequence, we can satisfy all the requirements by using forwarding
- This would be available to any other instructions if required.
- How do we know when to forward?

Dependencies & Forwarding

- The value of register $2 at that point in time says that register $2 initially contains the value of 10
- Anything reading the register before that will see an incorrect value
- Look at "and" instruction following "sub"
- "and" needs the value for execution to perform in CC 4
- It wil perform a subtraction in the ALU at the end of CC 3
- In the EX/MEM latch, the value of -20 will be stored
- It is stored in the latch between the two stages
- Value can be communicated in the execution stage
- Replace the value with -20 by forwarding from the subtract to the and
- Propagates that -20 into the MEM/WB latch at the end
- Keep the -20 value and send it to "or" as a forward
- sw can also receive a value from the register file

Detecting the Need to Forward

- Need to be able to tell what the register input sources were
- Should the instruction have received a forward or not?
- Add a 5-bit specifier until it reaches the data memory stage
- Compare the register destination from older instructions
- We can see cases when we do have a data hazard since we have to have register writes to indicate the writes are going on.
- Hazards matching up sources and check the register write to ensure we are changing the data contents
- The instruction in memory has instructions that match things in execution
- Matches RS/RT of that instruction
- Latch that is in front of the execution stage
- This would be the ID/EX latch
- Instruction that has its state contained in that latch is the instruction currently in execution.

- Righthand side of the instruction will have ID/EX.RegisterRS or .RegisterRT
- Checks against instruction in execution stage if it matches what is on the lefthand side.
- Instruction in EX/MEM latch is currently in memory
- ID/EX is in the write back stage
- Check when the instruction is in the execution stage because that is when it needs the value.
- Additionally, we also need to know if the register is even being written
- The MEM.WB will also check if the instructions in memory and write back are writing to the register file
- We don't do forwarding in the case where we are writing to the register.
- We aren't going to forward if writing to register zero because we should never change the contents of register zero.
- See if we are writing to specifically register $zero

Forwarding Paths

- Augmentation of the pipeline that shows some of the forwarding
- The execution stage: put two MUX's in front of the ALU
- These MUX's should select what goes into the ALU and it should be the ALU source MUX
- These figures leave that particular MUX out
- Assume we are getting only register values and we want to change this to go into immediate as well.
- Three possible sources
- Original register value that was read
- 2nd input is coming from the MEMWB latch and this was 2 before the current instruction
- The 3rd input into the MUX is currently in the Memory stage and take some time to trace those particular inputs in
- Notice if you have an instruction that was feeding a value to a sequentially next instruction, that memory would currently be in the memory stage
- What you want to do is copy the value that ha just been written into the EX/MEM latch and provide it as input to the ALU
- Forward this to the input of the ALU
- Look at ID/EX latch at the bottom
- Having Rt and Rd, in addition to having those, we need RS and RT
- These go into the forwarding unit and use these to determine whether we have to forward or not.
- This will be the control logic to be the input if it's control or not.
- Forwarding unit takes in RS or RT
- It also takes in RegDst for the EX/MEM latch and the MEM/WB latch
- What is missing from this figure is a check for the RegWrite and the check for the MEM/WB
- Additional inputs to this forwarding unit

Forwarding Conditions

- Two different conditions
- EX hazard
- We have a hazard coming from EX/MEM latch coming into the ALU
- MEM hazard
- Coming from MEM/WB latch into the ALU
- It isn't the instruction in the execution stage; it's the instruction in the MEM stage
- This naming of the hazard is a little off.
- The first part of the latch is the value that we will be forwarding.
- In the case of the EX hazard, there are two possible choices that can be made:
- Either forward into A input or B input
- If we are forwarding data value going into RS or RT, these checks will mirror that
- Check if the instruction we are forwarding from and see if it is writing to a register file
- Check and se if EX/MEM latch is high.
- We then want to check if EX/MEM RegDst == ID/EX.RegRS
- In this case, we will set up ForwardA to be 10
- We have the same set of constraints in the 2nd part, but now we are comparing against RT
- Instruction writes from Register 0 and this matches to ForwardB
- With this set of forwarding conditions, we should have outputs for ForwardA and ForwardB

Double Data Hazard

- If you have a sequence of adds that reuse heavily
- $1 in this case
- Both hazards occurring with respect to the last adds in the sequence
- Revise the forwarding conditions accordingly

Revise Forwarding Condition

- We have to check and make sure EX/MEM did NOT have a match
- If EX hazard didn't occur and MEM hazard did occur, we can proceed

Data path with Forwarding

- Has the full data path for forwarding and adds the control inputs that indicate whether or not we have a REG/WRITE

4_11

Load-Use Data Hazard

- load instructions differ from R-type because the register value is NOT ready until the end of the Memory stage
- load followed by an R-type
- The load value will be ready at the end of the fourth cycle

- If there is NOT a bubble, the “sub” instruction will need its value to perform the operation
- We need the value to come into the execution stage.
- Memory is reading on the 2nd half of the cycle
- Lengthening the cycle will cause an overall increase in the cycle length
- No way of communicating that value into the execution stage
- IN one implementation, we would have to add a stall
- One bubble in the pipeline will cause it to line up perfectly.
- You can better see the issue here and circled in red is the communication that would have to occur
- This is simply NOT going to happen
- If we did allow it to happen, it would hamper the Cycle Time of the processor

Load-Use Hazard Detection

- We need the instruction that is dependent on the stall on the ID stage
- Set the control bits and inject in a bunch of zeroes when we are doing a bubble
- Let’s say we have an add instruction following a load
- The load instruction should move on to the memory stage
- The effect of the bubble is that the load instruction will move to the memory stage
- Set things up so that when the load moves on to the write back stage, the load will be able to forward the value contained in the MemWB latch
- The forwarding that is described has already been implemented.
- We have a mechanism for the MEM hazard
- We need to implement the stall
- We need to know if the instruction has the input provided by the load.
- Check against what is in the EX stage
- Check that MEM/READ coming out of EX latch
- Destination register for load will handle there RegisterRS

How to Stall the Pipeline

- Set control values in ID/EX register to be 0
- This is going to force all of our instructions to execute that particular bubble and they will do some work.
- They won’t have MEMWRITE or MEMREAD or anything, so they won’t update values
- Don’t update IF/ID register which keeps instruction in decode
- Don’t update PC which keeps instruction in IF

Stall/Bubble in the Pipeline

- Sustain a one cycle bubble and the “and” that follows becomes a “nop”
- This diagram doesn’t show bubbling in the way we normally draw it.
- In this case, what they show here is a slight change in the accuracy by showing that the “or” did enter an instruction fetch in Cycle 3

Data path with Hazard Detection

- There is a hazard detection unit that tells us when it stalls, and we can find out RS and RT
- It controls the MUX leading up to the CTRL signal write, whether we put in all zeroes or it changes whether we write to PC's or the IF/ID
- Come up with something similar to the forwarding unit.

4_12

Stalls and Performance

- Stalls - maintain correctness but reduce performance
- We can also have the compiler rearrange things but this hurts portability since the compiler has to understand the pipeline structure.
- Look at why compiler's can be better in some cases and in other cases, using the hardware is better
- Static vs dynamic optimization
- Static: before it's run (usually through software)
- Dynamic: do things on the fly (usually through hardware)

Control Hazards

- Arise because we have changes in control flow due to instructions
- All these types of instruction change the actual PC, which changes control of the program
- We can think of a control hazard as one instruction depending on the next
- If the instruction that determines the PC like the branch instruction is still executing, what do we fetch in the IF stage?
- Variety of alternatives in this class that are by NO means exhaustive.

Dealing With Branch Hazards

- There are both hardware and software solutions
- Hardware solutions
- Wait!
- You can not put anything in the pipeline until you know the direction of the branch
- The problem is that you are exposing the pipeline depth again
- Reduces performance by increasing CPI
- Guess!
- Prediction mechanisms are very useful in general because you can break dependencies if you guess WELL
- You need to be prepared to fix when you guess wrong.
- The penalty for predicting has two parts
- How often you're wrong
- The actual penalty when you're wrong
- Risk reward for prediction
- Prediction is useful when you have a reasonable amount of accuracy
- Static branch predictions

- Base guess on instruction type
- Dynamic branch predictions
- Use runtimes of the program to guess what will happen
- Reduce the branch delay

Stall on Branch

- The “or” is conditioned on a branch but it isn’t clear if it will take that particular branch.
- You will have the “add” instruction going first, then the branch instruction will go next.
- Once we have fetched, we start injecting bubbles and we keep stalling the IF.
- We need a mechanism that fetches a bogus “no-op” instruction that goes through the decode in the following stage.
- Stall until the branch actually resolves.

Branch Prediction

- Stalling ends up on every single branch having to force a pipeline stall
- This doesn’t include the performance penalty from a branch with a long latency operation
- For more realistic dynamically scheduled machines, branch prediction is better.
- Relatively large #’s of values would require too many resources, so we only use this for branch prediction
- Assume you are correct, but if you are wrong, you have to correct your errors
- We can guess that the branch is NOT taken and just use PC + 4 as the next address

MIPS with Predict Not Taken

- The load word would have already been fetched and we would have had a correct prediction
- Otherwise, we would have to flush and convert.
- The penalty would NOT be worse than stalling and there would be some complexity at the hardware.

