CS M151B Class Notes

W 1 M Lec 1-4-16

- Tradeoffs - design plate of silicon
- How do you spend your silicon budget
- Tradeoffs - performance, area budget
- How do we build devices like this smartphone, etc.
- Very low-level concepts to satisfy the school requirements: certain things need to be covered
- Homeworks - 10% of grade
- Not graded very aggressively
- Homework is used to review the material
- Midterm - 40% of grade
- Final - 50% of grade
- Practice exams to start off with.
- Avoid pattern matching and dump out the answer
- Setup will be different and emphasize learning the concept
- Cheating
- If something is suspicious, reported to higher ups.
- NOT Cheating
- Collaboration and that was fine
- Cheating
- Anything that has “Cheat” in the name
- Exams
- Open book, open notes
- No sharing stuff, no electronic stuff
- Basic plan for the class
- Performance - evaluation of circuits for design requirements
- High level metric of how to observe the value of the machine
- ET = IC * CPI * CT
- Execution Time = Instruction Count * Cycles Per Instruction * Cycle Time
- ISA - Instruction Set Architecture
- 2 Lectures worth
- ALU - Arithmetic Logic Unit
- Different tradeoffs and space of the design
- Generally trading time for area
- Single Cycle Datapath
- Fully functional machine that can execute a subset of the MIPS ISA
- All MIPS in this class
- Pipeline Datapath
- Hazards in a pipeline
- Control and data hazards
- Advanced pipelining
- Super scalar processors
- Out of order processors
- Caching/Memory
- Multiprocessors/Advanced caching

- Focus of class is microarchitecture
- Has some interface
- Assembly
- Registers
- Condition codes
- Manipulated by software side and it is not the extent of what the architecture could do
- Tradeoffs for using that kind of interface
- This set of terms makes sense.
- The design of the architecture is influenced by a lot of things
- Application side
- Peach model
- We have application software in the outer most layer
- Then we have systems software i.e. OS, compilers
- Hardware is the core
- The system software includes compilers that the hardware can manage and the OS may have drivers that use lower-level languages
- Has assembly code that manipulates those different applications.
- Hardware is responsible for executing those instructions.
- Satisfy the interface that is specified by the ISA
- Levels of Program Code
- High-level language
- Translated by a compiler into assembly
- Go down to the low-level stuff and talk about how bits flow through the pipeline
- Understanding Performance
- End-user who is application writer
- Write a different sorting algorithm
- Different approaches to make the program run faster
- The types of instructions executed and all these influence performance
- Programming language
- Can change the type of language which influences the types of instructions, choice of instructions
- SIMD - makes use of more efficient hardware and more powerful instructions
- Someone who is a processor architect or designing the memory subsystem decides how efficiently things are influenced.
- Come back through all of these and go over it in more detail
- Talk about how different individuals affect performance equation
- Application writers are the main consumers of the architecture
- Uses real-time constraints and achieve that for application workload
- Cost influences the architectural design
- Improves yield
- Capacitance
- An issue with tall, skinny wires

- If you have parallel plate looking things - parasitic capacitance/fringe capacitance (?)
- Caused problems for caching
- Not covered in this class
- Poor memory scaling
- DRAM capacity is growing tremendously
- Bring it to memory wall
- Area
- The area effect is one of the influences - silicon transistors
- Chips are transistors and wires at a high level
- Abstraction above that
- Transistors are workhorses
- Different chip generations and there is an exponential scale on the y-axis and in terms of areas on the transistor
- Transistor density has grown enormously over time.
- We have a huge transistor budget to play with.
- Scaling
- How do we continue to get performance out of these transistor chips
- Scaling Challenges
- Exponential growth of performance that we need with all the silicon we have to play with
- Golden era of 50% per year
- Incredible gains from clock scaling and architecture itself
- 2003 was a dropoff point
- They started to use multicore processors as a means of improving performance
- Manycore era - increase the number of cores in an exponential way
- Monotlithic designs and push it off to the programmers to design it
- I would much rather have single core to make it efficient
- Physical design has forced us to go a different way with this core evolution
- Clock Frequency
- Saturated and why it is difficult to scale
- This shows overtime how clock frequency has dwindled and bottled out over time.
- In terms of what we have for this silicon situation, clock frequency is not the answer.
- Power Density
- This is the Watts/cm^2: How long do we reach the power density of a nuclear reactor or the sun's surface
- Move it to a direction that is many core and more towards intelligent design
- Intel's Right Hand Turn
- The mobile designers were pushing energy per instruction
- Try to get battery life to last longer