More-Realistic Branch Prediction

- A better strategy would be to use static branch prediction
- Can be based on more specific things like if we have a backwards or forward branch
- If (forward branch)
- loop (backward branch)
- Account for largest % of statements in program execution
- Adjust prediction based on that
- Dynamic branch prediction says to maintain hardware structure
- Figure out if the branch is likely taken or not taken

- Update the table as you execute your program
- Observing past behavior as indicative as a future trend

Branch Hazards

- Assume that the branch hazard is determined in the memory stage
- In the memory stage, we write to PC
- We can argue that we can write to PC in the execution stage, but the book says the branch outcome is determined in memory
- beq goes through pipeline and assume we made an incorrect prediction
- If the “and” is now a follow-through path, they enter the pipeline while the branch is executing.
- If PC is written in CC4, it won’t be corrected until then end of CC4/ beginning of CC5
- load instruction is correct while “and”, “or”, and “add” were incorrect predictions
- Flush the predictions!
- Convert “and”, “or”, and “add” to no-ops
- Set their control bits to 0
- Processor state gets changed in the data memory stage or the write back stage
- For flushing, the “and”, “or”, and “add” have NOT done anything yet to the pipeline
- As the “and” continues to execute, it will NOT write to the register file because we turned off the register flag
- When you are correct, you won’t see any penalty at all (advantage of prediction)

Reducing Branch Delay

- Just put the hardware outcome to a different stage. This would have a larger impact
- Reduces the # of instructions that you would have to flush
- If I move that hardware to the ID stage, we can now put an additional piece of hardware (comparator) that compares 2 register values to determine equivalence
- If I have the register comparator added to my ID stage, we can reduce the branch delay to 1
- Reduce it to a single cycle and if we are wrong, we only pay a single-cycle penalty.
- See if you agree with this as a technique.

Example: Branch Taken

- In the ID stage, there is a new oval that is the equivalence check
- Effective address calculation earlier on with an output that comes from the sign extension and the PC + 4
- Hope is to get a 1 cycle branch delay
- Circuitry that lets you determine the output of the comparator

- Assume that this one cycle is implemented correctly and you have an "and" instruction that you predicted to be correct in the IF stage
- Compare these together to determine if they are equivalent
- If they are equivalent, branch
- Write this value into the PC
- We would be writing the new computed value into the PC
- Change the "and" instruction in the IF stage and make it into a Bubble when it reaches the ID stage
- We would have correctly fetched the lw instruction

4_13

Data Hazards for Branches

- If the branch is dependent on instructions that are before it and it has NOT yet gotten the correct register values, it would NOT benefit from the forwarding logic.
- Instead, the forwarding logic would have arrived too late since the branching has already resolved
- Instead of forwarding to the execution unit, we will forward to the ID stage
- If you have an immediately preceding instruction, it will NOT have completed
- Cannot use forwarding alone
- We need an additional stalling cycle
- beq is dependent on the add
- We need to stall the beq one cycle
- If we had a load word, we would need 2 stall cycles
- Trying to reduce the branch penalty to 1 will add considerable complexity to how we handle data hazards

Dynamic Branch Prediction

- Relates to how much it costs for a branchless prediction
- Make more sophisticated predictions i.e. dynamic branch prediction
- Superscalar pipelines: execute more instructions per cycle
- In more advanced processors, the branch penalty is more significant
- 10 to 20 more cycles
- Stalling is certainly NOT an option
- Do something else like swapping 2 threads
- Some form of prediction table
- Pattern history/branch history table
- Usually indexed by the PC of what you are trying to predict
- 1 or 0 gets encoded based on the predictor
- Either a taken or not taken decision
- There are branches like indirect branches where there are multiple different locations
- Executing a branch
- Check the table
- See its prediction

- Go with it as if it were correct
- We might have to flush the pipeline and actually correct the prediction so we fetch the correct path

1-Bit Predictor: Shortcoming

- Imagine we have a big array indexed by the PC
- Use PC to look up either a 1 or a 0
- Set the bit by checking the branch PC in past history
- Set to 1 if taken before
- Set to 0 if NOT taken before
- Hysteresis: There is some inertia to accomplish this
- If statement that is taken and not taken alternately
- The branch predictor would correct it but this could lead to false behaviors
- Checking based on beq of inner loop and beq of outer loop
- Predicting incorrectly twice and you end up executing it as the wrong statement
- Eventually the inner loop will exit and you set it to 0 because you were wrong
- You are going to be wrong twice!

2-Bit Predictor

- Add hysteresis
- You need to see the event happening twice in a row
- Different states shown in the oval
- The top left one is strongly biased “Predict taken”
- See “not taken” twice to go to that state
- Improves accuracy
- In terms of a mapping of bits, the upper left would have the value 11
- Upper right is 10, etc.
- Weakly biased: one opposite prediction would flip the predictor
- If you see two same predictions in a row, you would have a strongly biased prediction
- Difficult concept to try to grasp and see why it works.

Calculating the Branch Target

- Needs an adder that grabs the immediate field and adds it to the PC + 4
- There is really no easy way to extract that instruction.
- Usually there is a branch target buffer, that lets you get the next PC on the taken branch
- Fetch is extremely complex and treat it at a very high-level
- More complexity to what we see in this slide
- Just focus on direction predictors that we cover with the two bits

4_14
Exceptions and Interrupts

- “Unexpected” events requiring a change in control
- Exceptions arise in the CPU
- Divide by 0
- System call that requires a resource not there
- Interrupts: Keyboard “CTRL + C” is hit
- The processor needs to pay attention to what occurred
- We don’t know when it will actually arrive
- Exceptions generally have a performance impact
- Designed to NOT happen very frequently

Handling Exceptions

- System Control Coprocessor (CP0) handles exceptions
- Architect’s role is to understand what caused the exception
- We have to store what caused the problem itself
- Stored in the Cause register in MIPS
- Assume we have a single-bit here (two exceptions can be handled)
- Either 0 or 1 (overflow exception)
- Jump to a handler address at the end of it (8000 00180)
- Try to fix the problem with the exception

An Alternate Mechanism

- Analogous to a jump table
- Have a cause register and a base address of the table
- Depending on your problem, you would dress to another address

Handler Actions

- See if there is some action that is correct and eventually, it may have to return to the original executed code
- Use EPC and begin executing it again
- If there is NO way of restarting, then we would terminate the program and EPC would be another piece of data that is reported as we execute the program

Exceptions in a Pipeline

- Store it in a separate register to hold the return
- Overflow example
- Raise an exception on the add because you don’t want it overwritten by the add instruction
- You don’t want the add completed because it will overwrite values
- Correctly set the cause of the PC value.
- On a branch, you let the branch continue executing
- Exceptions, stop and restart from the error

Pipelines with Exceptions

- Has a single-cycle branch delay
- “or” gate can have two inputs coming into it
- Determines if you have a hazard or you flush

- Flush controls for the EX stage as well as the IF stage
- EX flush lets us replace the writeback and memory stages
- The IF flush doesn't show much of the operation there.
- Cause register would have additional logic to determine what caused the exception

Exception Properties

- Once you flushed the pipeline, you can perform things on that particular register

Exception Example

- Instruction causes exceptional condition and the handler has to be able to sw
- When the add encounters the exception, it will cover the instructions into bubbles
- We have this bubble injection through the pipeline by selecting 0's instead of the actual control bits

Multiple Exceptions

- There would only be one overflow at a time but we can have an undefined op code
- In that case, we would take the earliest instruction, deal with that, and flush any subsequent instructions
- The problem is that precise exceptions become difficult to maintain
- Talk about the hardware required but it WILL NOT be on exams

Imprecise Exceptions

- Stop the pipeline and whatever point that has an execution, save the state of the pipeline
- Determines what to flush, what to complete, it probably has to do it manually
- Software does more of the heavy lifting here, but this makes the overhead of exception a little bit higher in this case.
- Not feasible for more complex pipelines.

4_14 (continued)

- This lecture will cover exceptions and interrupts

Exceptions and Interrupts

- Refer to an exception as something arising in the CPU
- Overflow in the ALU
- Undefined op code
- Divide by 0
- Interrupt occurs when you have something external to the CPU that causes you to change the control flow of your processor
- CTRL + C on keyboard

- Disk access finishes and the processor has to pay attention to what occurred externally.
- These are all changes to flow control because this needs to change to the program that was being executed.
- The problem with exceptions is that you usually have performance impact
- Fortunately, it doesn't happen very frequently.

Handling Exceptions

- There needs to be a mechanism to hand off a particular case to the Coprocessor (System Control Compressor aka CP0)
- The exception program counter (EPC) stores the PC of the interrupted instruction
- In the implementation that we have in these lies, assume a single bit (two possible exception results)
- Handler address where the software figures out what happens and try to fix the exception

An Alternate Mechanism

- Have a different address from each interrupt
- Analogous to a jump table in CS 33
- Scale to the base address of the table and jump to a different address.
- Jump to a real handler with more context of what caused the exception in the 1st place.

Handler Actions

- See if there is some action that is correct and return to the code that was originally executed.
- If the code is restartable, we can use the EPC and begin executed it.
- Otherwise, if there is NO way to respond, EPC is reporting upon executing the program.

Exceptions in a Pipeline

- Branch to a handler address and they can potentially be returned
- Have EPC registers to store the return value
- Raise an exception if we don't want r1 to be overwritten by the add instruction
- Make sure the instructions beforehand were complete, but we want to make sure the add didn't finish execution yet.
- Raise exceptional state and then set the cause and EPC register values.
- Extra registers required for storing register state.
- The key is on a branch, you let the program continue executing

Pipeline with Exceptions

- This diagram has a hazard distinction unit, which could set up a stall in the case of a load followed by a dependent operation
- ID flush in the control logic

- If either the hazard detection or the flush finds out, we can pull a 0 from the latch
- Potential results: Overwrite REG/WR
- EPC = grabs PC + 4
- Cause register will determine what caused the exception

Exception Properties
- PC + 4 is saved

4_15
Advanced Pipelining

Instruction-Level Parallelism (ILP)
- Involves looking for independent instructions and trying to execute those instructions in an overlapped manner
- Try to find different instructions without hazards or correcting the hazard.
- Exploitation of instruction-level parallelism
- One approach is to deepen the pipeline and find longer pipe depth for cycles that are hopefully shorter.
- Trade off for pipeline stages and the hazards that increase based upon that.
- There might be parallelism and true parallelism with instructions int he same stage
- Multiple issue
- Super scaler
- Different instructions flowing from different pipelines at the same time
- Allows multiple instructions to execute at the same time.
- This assumes that we don't have hazards in this case.
- Usually we use instructions/cycle (inverse of CPI) as our metric
- If we have 4 GHz 4-way multiple issue, we would have a peak CPI = 0.25
- The hard part is finding multiple issues to execute in the same cycle
- Instruction-level pipelining is also known as super pipelining

Multiple Issue
- Design an instruction fetch that has more ports that can support more instructions per cycle
- Resource challenges where we have to come up with the design of a multiple issue processor.
- Static multiple issue are two different options.
- Compiler figures out which instructions are independent and which should be done together
- Compiler handles all the logic for figuring out the hardware.
- You are now restricting to a particular machine and you cannot run an just ANY hardware.
- Dynamic multiple issue
- More common now

- The actual out of order hardware examines the instruction stream
- May do these out of order but this adds some additional complexity since we may have to find ways to handle various types of predictions
- The basic overarching idea is figuring which instructions to execute at any point in time.

Speculation

- We “guess” what to do with an operation
- Start it as soon as we possibly can and evaluate if we were right or not.
- Roll back the state of the processor and try to do the right thing.
- Different from branch prediction: we can determine if a branch was taken or not taken before it took the state
- You need a mechanism to rollback/recover the state in case you were wrong.
- If you speculate a branch is taken instead of not taken, and you start executing a store instruction, you need something to replicate the memory structure.
- If you were wrong about that, you will have to undo the effect of the store.
- This could be a little challenging but in time, we could do a higher-level detail

Compiler/Hardware Speculation

- Do some form of speculation and allow you to recover from incorrect speculation.
- We could do advanced loads and ensure we did the check correctly.
- Buffer the results that are executing before allowing new instructions
- If the speculation was incorrect, we could flush the buffer and start the speculation again.

Speculations and Exceptions

- If there is an exception that occurs, we need to be sure that the instruction was actually executed.
- If the instruction gets squashed, the exception gets squashed as well.

Static Multiple Issue

- The idea is that the compiler bundles things together into “issue packets”
- Compiler statically determines that the instructions were independent
- If the architecture can handle any arbitrary instruction, it can group them together
- The compiler needs to know that so it knows which types of instructions to group.
- Rather than being 32-bits, we can think of it as a 64-bit instruction bundled contiguously in memory.
- This is often called VLIW architecture aka Very Long Instruction Word
- Part of the problem is finding enough independent instructions
- If it can’t, we have to use no-ops (software)
- Pad slots that cannot be filled

Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
- Compiler optimizations and intelligent register coverings.
- We can have dependencies between packets but we have to ensure the compiler knows that those dependencies exist and expose more details of the architecture
- Decreases portability.

MIPS with Static Dual Issue

- We can issue two instructions at once
- Two instructions flowing down to the register file
- Execution: execute two instructions at a time, access memory
- Write back up to two instructions at a time.
- One instruction has to be an ALU or a branch
- Load or store accesses memory and makes use of its own adder and is a little cheaper than a full ALU
- Specialization reduces resources used in the pipeline
- Silicon cost is the reason for this
- Assuming one slot, we can see the addresses are 64-bit aligned and we need an ALU branch instruction.
- Those two kind of follow the buddy system, IF/ID/EX/MEM, etc.
- Just like the pipe we had before in terms of hazards and control signals, allow multiple instructions to go down the pipe together.

MIPS with Static Dual Issue

- Black line is first 32 bits, blue line is the next 32 bits
- Notice there is something wrong with this figure in the Fetch architecture
- Two instructions buffered in the IF/ID latch
- Black lines represent ALU and this will be latched in the ID/EX latch
- Possible branch instructions with ALU at the top at the ID stage
- Implement addi and any other immediate instruction
- Blue path has rs and rt being read out.
- These read out the values of rs/rt
- Sign extension logic that lets us extend it to 32-bits and this can be latched to the ID/EX latch
- Writing to the register file
- On the bottom portion of the data path, we have two lines coming into the ID/EX latch that represent rt and rd.
- In the execution stage, we have two ALU's but the 2nd one needs to be an adder
- The 2nd ALU handles loads and stores and needs to do an add
- Propagates the result of the ALU to the data memory for addresses for load and store instructions
- Don't need to choose if the data output goes into the register file
- Always the ALU output that leads to the register file

- The one that gets hit hardest is the register file
- Support two instruction issues
- Biggest challenge of dynamic multiple issue is supporting enough register ports.
- Hazards in the Dual-Issue MIPS
- Cannot use ALU result for the Store in the same packet
- Adding into $t0 and loading based on $t0, those would have to be separate packets.
- In that case, any hazards that we had before or use full stalling, we would have to use 2 instructions
- Greater penalty for CPI
- More costly to have a stage idle
- Need more aggressive schedulers to have that ILP to exploit the parallelism of the application

Pipelining

- Cover an example of a static issue on the data path.

Scheduling Example

- We are going to use this example piece of code on the dual-issue MIPS processor
- Has a label called loop that will eventually become a branch target
- Puts the value into the same register as the lw above, and it decrements
- Three data dependencies
- Gray, green, and brown
- Bundles it into packets of instructions with two instructions
- In this case, we need to fill this table below with cycles for instructions 1, 2, 3, 4
- There is going to be a load use hazard here no matter what.
- There is no way we could support the load word followed by the add
- Place the addu two cycles later
- Pay attention to the hazards here
- Put load word in cycle one
- Two cycles later in CC3, thea edu works because $t0 is available
- Store word goes after add because it can get its value from forwarding
- bne is put at the end of CC4 because it will branch and will determine if you call lw again
- Specialize the slot to decide which one you should use.
- addi can go anywhere above the bne
- I can change the value of the displacement field of the load or a store if we are dealing with an addi instruction
- When they have placed the addi at CC2, in terms of the code, you put the addi above the add and the sw
- There is an **anti dependency**, which means sw is writing to a location 4 address spaces BEFORE its actual location
- The sw might write to location 996

- The sw might have a 4 for its displacement and the way to think about it is if addi passes this, it will deposit or decrement it into the displacement of the sw
- Unique compiler optimization for immediate instructions
- For cases where the value is known **a priori,** compiler can perform this optimization
- Fixed increment or decrement gives you more flexibility of moving around the instruction that increments or decrements
- You can see that the instructions/cycle has an IPC of 1.25 with peak IPC = 2
- Padded a lot with no-ops because we didn't have parallelizing

Loop Unrolling

- We do want to talk about the tradeoffs, so we have one compiler approach that we can better compare this against dynamic optimization
- Takes a loop body and make more multiple copies of the loop body to exploit more parallelism
- We don't want to have storage hazards and we don't want to create an artificial dependency.
- On the lower left, we have the same scheduling that we did before.
- The right is the first step of the process of loop unrolling
- The way to determine it is to see how many registers being used and we want to factor in how much parallelism we are extracting.
- We may know something about how many times the loop is taken
- If we know the exact # or a factor of a certain # i.e. multiple of 2, we might enroll the loop twice and know it will be executed an even # of times
- Reinmann will always tell you how many times it will be unrolled
- If we unroll 3 times, this means we will make 3 copies of the loop body and we have a total of 4 version of the loop body (1 original + 3 copies)
- The end instruction will still be the bne (go from 5 instructions to 17 instructions)

Loop Unrolling (2)

- addi can all be combined together and they are basically decrementing $s1 by 4 each time
- Move addi and condense the addi's into 1
- Take the first addi and move it all the way down to the bottom
- If we move it before the bne, we would have to adjust every load and store we pass by decrementing the displacement by 4
- Change the 2nd load word to be -4 and every subsequent load or store would have to get a -4 in its displacement
- Keep subtracting, for each addi
- Combine each addi to get -16 eventually
- Possible because this is always dealing with s1 and the compiler could certainly combine these together
- Definitely would be able to drop it to -16

- They all have the appropriate amounts of decrements added based on the # of addi
- If we didn't know that, we would have to add additional intervening code that adds complexity to the loop unrolling

Loop Unrolling (3)

- In this case, we will have 14 instructions instead of the original 17, we still have a problem
- Parallelism means that for example, if we look at the dependence chain here, we would keep repeating the process
- 4 independent load word chains
- Also the 5th independent chain (s1 is still used by loads and stores for addi and bne)
- If you look at the lw chains, we use the same register ($t0)
- You cannot use lw in parallel because you would keep overwriting to the same register
- Need additional registers for additional space
- Analyze space-efficient algorithms vs time-efficient algorithms
- This is very space-efficient but poor in terms of time-efficiency
- The version on the right is a renamed version with more registers, so better time efficiency and worse space efficiency
- There is an increase in the cost of registers here but improved in terms of parallelism

Loop Unrolling (4)

- Schedule that in the static slots
- In this case, there will be more cycles on the lefthand side and we want to fit these into as few instructions as possible.
- We still only have one slot and in this case, the load words can go in cycles 1, 2, 3, and 4.
- We still need to obey the data hazards
- First hazard is to the first add, which goes in CC3
- Communicates in register dependency to CC3
- Next lw communicates through a cycle dependency in CC4
- These are all put into the slot for ALU operation
- First store word could go in CC4
- We already have something in the data transfer slot, so the 1st transfer could go into CC5
- We could now fill in the rest of the 5th dependence chain and this is where the PC will change.
- Now we have flexibility
- 3 slots left and the easiest thing would have been to put the addi into CC7
- Put the addi into the top at cycle 1
- The addi moves above every instruction except for the 1st lw
- Notice that the last sw is adjusted by adding 16 to the original value

- Original value was -12 but the addi was decrementing but we need to make this relative to $s1
- The first lw because it occurs in the same cycle as addi, it cannot receive forwarding from addi
- The lw doesn't have to be adjusted since it will NOT see the impact of the write.
- No forwarding of instructions in the same packet
- We are able to reduce this loop to 14 instructions in 8 cycles
- Had we done 4 iterations of the loop, those 20 instructions would have been done in 16 cycles

Loop Unrolling Example

- It shows that the IPC has 14 instructions/8 cycles = 1.75
- There is a cost of using more registers and this may mean we have to do more spilling => more loads and stores
- By replicating the instructions, we are adding to the instruction footprint in memory
- This means we may need more cache base to hold more instruction footprints.