- Power and heat have an interplay together - ability to dissipate that power
- Processor designs are shrinking in terms of the spacing together
- This is the motivation of the width shrink
- This design on the bottom -> EPI design: more efficient in terms of compute per power
- Go towards the mobile design and a lot of architectural innovation
- Had to be scaled back for more recent design
- More performance and cores
- Rise of Chip Multiprocessors
- Integrated graphics that have a GPU component used for more general computation
- Different cores that are similar and more power efficient but constrained cores
- Heterogeneity
- We specialize and have different kinds of designs within the application for different purposes
- The future may be an era of increasing specialization

Q. What is the difference between multicore and manycore?
A. We originally had Pentium and Pentium D (two cores in one).
Eventually, Intel started scaling up things relatively slowly and we wanted to make it multicore and soon we looked at 100's or 1000's of cores
Many cores is a fine distinction

- It isn't bad to have them go to many different cores and solutions don't really work afterwards.

Q. Heat is localized in the cores?
A. If you look at the processor design, we look at tradeoffs and bottlenecks. See if it kills off the performance of the application

- Eight Great Ideas
- Design for Moore's Law
- Increase in silicon real estate. How do you spend your silicon budget?
- Increase the use of uncore accelerators
- Using abstraction for the design
- Try to avoid going into transistor level as much as possible
- Learn the constraints of the physical design
- ALU deals with logic-gate level and once we get past that - we only stick with level of components in the single cycle data path
- Making the common case fast
- If we are geared toward multiplication, addition, and subtraction, why bother putting a lot of area for division?
- Parallelism
- Improve performance by overlapping latency
- This costs area because you have to have more area to run things in parallel

- This can allow you to overlap threads of execution
- Different multitasking and it lets you improve performance by parallelism
- Pipelining
- Another means of achieving parallelism is through pipelining
- Multi cycle is another data path but it is the most relevant one in this class
- Analogous to an assembly line
- Different stations where they put things together on the wheel
- Different cars are worked on at the same time
- Don't put two tire putter-onners on the same pipeline
- Prediction
- Debatable "great" idea
- Rather than necessarily knowing ahead of time, you guess ahead of time
- Take an if statement
- has a then clause, and an else clause
- It is hard to know at compile time which will get executed, no way of being 100% certain for all cases
- It may be more beneficial to guess ahead of time so it can start working on one or two paths
- Why it has to make guesses? What addresses will be used, whole values, etc.
- Hierarchy of memories
- Impossible to build memories that are both fast and high-capacity
- Some are large and slow and some are small and fast
- Dependability via redundancy
- Fault
- Tolerance - something that can go off into space and still survive after getting hit by radiation
- Being able to tolerate a lot of stress and physical load
- Manufacturing ICs
- Sliced up into circles that are wafers and it goes through processing
- There will be a designer who creates the chip in RTL (Register Transfer Level)
- The ones on the edges are squares (chips)
- Wafers get tested many times and they get sliced up and crossed out if they are defective
- Remove defects and the rest of the tested dies are adhered to individual packages that you see in chip products
- Part tester
- One part that is not shown is eFuses and it is not allowed in chicken bits.
- E-Fuses
- The cell processor is designed to work with a variety of different voltage domains
- The voltage is part of a clock generator and there is a set of E-Fuses which set the Voltage Identifier

- This patterning stage shows that the transistors vary a bit and shows the right level of power for a particular chip
- 3.2 GHz that is an unfavorable amount of deposits for the transistors
- 3.0 GHz to adjust the voltage to accommodate the higher frequency
- Distribution of transistors and save power.
- Additional customization of silicon step
- Chicken bits which are D-featuring registers and don't tweak those on your own.