4_17

- More advanced pipelining concepts: dynamic speculation

Dynamic Multiple Issue

- CPU decides how many issues to use per cycle
- Avoids need for static scheduling
- May help get the compiler involved for certain optimizations
- Makes things easier for CPU to find things
- The fundamental tradeoff between dynamic and static issues
- Dynamic is good for knowing any period at any given time
- Static is good for longer distance optimization
- We need to focus in on what the hardware does for multiple issues

Dynamic Pipeline Scheduling

- You have a subtract that feeds an slt
- lw and sub are independent instructions
- No reason why they couldn't start at the same time
- The CPU itself would be the one that tries to figure out that load and store are independent and get them in parallel
- Even though I am executing things out of order, we are still going to commit
- Commit: Place registers in order and let them affect the order

Dynamically Scheduled CPU

- Organize this slide by distinct phases that occur
- Fetch them and decode them in order

- Order in which they are executed in a conventional machine
- Sequence of out of order execution where we execute things around there dependencies and allow speculative execution.
- In order commit stage
- Put things back in order and maintain precise exceptions
- In case, we did any kind of crazy speculation
- The in-order commit phase should be able to handle that.
- Out of order execution
- Can use a reservation station
- Sits and waits for a functional unit
- Case in point would be an ALU
- There are a few that handle integers, floating points, load-stores, etc.
- All these different functional units is where executions occur
- Instructions sit and wait until register operands are ready.
- Its dependent instructions would go into reservation station and out of order execution would enable you to use the subtract instruction to execute the same time as the word.
- Any instructions could execute in a functional unit
- These would all eventually drain out to the commit unit

Register Renaming

- Used in the context of static issues
- Reservation stations need to be renamed at certain times
- Register renaming occurs when every logical register is put in, and this changes a physical register
- Anti-dependences would be renamed, and by renaming from logical to physical, we can execute the instructions in any order
- The real beauty is that the ISA is limited by the # of bits for specifying a register
- There is no limit at least not in theory to how many registers we can map.
- We can make it as large as possible in the given technology.
- The reality is that register field are the more expensive parts of the design and there is a limit to the # of machines used in the low 100's.
- There is one machine that makes use of 512 but this was eventually cancelled
- In terms of register renaming, the registers get renamed at the front of the machine
- Physical register specifiers indicate the dependencies and in physical registers, the instruction that completes gets written to a BUS that gets snooped to register reservation stations

Speculation

- Go crazy with executing instructions and all of this can happen aggressively in the out of order component.
- We will wait with that commit until we are sure that actual instruction will be executed.

Why Do Dynamic Scheduling?

- The question is why put all the effort into Silicon and this is much more difficult in many ways in software instead of sticking all of this in the compiler.
- Majority of devices embrace dynamic scheduling
- Difficult to predict stalls at compile time
- Direction the branch might go, many factors come into play!
- Depends on the actual data coming into the input
- This is something hard to know statically and is something more to be known at runtime.
- Difficult to schedule branches in ISA's
- Difficult to schedule around load and store hazards in some ISA's
- Portability
- If we have an ISA that is implemented in a machine organization with a certain set of hazards and stalls, we need to expose those to the machine
- Change the compiler every time we change the machine organization
- Reduces the economy of scale out of a single compiler design.

Does Multiple Issue Work?

- Yes, but not as much as we'd like
- Why we turned toward multi-core and multi-threaded applications.
- Diminishing returns and we have given up on multiple issues
- Fast majority of machines are multi-issue and we had to add another tool to our tool belt.

5_1
Large and Fast: Exploiting Memory Hierarchy

Memory Technology

- Focus on in this class are the differences between static RAM (SRAM): Used in caches, and dynamic RAM (DRAM)
- Static RAM is the most expensive but you can get data fastest
- Dynamic RAM has an order of magnitude more latency than the static RAM but it is also significantly cheaper
- Magnetic disk has even lower cost but extreme latency compared to the other two.
- Difficult to find the balance and there is no one memory that does it all and this motivates us to build a hierarchy

DRAM Technology

- Implemented as a single capacitor per bit and it is a destructive read.
- You have to put charge back into it since charge is removed from the capacitor upon read.
- A lot of work has been done to try to reduce the cost of an access
- Focus on implementation in this class
- Volatile: lose power => lose data

- Higher in capacity than cache but not as high as a hard disk

DRAM Generations

- The table on the left is emphasized. The progress of time shows DRAM's scaling. Enormous gain over the years as capacity scaled dramatically
- Latency is not scaled as the same way as the transistors are scaled.
- Getting the memory to the CPU to get it fast enough for aggressive cores.

Flash Storage

- The basic idea is that Flash Storage is non-volatile, doesn't lose data when it turns off.
- It is a little more expensive than DRAM in terms of cost per GB
- Some people use Flash to replace hard disks
- Can't get the same capacity on the disk
- Lacks some of the capacity in the latency benefits of DRAM
- Nonvolatile storage
- We could use flash as our intermediate stage between memory and the hard disk
- We could store things in nonvolatile state on the flash storage

Disk storage

- Highest capacity which can reach terabytes
- Made of a stack of different disks of rotational disks with read heads.
- You have the latency of turning the disk and moving the write head
- Contributes to long latency access of disk storage
- Large capacity comes from the densities of the materials

Principle of Locality

- How do we build something that gets around these limitations
- Not concerned with how hard disks work and storage constraints
- What do we need to do to make the best use that we can.
- Exploit the principle of locality and is used in most applications.
- Some applications do not have any locality at all, but some of these might work because of locality
- Talk about class applications that don't work with locality.
- Locality refers to accessing the same data over and over again
- Temporal locality: Items accessed recently will likely be accessed again soon
- Executing the same instructions over and over
- Spatial locality: Accessing things close together in time
- If you have an array and you are pulling in a block of data at a time, once you access one element, it is likely you will access the next element in the array
- Exploit accesses that are near to each other in space
- We will now look at leveraging these and see if you get it.
- At the highest point of the hierarchy, we will have cache memory
- High turnaround speed and good access latency

- The next layer will have larger caches and we will have a longer latency and it will take longer.
- Eventually, we will go to main memory and we are paying more for longer latency.

Taking Advantage of Locality

- Everything we want is on that disk and this is the final location that has the highest cost in terms of latency
- Take advantage of locality by accessing something using DRAM or disk
- By bringing things we recently accessed in time, we will be able to exploit spatial and temporal locality in the address string.
- Bring it up to our 1st level cache

Memory Hierarchy Levels

- Accessing one level of a hierarchy and a 2nd level of a hierarchy which is an even higher capacity component.
- Look at faster access, find what it wants, then goes to the next level of the hierarchy.
- Data is transferred in blocks
- Chunk size of data that we are pulling
- If we have an access to the 1st beta level, we will bring an entire block of memory into the first level of the memory hierarchy.
- This is the concept of hits and misses
- SRAM might not have the capacity to store everything.
- We need a concept of hitting and finding what we want in the cache.
- Get things from the next level of the hierarchy until we find the actual data.

5_2

Cache Memory

- We will focus in on cache memory
- Implemented in SRAM
- Relatively fast in access latency but expensive in cost per GB
- Relatively small compared to main memory or disk
- Fast enough to give high performance on our processor
- In an architectural sense, how do we use this to our best extent
- Bring data in and manage it!
- Closest to the CPU and multiple levels of caches
- Three levels of cache
- L1: Usually the smallest but has the best access latency
- L2: Happy medium between L1 and L3
- L3: Largest size but longest access latency
- Imagine program is executing and you have loads and stores
- Access streams and use those different accesses up in a cache
- Grab the data fast out of the cache or you would have to miss to the next level of the hierarchy

- This gets to the idea of how you organize the SRAM into a cache

Direct Mapped Cache

- Lowest overhead cache to implement
- Cannot hold everything
- Part of the challenge is to hold 8 chunks of data
- These are called blocks
- To hold three chunks of overall memory space and we have some access for block X
- The location where I look will be determined by my address
- Only exist in one block of that cache
- Narrows things down quite a bit and makes it easy to implement what we want as opposed to checking the entire cache.
- Can hurt the hit rate.
- A notion of associativity
- How many cache blocks can associate with a part of memory in the cache
- Take address of the individual block and do that % the # of blocks like in a hash table
- This illustrates this by the blocks in gray mapping to a single location in cache
- Using only blue blocks would be worst case behavior and the reason we are using this mapping is because the cache holds a subset of memory and we need to know where to look in the cache for a particular address
- The cache's they are showing here is using the block addresses
- All the blue blocks end in 101 and they map to the same location
- Use three bits to see to which entry they exist in the cache
- First example we are showing in the book

Tags and Valid Bits

- Because the cache holds a subset of the overall memory, we need more information i.e. tags and valid bits
- The tag helps us know which particular block is stored at a particular location
- The blue blocks all map to the same location, so we can only hold 1 at any point in time.
- Add additional bits to tell us if we have a match or not.
- The cache doesn't have to hold data
- It is volatile when the power turns off
- Initially, there won't be anything in the cache
- We need to mark it as validated vs invalidated
- We need some way of telling
- Flush the cache between contact switches
- Valid bit is 0 means there is nothing in the block

Cache Example

- This is an 8 block cache
- This example is one word per block
- 32 bits in a single block
- A # of indices and this is distinguished between 3 bits
- V is the valid bit set to 0 initially or NULL
- Tag is empty to show there is currently nothing in the cache at all.
- Cache memory that we have been talking about is byte-addressable memory
- For the purposes of simplifying this interruption
- When we start looking at a more general form, we are looking at byte addresses and we need to address this methodology a bit.
- 22 in binary is 10110
- 8 bit entry, direct mapped cache
- Use lower 3 bits of the word address to indicate to which index we should go.
- Lower 3 bits would be 110, so we will go to the 7th entry if we are counting from 1
- This will be a miss since it is NOT initially in the cache
- At the index 110, we mark the Valid bit as Yes, mark the Tag as 10 since this is what is left for the index, data will be whatever it is in memory at that particular location.
- Access address 26, we will look at the 3rd index and it is also a miss, so we have to grab it from memory and mark the Valid bit as yes
- Set the tag to 11 and put memory contents into the data portion of the cache block
- If we were to have two more accesses i.e. 22 and 26, they need to be put into location 110 based on the address and see that the tag matches the expected tag that we are looking for.
- Use this in our processor for further computation, and this is a case of split locality
- These instruction addresses will be put for both temporal locality and holding it in the cache will have 1 hit and 1 miss.
- Subsequent addresses that access it.
- The first access is 16 and that is a miss
- The second access is 3 and is also a miss.
- The third access is 16 and it is finally a hit
- This new word 18 is a prime example of the limitation of a direct mapped cache
- We cannot use other open spaces because the address indicates where we need to put the cache (like a hash table)
- We have to kick out the earliest element in the cache so there are many collisions

Adress Subdivision

- Have a rectangular box showing the address
- 3 parts to the address

- Lower part is the byte offset
- Chosen by the size of the individual cache block
- Fixed for a particular machine organization
- It is usually based on how large the spatial locality is in the cache hierarchy
- If blocks were 32-bit like the previous example, it would be different than if the blocks happened to be 32-bytes.
- Form the byte offset of the blocks, and use 2-bits to index.
- This means that there will be a total of 4 possible bytes within that 2 bit offset.
- 4 different bytes we can select
- If we had driven out that 32-bit chunk of data and we were doing byte-level access of the memory, there are 4 different bytes we can address
- The cache doesn't deal with the byte offset; the byte offset is used when we read from individual blocks
- Move on to the next chunk to the left (10 bits coming from the middle of the address)
- This is called the **index**
- Tells us where we need to look in the cache for a particular block
- Indicate which of the entries we need to look into
- 10 bits after the byte offset to index into this particular cache
- Based on the # of entries into the cache
- Tag is whatever is left (20 bits in this case)
- 32 - 10 - 2 = 20 bits
- These 20 bits compare on an access what is in the cache tag portion and the comparator reads to an AND gate
- The AND gate has the valid bit coming out of the cache to detect if we have a hit and that lines tag matches the tag of the request
- Make use of the 32-bits of data that is driven out of the cache.

5_3
Example: Larger Block Size

- Contiguous chunks of data
- Cache blocks can be a variety of different sizes
- Think about this in terms of if we have a larger block size in a given miss and we send it into the cache, we can exploit more spatial locality
- Grab more things closer to a particular miss
- Limited space in cache may inhabit how many miss streams we can access
- Lots of different misses can limit in how many we can manage since we cannot predict where the misses occur.
- Given one particular block size, how do we handle the mapping of the Tag, Index, and Offset
- Use bits as an index into the actual cache itself
- Take the block size of the cache and do a log_2 of the bytes we have in a given block

- Use this value as our offset
- Imagine you have 32-bit blocks and you have 5-bits for the offset (skip these)
- Head up to the next set of bits to index

Block Size Considerations

- You can exploit more spatial locality to reduce the miss rate
- Large blocks can mean we have fewer total blocks in the cache
- If I have 4 byte blocks, we can have 16 of them total and if we had 32-byte blocks, we can hold 32 of those blocks.
- If we have different misses across the address stream, we can lead this to pollution.
- In terms of transferring the data, it will take longer and we can try to handle this by taking the critical word first
- We can use this as a consideration when taking into account larger block sizes

Cache Misses

- On a cache hit, we have what we realized before with an ideal architecture
- Blocks could always be accessed in a single cycle
- On a cache miss, you may have to do different things
- Stalling the pipeline and awe can continue executing instructions
- The reason for stalling the pipeline is to get the next block in the memory hierarchy
- On an instruction cache miss, you have to stall the front end of the pipeline and bring those instructions out of memory

Write-Through

- Loads and stores can cause misses in the cache
- How do we handle the disparity between the data that exists in memory?
- It can be inconsistent with respect to memory
- The question is how do we modify the contents of address X along with the modification you make to the cache?
- Or do we accumulate as many changes as we can and then we evict the block.
- The difference between these two approaches is called **write-through** vs **write-back**
- If I write to the cache, you also write to main memory immediately
- Makes writes take longer because you have to use bandwidth of memory hierarchy to keep writing multiple times.
- The benefit here is if I evict a block, I don't have to wait for the update to occur
- Updating ahead of time in order to reduce the penalty later on.

Write-Back

- Tracks when blocks have been modified
- Set a bit called the dirty bit
- Indicates we have modified the cache contents
- If I am forcing a block out of the cache, we check our dirty bit first.
- If we have lots of stores to the same blocks (uses temporal locality), you only need to update once in memory
- You need to do this update when you replace the block
- A load will not only wait but it will write back to the current block.

Write Allocation

- If we have a load miss or read miss, we will bring the block into the cache
- For stores, there might not be locality between stores and loads
- We don't want to pollute the cache and remove things that may have temporal locality in near future
- If we have a write miss, we allocate a block and this is most useful for high stores of locality
- Write around doesn't fetch the block
- Write back usually just fetches the block
- Write allocation uses both write around and write block

Measuring Cache Performance

- Look at a better quantitative model and we can do this by considering the frequency in which we have a memory stall.
- Overall, the way we consider this is to look at the # of instructions in the program and memory hierarchies can have multiple misses and differing miss penalties.
- Look at this notion of frequency at which an event occurs, and the subtle levels of the hierarchy and penalties.

Cache Performance Example

- Two caches are backed up by memory that takes 100 cycles to access
- Loads and stores make up 36% of all instructions
- Baseline CPI is 2 in this case
- Given all these initial conditions, find out all the missed cycle.
- 2% of the time, we will use a miss
- Only 2 instructions out of 100 will miss, but we will see 2 cycles per instructions per miss
- 0.02 * 100 = 2
- Actual CPI = Base CPI (ideal cache) + I-cache + D-cache

Average Access Time

- Instead of considering the impact of misses, we have to consider the hit time for a particular miss
- For CPI, we were only concerned with the miss rate or frequency
- If we look at a CPU with a 1 ns clock, there would be a single cycle to access and the miss penalty is 20 cycles

- AMAT = 1 + 0.05 * 20 = 2 ns
- AMAT = clock + I-cache miss rate * miss penalty

Performance Summary

- As CPU performance increases, the miss penalty becomes more significant because of the poor scaling of memory
- We need to find a significant amount of time in memory stalls
- Figure out a better way of handling the memory hierarchy and we cannot make assumptions about ideal memory in terms of accuracy in the pipeline.

15_4

Associative Caches

- Choose larger block sizes to expose greater spatial locality
- Another factor that can influence the cache is associativity
- How many different entries may we use for a particular block of memory
- We are completely low on flexibility, but we only have to look at one location for any given address
- More flexibility means we have to look at more than one location for an access
- The cache addresses that I pulled in from memory can have complete and total flexibility
- We cannot just look in one place, we have to look all over the cache
- This requires a comparator for entry in the cache and this can be expensive in terms of power.
- Fully associative is generally used for smaller #'s of items because it gets expensive
- n-way set associative
- We need to organize the cache into sets with *n* entries
- Only need n comparators to limit the amount you have to compare for byte addresses alone
- This introduces some new vocabulary for caches
- Looking for a particular block and the ways are the particular entries for each set.

Associative Cache Example

- In all three examples, the idea is that we have 8 entries and this part doesn't change
- For a direct mapped cache, we use 3 bits to determine where to look in the cache and we only look at one place.
- In the middle, we have a set associative cache and n = 2
- Two way set associative cache means we have 2 entries associated with it and there will be 4 sets total.
- Set 0 has the first 2 beta blocks associated with it
- If we select set 0 as the mapping, we have to look at both of the data blocks

- Comparison of two different tags and in terms of the fully associative caches, we have to search all the entries in parallel

Spectrum of Associativity

- For a cache with 8 entries, it can be direct mapped (one way per set)
- 8 different sets with one entry per set
- 2 way associative
- 4 way set associative
- 2 sets each with 4 entries
- With greater associativity, you get greater flexibility and larger amounts of power and timing to do comparisons
- However, with smaller associativity, you have less flexibility and miss rate would increase

Associativity Example

- Direct mapped implementation which is 1 way set associative
- Fully associative alternative with single set and 4 blocks
- To access these with 0, 8, 0, 6, 8; in a direct mapped implementation, block address 0 would have cache index 0 and this would miss in the cache
- To access the block 0, we would pull it into the zeroth set of the cache
- At index Mem[8] it would map it to the same index and we can have another miss
- This is known as **cache thrashing**
- In a direct mapped cache, they cannot coexist in the same index!

2-way set associative

- See that in this case, there is a different mapping of our addresses since we have two ways only rather than three different
- Four sets and now down to two sets
- In this case, 0 and 8 can coexist since they map to the same index
- Two different entries and they can access each other
- Because of 6, we now map to the same index as 0 and 8 and we do some thrashing towards the bottom.
- For a fully associative cache, we can have 0, 8, and 6 to coexist and hit on any subsequent accesses

How Much Associativity

- Increasing associativity decreases miss rate but you have diminishing returns
- Shows less performance impact on a real example over time
- Vary the associativity, shown in a sequence of #'s below
- Direct mapped at 10.3% miss rate
- Kept dropping as we go to higher and higher associativity, but the drop change gets smaller and smaller

Set Associative Cache Organization

- Two byte block size with two bits used as offset in the address
- Because we had 1024 total entries, we will have 256 sets
- Each set is represented and each index bits on the lefthand side will be 0-255
- Two different chunks of Tag and Data block
- Different ways of the cache
- 4-way in this case
- 8-bits in this index will be used to select a particular row in the cache
- Drive out the text and we have 4 comparators at the bottom
- Check the tag bit against the tags that are stored against the cache ways and check if the valid bits are set.
- Represented by an OR gate at the bottom
- Allows us to select which of the data blocks are output in the case we have a hit.

Replacement Policy

- Don't have anymore space in the cache and we don't have blocks that are invalid
- In a direct mapped cache, we don't have a choice
- Kick out a block
- Set associative will have 2 or more choices
- All four elements are valid, then we have to choose one of those to remove from the cache
- Make use of a replacement policy and choose which cache element was the victim.
- Least-recently used (LRU)
- Choose element of set that has been unused for the longest time.
- Easier for 2-way, harder for 4-way since you need to manage 2 bits, but much harder for anything at the end
- Random
- Use FIFO or LIFO algorithms and this gives about the same performance as LRU for high associativity
- Implementation and results will vary
- Much more important for a small associative cache to have an intelligent algorithm for replacement
- So many different choices, we can approximate LRU behavior

5_5

Multilevel Caches

- Single cache backed up by memory
- The reality is that most cache hierarchies have multiple levels of caching
- Beyond L-3, we go into main memory
- Focus having multiple caches attached to the memory hierarchy

Multilevel Cache Example

- If we have a miss rate of 2% (1 cache for instructions and data)

- If our main memory access time = 100 ns
- If we only had a primary cache
- Miss penalty = 100 ns / 0.25 ns = 400 cycles
- Miss penalty = main memory access time / (1 / clock rate)

Example (cont.)

- Add L-2 cache
- What we will see is that the primary miss and L2 cache hits would have a penalty of 20 cycles
- This means that if the L-2 also misses, we will pay 400 more cycles in addition to that penalty
- CPI will have 1 + 2% of the time (from 1st level cache) * (20 cycles + 0.10 * 400)
- CPI = 1 + L_1's miss rate * (L_2's penalty + L_2's miss rate * extra penalty)

Multilevel Cache Considerations

- First level cache will try to focus on reducing the hit time as much as possible
- This will be the go-to component
- L-2 has more backing storage and capacity to reduce the amount of time you have to go into main memory
- Multiprocessor designs will enable some sharing of data and they will have a larger capacity but longer latency
- Block size may differ between L-1 and L-2 cache
- May be able to sustain larger blocks if the block sizes are varying

Interactions with Advanced CPU's

- Instructions can occur during a cache miss
- With this possibility, we have a great deal more sophistication if we have multiple cache misses
- But, we need more specification and this is generally only applicable to simple machines
- Allowed to overlap memory accesses
- These analytic models can only do so much
- It might be better to use a real system simulation

Interactions with Software

- Ordering in terms of different accesses and localities is based on software itself
- Whatever compiler optimizations we have done with memory accesses
- Compiler optimization of blocking, break the data set up into chunks that are more easily cacheable
- Access large memory structures to improve memory usage considerably

5_6

Virtual Memory and the implications of the cache when dealing with virtual memory

Memory

- Fundamental idea where what we look for is rather than a shared address space for applications in execution, we allow each application to think that it has its own virtual memory space all to itself
- Possible for applications to have overlap in their original memory space
- We don't need to know what is running at the same time as other applications
- Give each one the indirection that it has its own address space
- From our standpoint, a lot of the implementation will be seen from a hardware perspective
- We will look at software a little bit but only on a need to know basis

Address Translation

- Linker assigns virtual addresses with the assumption that you have a large space all to yourself
- Map virtual addresses to some physical addresses in real system memory
- Disk is the ultimate backing store
- Cache holds some subset of the overall pages and the remainder
- Granularity of data movement is moved at a cache-block granularity
- Virtual memory is page granularity
- They are similar but historically, they are similar things
- Pages are fixed size and we can optimize for a particular size
- The top is the virtual address and the assumption we have here is the virtual address is 32-bits wide
- The main idea is that we are starting with 1 address size and translate a physical address.
- Page offset is analogous to a cache addressing scheme
- Based on the size of a page rather than the size of a block
- We would have 12 bits to condense it to our page and we would have a page offset
- The page offset is kept the same and the page which is 4 KB will be in either virtual memory or physical memory
- Offset does NOT change based on the translation of that address
- What does change is the upper bits.
- If you think about it, taking the total size of the virtual memory (2^32 or 4 GB) and divide through by the page size (4 KB or 2^12), you will be left with 2^20 pages in your virtual memory space
- You can identify things based on the virtual address
- These 20 bits get translated to a physical page #
- For a 30 bit physical address, so we have 18 bits for the physical address
- 2^20 for this particular application

Page Fault Penalty

- We would go to what is called a page fault

- Mechanism to perform a translation and we have to go to disk and place it into physical memory
- Takes a lot of time in terms of clock cycles
- Handle what we kick out
- Handled by the Operating System
- Many ways to minimize page fault rate
- Our job isn't to handle page faults; we want to make page hits work faster

Page Tables

- Primary location where translation occurs
- Resonant in memory (not a hardware structure)
- Each memory application can have its own page table
- This is a hardware register
- Points to the base of the page table in physical memory
- There may be pages of the page table and let's assume the page table fits completely in physical memory
- Perform a load into memory and check if the page is there.
- There may be other info we need such as protection info, dirty bits, etc.
- If the page is NOT present in memory, we have to check where on disk it is located.

Translation Using a Page Table

- At the top, you have a page table register
- Memory resident structure
- NOT a hardware structure
- Tells us where to look in memory to get a page
- There is a valid bit that indicates whether that page is in physical memory or not
- If not, we have a page fault
- If yes, then we look at the physical page, which retains its 12 bit offset.
- These make up the full 32-bit physical address and these are all #'s of bits that can vary based on the implementation

Mapping Pages to Storage

- We can indicate where the component is in disk storage.
- The idea is that the page table has the valid bit to indicate when the page is in physical memory and extract it from disk storage instead.

Replacement and Writes

- Disk writes take considerably longer than other memory accesses
- You usually have large size pages and you don't use write though policies with virtual memory because writing to disk so frequently would be costly
- Try to use bursts as much as possible to avoid having to use too many disk writes.
- LRU policies are managed in Virtual Memory in order to avoid page faults

- OS is the one in charge performing the management as well as making use of pages as efficiently as possible.

Fast Translation using a TLB

- How to make translation fast?
- Hardware structures come into play here
- Every address translation that is required for every instruction would have to use an additional memory reference
- The first thing we need to do is load in the page table register
- Then, we need a 2nd load to actually extract data from the physical page.
- Memory accesses are already expensive, so notice page table accesses have very good locality.
- Page tables are huge!
- We have good locality with respect to page table translation
- Cache the page table
- To some extent, this happens via a conventional cache
- We need a separate hardware specific translation mechanism that is a cache of the page table memory
- The translation look-aside buffer (TLB) doesn't hold the entire page table, but it is a cache of the page table
- Usually, TLB hold 16-512 PTE's
- Low amounts of latency
- Misses are more expensive because you either go to the cache or the next level hierarchy
- Miss rates are pretty small!
- Page table walk is required in a miss
- Handled in hardware usually
- Could also be handled in software

Fast Translation Using a TLB

- Used as the 1st level of the page table
- Upper bits of the virtual address
- Check for a physical match in the page address
- In a TLB hit, we can look it up in physical memory without having to go to the page table at all.
- Get the requisite page address from there and put it into the TLB next time.

TLB Misses

- If a TLB misses, if the page is in memory (no page fault), we can go to the page table entry and retry the actual load
- Can be handled in hardware and accelerate a TLB miss
- In most modern hardware structures, we need some form of page table walker to avoid exceptions at all.
- If it is NOT in memory, the OS has to be invoked to figure out what to kick out from set of pages in memory itself.

- We are going to assume some form of kernel interruption has to occur for a fault.
- The hardware structure has to be able to handle the misses.

TLB Miss Handler

- Because we are going to grab the physical address from the page table, we need to stop requests coming out to the actual cache
- We cannot let register value for a load to be overwritten
- Handle the notification to the rest of the pipeline so everything else doesn't fuck up.

TLB and Cache Interaction

- Cache itself can use physical addresses and then we need to do translation ourself.
- Caches can make use of virtual tagging or a hybrid of a virtual address for the index and physical address for the tag
- Attractive because we can access the cache in parallel
- Use the physical part for the tag comparison in the cache.
- What is shown in this slide is one organization of the cache with purely physical addresses
- In this case, they are using a fully-associative TLB
- When the physical page # is found, we use this to calculate the offset
- This is then used to access the actual cache and determine a hit or miss.

5_7

- Last set of slides in memory hierarchy

The Memory Hierarchy

- Exploit locality through caching
- Virtual memory, actual cache memory, etc.
- All have a common set of attributes that apply
- Different mechanism for block placement and mechanism to tell when we miss and select things to remove from the cache

Block Placement

- More associative selections will have to deal with thrashing issues inevitably.
- Direct mapped cache is used to work around trashing

Finding a Block

- With direct mapped, you only look at a single place and single tag comparison
- You could do something equivalent where you stream down cascading images.
- Fully associative means you look everywhere

- In terms of hardware caches, the problem is that you have mechanisms that help you reduce the # of comparisons to drop that structure but are used in order to reduce that.
- Full associativity becomes more feasible if we have a smaller # of items
- There is NO real miss rate, rather than a page error that we compare against.
- With virtual memory, the full page table approach is never used by itself.
- In most modern size, the TLB makes you look small

Replacement

- Pick a box and then choose an item within the set.
- Least recently used
- Pick one of the photo of using graph and design
- 4-way associative needs 4 pits
- Random replacement
- Gives some feedback from LRU
- Can base it on pure randomization, the class structure it is embedded it, etc.
- FIFO: Takes a round robin approach based on the Eggert
- If a cache misses, you have the opportuniity scheduling you time effectively.
- Write-through means you make modifications immediately
- Write-back means you only update that particular level you are talking about and you defer the modifications on the next level until the contents get kicked out
- Virtual memory only allows write-back given the cost of writing to disk
- The amount of writes to disk simply doesn't make sense if you have to amortize the cost by buffering a larger block.
- Write back is the only alternative for virtual memory

Sources of Misses

- A miss in general can be identified as a particular class.
- Helps you avoid those types of misses.
- Compulsory misses (aka cold start misses)
- If it was not in the cache, that is called a cold start miss.
- We haven't seen the block and it wasn't a bad mapping. There was enough space.
- You simply haven't seen the block before.
- Capacity misses
- You cannot hold all the blocks you want to use
- Cache size is an issue
- Conflict misses (aka collision misses)
- Only found in a non-fully associative cache
- You simply cannot hold all the blocks you want to hold in the cache.
- Holding two blocks would mean the cache is directly mapped.

- In this case, even though we have plenty of space in the cache, those blocks simply couldn't coexist.
- Constantly getting misses that are conflict misses because of conflicts of the same index in the cache

Cache Design Trade-offs

- The effect on different types of misses would be to reduce capacity misses
- Overall cache size was NOT sufficient.
- Increasing cache size can help in regards to conflict misses.
- Try to reduce the capacity misses.
- Increasing the cache size can affect access time as well as power in the cache.
- Try to get rid of conflict misses from enough space in the cache but we couldn't get all the space because of restrictions in block misses.
- Hopefully we can leverage more of our cache capacity.
- Associativity means we have to do more tag comparisons and we can use data blocks on a particular access.
- Increase in block size will decrease compulsory misses
- Larger block means bringing in more data
- This means more spatial locality
- This compulsory miss might turn into a hit because we have brought in a larger block including the compulsory miss.
- Takes a longer time so miss penalty gets worse.
- Large block size with a relatively small cache size means an increase in the miss rate (capacity miss or conflict miss)
- You cannot track as many blocks anymore.
- You have fewer blocks you can hold in the cache.

Concluding Remarks

- Physical design impacts this
- Design of computer architecture is affected
- Problem in the physical design sense: we don't have a large, fast memory
- We need more and more capacity to hold data for our applications
- Create a higher cache at the top that provides fast access with low capacity
- At the bottom, we will have disk storage that provides backing stores and there will be more levels to the local disk.
- For our purposes, we have enough capacity with just the local disk.
- This hierarchy works and nearby accesses that are used frequently will allow us to leverage this hierarchy and give the illusion of a large, fast memory
- These systems require a larger amount of memory to feed these cores.

7_1
Multicore, Multiprocessors, and Clusters
Introduction

- Goal is when we start running out of performance we can extract out of a single thread
- The cost of extracting an ILP is such that it becomes difficult because of energy, performance, and scalability issues
- This is what happened with the Pentium 4!
- It was way too expensive to keep extracting performance from a single thread.
- The idea was that if there was some software support to break up the application, then we can design more energy efficient compute units.
- This would collectively work together to attack the problem by exploiting TLP (thread level parallelism)
- Look at a many core device with a lot of cores on a single piece of silicon.
- Some of the benefits are that we have distributed reliability by having multiple workers
- If one goes down, we can have the others compensate
- Scalable because you can add more workers or take away workers freely
- Talk about why there is a difference between a simple worker and a complex worker
- Doing different tasks (multitasking) or having cooperative multi-threading.

Parallel Programming

- Task of learning how to write software for many core or multiple processor systems
- Taking software and decomposing it into multiple threads
- Pushed the issue of finding multiple processes to the programmer
- Difficulties
- Difficult because you can run into race conditions
- Partitioning: how do we divide up work among individual threads
- Coordination: threads need to start on time
- Communications overhead: very different depending on how the hardware implements this solution
- Overlap communication and coordination with each other.

Amdahl's Law

- Sequential vs parallel parts
- Sequential
- Can only be run on a single thread
- The general idea is that sequential components is that they can limit the speedup.
- Image 100 processors (workers) and we wanted to achieve a 90x speedup with those 100 processors
- There is the reality of limited amounts of parallelism and we can stick this into Amdahl's Law
- This would be the time that is parallelizable divided by 100
- We want find that we need the paralellizable portion by 0.999 (sequential would be a very small fraction of the original time overall)

Scaling Example

- One of the things that is often done in parallel processing is determining how well a set of resources is scaled.
- The reason scalability is so important is because we want to continue to improve performance by adding on more processing nodes
- Sustain more nodes and add these as demand grows and there is an increased demand for the sizes of data.
- Workload where we have a sum of 10 scalars (each is a single value)
- Imagine that we follow the sum via a 10 x 10 matrix)
- See if we went from 10 to 100 processors how much speedup do we see
- Find the time it took for scaling up 100
- Single processor would do everything sequentially
- Going to be 10x the time for an add but the parallelizable portion would be knocked down to 100/10 * time for an add
- Overall, we would have 20 * time for an add = 5.5x speed up
- Sequential region is a pretty large portion of the overall time
- Time for the parallelizable version is 100T
- If we go up to 100 processors, we can see that we can knock down the time for the parallel portion to the equivalent of an add time.
- Our speedup is 10x, which is 10% of the overall potential
- Increase the processor by an overall magnitude and we only saw a doubling of performance
- Large amount of time taken by the sequential performance
- Assumed we could divide evenly
- This is somewhat of an idealized analysis

Scaling Example (cont)

- If matrix was 100 x 100, it would greatly increase the amount that could be parallelized.
- The time for the 100 x 100 matrix sum is greatly increased
- With 10 processors, we can get 9.9x speed up which is 99% of potential
- Dramatic speedup that scales well with the # of processors
- If we do a speedup with 100 processors, we get 91 which is 91% of potential

Strong vs Weak Scaling

- Strong scaling means we fix the same problem size
- If we have an image that is 1024 x 1024 scaled pixels and we want to perform a blur algorithm.
- Try to improve the speed but the goal would be to have the same size processor sent faster.
- In that case, to improve image size, we would look and see what kind of processor would be required to create strong scaling.
- We have to have this distinction to compare whether the problem size is fixed or proportional.

7_2

Shared Memory

- One type of architecture that is possible for parallel architectures is a shared memory multiprocessor (SMP)
- # of processing element, each possibly with their own cache
- Communicate via an interconnection network
- Shared memory infrastructure that allows a single address space
- Share different variable collaboratively in memory
- Memory access time
- All processors can see a uniform memory access time, or they could be distributed in a way without uniform memory access time.
- Creates heterogeneity in the process itself.
- Purely a function of how it is shaped in terms of topology

Example: Sum Reduction

- Uniform memory access (UMA)
- Sum 100,000 #'s together
- Reduction is a parallel processing term that takes many #'s and reduces it down to a single #
- Each application is going to be the same and each processor will have a unique identifier
- # will be in a range from 0-99
- What we will do is partition 1000 #'s for each processor
- Imagine we are splitting up each of the processors and we will have to eventually sum them together again
- 100 sums we will be tracking
- If you notice the code below, sum[Pn] = 0
- Each processor initializes its own sum from the array
- This will e the sum location
- Go from i = 1000 * Pn
- Array will be A_i and we will go while i = 1000 * P_n
- Defines the bounds of our 1000 # range and the sum of P_n is equal to P_n + A[i]
- We can recognize this as a bad piece of code because we are using a memory reference rather than a temporary variable
- This avoids the overhead of accessing memory each time
- We will be forgiving of poor programming style in this class
- After we form this code, we can have a sum in each element of the array's sum
- Executed in parallel across all 100 processors
- Aggregate them into a single value using divide and conquer
- Only use 50 processors to start with and divide it into 25 processors until we have finished off all of the summations
- The unfortunate nature of divide and conquer is the amount of parallelism that exists at any point in time keeps getting smaller

- You will see more work as you start the process, even though the initial amount of work is low.

Example: Sum Reduction

- half variable keeps track of the collaboration at any time
- This repeat loop goes and hits a synchronization point first
- All the threads will reach this first
- Check in for edge condition and if it is NOT evenly divisible, we will put extra work on processors.
- This is one of the processors who will NOT be collaborating in this piece of work.
- Grab the sum and combine it into my element
- How shared memory architecture works as well as having access to a common memory.

Message Passing

- The idea is that rather than everyone sharing a single memory, each processor will have its own main memory
- Each would then communicate via interconnection network
- In the shared memory example, we had to do this in one uniform access topology
- We could distribute the memory in such a way to have it all in one particular node
- Reach all of that memory and it allows a little more scaling of memory capacity
- Use it distinct from other nodes
- Lose ability to have ashamed variables int hat memory structure
- You have to then explicitly copy it across

Loosely Coupled Clusters

- A # off machines that are relatively close together
- They would have their own private memory and OS
- They could communicate via their own shared interconnect and we would want applications without a lot of sharing
- Easily partition them up and only have a certain amount of synchronization between them.
- Creates a really good, cost-effective solution of providing a parallel system
- Difficult to have interconnected pieces together
- Any type of communication that occurs will have higher overhead
- Applications that are difficult to split up would e much more difficult on a message passing interface.

Sum Reduction (Again)

- On 100 processors in a loosely coupled cluster, unlike the shared memory example, we need to distribute the #’s to each individual processor

- Distribute all 100 #'s to each individual cluster
- One processor that had all these values and they would send the interconnect 1000 #'s to each individual node in the system
- Then you would do the partial sums and you would use your own array in local memory to perform those references.
- Once you have your own individual value, we cannot do the same type of reduction since the algorithm is similar
- Split it up to 100 processors, but this time, we have to explicitly send and receive messages.

Grid Computing

- This works really well for tasks with little communication
- Embarrassingly parallel problems
- NOT a lot of communication overhead required

7_3

Multithreading

- Different architectures exploit parallelism by running separate threads
- Another alternative is to increase the # of threads that can run on a single processor
- Replicate the state with info such as registers, PC, etc.
- Provide some mechanism to switch between these mechanisms
- In an OS, we would have multiple threads that you might be selecting from execution
- Time-slicing the CPU
- Need a context switch inside.
- Considerable overhead of copying things in, copying things out
- By allowing more than 1 processor, we have increased resource utilization but we want to try to reduce the performance in this way.
- Improving utilization/throughput for the machine
- Fine-grain multithreading
- Choose which thread to execute in that cycle
- Issue to a particular function or ALU
- Any thread that sustains any stall can be immediately switched to a different thread and we can do TLP (thread level processing)
- Coarse-grain multithreading
- Only switch on long stalls
- Won't hide short stalls if you have nothing to issue

Simultaneous Multithreading

- Many may have heard this as hyper threading
- Used by Intel's Pentium 4
- Superscalar processor that can issue more than 1 instruction per cycle.
- Dynamically scheduled and we could allow multiple instructions for the same cycle.
- An example of this is the Pentium 4

- Other processes have been proposed in academic research
- The Intel Pentium 4 is the most famous example
- Two threads that could share processor resources
- By having these two threads in execution, we could use more utilization of the machine.
- Single thread degradation
- Even though all these thread's are effectively sharing and one of these threads can slow down considerably due to cache misses
- It doesn't need to interact with other threads, so we might NOT need simultaneous multithreading in this case.
- Options in the BIOS to turn this feature off.

Future of Multithreading

- Multithreading is beneficial because it allows you to switch threads while sustaining any stalls for things
- If you can switch threads based on LAX, just switch to a different thread channel!
- We no longer have to do branch predictions
- These issues can be solved via multithreading
- Increase in complexity and states required to store all those threads simultaneously
- Have a small # of threads for a complex processor or large # of threads for an easier processor

Instruction and Data Streams

- Instruction streams are shown on the lefthand side
- Can have a single instruction stream or multiple instruction streams
- We can have multiple streams of data and execute on different streams of data simultaneously
- Similar to what we covered in our pipeline data path
- We only had a single float execution to be maintained at once.
- If we were to look at multiple data streams, that would be like a miniature multi-core processor at Intel
- We have a collaborative multiple person result
- SPMD: Single Program Multiple Data
- Specialization with some reduction via an identifier to distinguish between cores on the same program

SIMD

- Single flow of instructions with multiple data streams
- Useful for a lot of parallel pieces of data
- This allows you to create something similar but we need to exploit parallelism

SIMD

- Looks at vectors of data

- Registers or operations can deal with more than just a single value
- Execute the same instructions at the same time, but they all do the same thing
- Add values together by simplifying the controls
- Makes synchronization a lot easier since they are executing the same instructions
- A lot of image processing
- Blur
- Averaging
- Do some iteration count: works really well for SIMD

Vector Processors
- They have specialized victory registers to hold larger # of elements
- The operations performed deal with all these elements in a bit-wise faction
- 32 registers with 64-element registers (64 bits still underneath)

- Pulls in each chunk and we take two of these registers and pair-wise add each element together to form a new network together.
- They key is you save on instruction bandwidths.

Example: DAXPHY (Y = a * X + y)
- Add wires to this and eventually brach and check we did it the next few times.
- Using vector support will help you to store the result
- Impact of branches can be reduced in cases
- We need to regularize expressions and do same things in term of the knifing version

Vector vs. Scalar
- Implied loop behavior and avoid checking in too many in hardware.
- Creation of vectorized code is an immense 100

7_4
History of GPUs
- Originally, the early video cards would have been video cards plugged into the motherboard slot in your processor.
- Would have memory on there for storing some version of the display buffer
- Responsible for uploading graphics content from the PCU itself.
- Originally only on higher end machines and there would be an increase in available transistors
- 3D graphics processing would soon become the norm
- Specialized with intent of handling 3D graphics tax.
- Orient them in a room according to some POV
- Check if individual vertices are visible, then convert it to pixels

- Involves fragment processing
- Creation of individual pixels that represent items on the screen
- Combine specialized hardware to be used for these particular tasks

Graphics in the System

- Increasing integration of GPU's into designs
- Upload computation from the CPU itself
- We would have increasing connections between the CPU and the individual GPU
- We can combine the CPU, PGU, and the Simpson's into

GPU Architectures

- Individual items can be readily parallelized.
- GPU tends to do the same types of computations
- Textures, stensil, etc.
- The thing that varies is the amount of data for vectorization
- Increasing combination of CPU:GPU systems
- The GPU is able to be used for a lot of the parallel workload.
- They are designed for different purposes
- Specialization of GPU results
- As we get to increasing unification of the language, the devices should be getting easier to use.

Example: NVIDIA Tesla

- Host CPU is NOT draw to scale with some bridge connecting it back to the GPU
- It may or may not have been a documented based recorder.
- May need some specialized logic for all the setup and circuitry distribution
- If you look at the designs of these, you have streaming multiprocessors that are easily scaled
- Languages can take advantage of this since they take an unknown # of SM's
- Interesting access patterns and the key thing to focus on is the SM
- The key thing are these individual, 8 SPs
- Work together as a whole
- Use a common instruction stream
- They do the same operations in a vectorized style
- If you have a stall, you can always swap out a different thread
- The warps are groups of 32 threads and they execute in SIMD fashion
- Mechanism to swap out individual warps
- They are NOT controlled by the actual compilation time
- How many elements in a warp vs how many SMs exist
- Give the GPU a little more power in its ability to handle dynamic events

Classifying GPUs

- Not every thread in a warp needs to affect the register state
- Controlled divergence in a warp means that of those 8 threads, 2 would execute code that actually affect things
- This is a pretty cool way of handling divergence but it could lead to reduced performance in these cases.
- They are in a different category from most SIMD/MIMD models
- They have either instruction-level parallelism or data-level parallelism
- It will either be static or dynamic
- The examples here are discussed here
- VLIW (very long instruction word) has been covered in class
- The superscalar had dynamic runtime behaviors in terms of extracting that parallelism
- SIMD or vector machines are generally statically discovered
- Something like a GPU like the Tesla have dynamic, discovered data level parallelism
- Can sustain branch divergence.

5_coherence
Large and Fast: Exploiting Memory Hierarchy
Cache Coherence Problem

- Memory consistency with regards to different aspects of the memory hierarchy
- Write-through vs write-back
- Deals with a multiple CPU case where you have a shared physical address space between two cores
- Each core has its own cache.
- Each of those is a distinct pipeline onto itself and they share memory.
- You would have a situation where changes in CPU A would need to be seen in CPU B
- Let's say there is some cache block X
- It is a cache block and if I read address X from CPU A, I will find that value in A's CPU cache.
- A reads 0 in memory and puts it in its cache.
- These time stamps are events are ordered by those time stamps.
- At timestamp 3, CPU A writes the value of 1 to cache block X
- CPU A's cache would contain 1
- Any subsequent reads of CPU B would find the value to be 0
- Shared physical memory address and we need to keep the caches consistent.

Coherence Defined

- Whatever reads to memory we have need to return the value.
- Assuming we have some mechanism of synchronization, we want our reads to return the most recently written value.
- Having synchronization doesn't affect cache coherence alone.
- P is able to read X and the read will return the value that was written.

- If P_1 writes X and P_2 reads X (Sufficiently later), then P_2 would have to read the value that P_1 wrote to X
- P_1 writes to X, P_2 writes to X
- There has to be some write ordering in this regard.
- The last write that is written should be the final value for X.

Cache Coherence Protocols

- Operations that caches and multiprocessors have to perform to ensure coherence
- Previous slides definition of coherence
- In a multi-processor, we have migration of data to local caches
- Reduces bandwidth for shared memory
- Don't want them all accessing the same memory
- There is going to be a replication of data shared for processor caches
- Reduces caches for individual accesses and creates a coherence problem
- This touches the tip of the iceberg on this type of thing and we have to look at the two basic mechanisms of the slide at the bottom.
- Each cache that shares particular memory has to monitor some interconnect
- This places the burden on the caches and they have to snoop on the bus to find reads and writes.
- Directory-based protocols
- Responsibility of maintaining coherence belongs to the next level of the hierarchy
- Maintains at a separate structure called the directory
- There will be some structure that maintains what the sharing status is for the blocks in the caches.
- Tracks coherence information and forces validations, updates, etc.

Invalidating Snooping Protocols

- Hidden in the programmer's perspective most of the time.
- More straight-forward of the two
- Snooping Protocols
- Cache when it can make a write to a particular block
- Some interconnect mechanism i.e. bus that everyone shares
- All the other cores and caches have this visible interconnect that can be seen and perform an invalidation
- Removes a block from my cache
- If one core writes to the notation, it would be translated on the bus and those cores would know to set the valid bit to 0 for that core.
- This is supposed to be an in-order sequence of what occurs in the time slots
- Vertically ordered
- Neither cache would have either CPU A or CPU B.
- CPU A comes along and reads this cache block

- Puts the value 0 into the cache
- CPU B comes along and reads into the cache
- Puts the value 0 into the cache
- When CPU A writes 1 to X, the bus has invalidation!
- This invalidation just needs to broadcast the address.
- Reduces the overhead.
- When CPU A's cache is modified, CPU B's cache no longer contains a copy of B.
- This relies on the fact that CPU A would have modified memory.