Q. Why don't they start off with a square/rectangular ingot
A. Reinman is not sure?

W 1 W Lec 1-6-16

- Latency vs Throughput
- ET = IC * CPI * CT
- static vs dynamic instruction counts, the count that we are concerned with is the dynamic instruction count
- lw
- add
- sw
- bne -> runs 4000 times
- Make the common case fast by optimizing the dynamic instruction
- IC - dynamic count
- Application writer (AW)
- Different ISA's could be designed from the ground up, and we should think about extending the ISA's
- ISA can be extended or could be changed
- If you break it down into simpler instructions, you will have more instructions - micro-op
-
-
- Make more complex instructions - macro-op
- Different orders of execution and different ways of implementing this
- Implement x86 and not going to add any instructions, you can influence the dynamic instruction count.
- Machine code in this case
- There is no way for microarchitecture to influence the IC
- Physical designers will NOT be able to change the instruction count

CT - cycle time

- Latency often affects the clock - synchronizing event called the clock that the processor is going to use
- Allows us to have stable inputs that arrive

- Counterflow - moving data through different parts of the pipeline to avoid the use of a clock
- Could never get close enough to the asynchronous solution
- Asynchronous advantage is power
- The circuits that we use will influence our cycle time
- ISA designers could give complex or simple instructions that could take you more effort to implement in the functional unit. This one is always debatable.
- Compiler, PL writer do NOT have much of an influence because it is too high-level abstraction
- Application writer is ALSO too high-level abstraction

BEQ - branch if equal
- branch

I -> immediate field

Example:

LW	5	20%
SW	4	10%
ADD	4	50%
BEQ	3	20%

CPI = 5 * .2 + 4 * .1 + 4 * .5 + 3 * .2 = 4.0

If you have the same ISA and same machine, if you compare the CPI's and you have a larger CPI, what if you have less instructions?
A. If you have the same program, same exact code, same ISA, different machines
- The cycle time could vary!
- CPI could tell you since you made everything else fixed
- Overall execution time will be better if you use more efficient executing instructions

Compiler, PL writer - Decompose into different instructions

CPI can be influenced by everything which is why it is the most complicated of the three!
Q. Whenever we see CPI, can we assume it is weighted average?
A. Yes

Q. How does physical design affect CPI?
A. Take a longer wire, if the interconnect has reduced latency, it can reduce the number of cycles AND affect cycle time.
The first processor we will look at is a single cycle data path
In this case, the CPI will be fixed at 1, and the more instructions that are added, it will be affected

Q. Can CPI be < 1?

A. If you have do-nothing instruction or you throw the instruction away. You have to determine the value you are writing is already in memory.
Parallelism can be used to make sure CPI < 1. If you have a data path that supports multifunctional units

If you have two simultaneous adds, the cycles get halved to 2.
Super scaler have pipeline go down lower than 1

Example: Same program/compiler -> same instruction counter can be assumed
Two machines A and B
Implement the same ISA
New material process (physical design that allows you to drop your cycle time by 20%)

CT | 20% A -> B
v

LW 5 -> 4
SW 4 -> 3
ADD 4 -> 4
BNE 3 -> 3

CPI can always change, CT won't change if same machine

Strategy: Break down IC, CPI, and CT into different components
Tricky part: relationship between IC and CPI

Latency vs throughput

- Latency is the amount of time to do a particular task
- Throughput is the amount of work you could do in a certain amount of time

Example: Growing tomatoes

- If you grew one tomato plant and it took 4 months, eventually you would grow another one after
- The latency is 4 months
- Throughput could be 1 tomato / 4 months

Improve the latency of my cache

- How long it takes to get data out of my cache
- If you improve the latency -> it influences Cycle Time (CT) for sure -> less time to stabilize the answer
- Could influence CPI (architectural change) -> pipeline it over multiple cycles
- Cycle time can be analogous to latency
- CPI can be analogous to throughput

Tradeoff between latency and throughput

- VERY HARD to get an improvement in both
- Generally improving latency reduces throughput and vice versa
- We look at latency and see its effect
- It may be more worth it in the overall scheme of execution time
- We won't know until we see what the impact was
- Some studies try to hold the clock rate fixed
- Then we can use CPI to perform that comparison
- The more complex architecture, the more difficult it is to get within that clock cycle
- Instead of having functional units, it might make sense to have multiple cores

Aggregate throughput from all the cores together

- less dependencies
- less critical paths

W 1 Dis 1-8-16

- Probably only slides 1_3 / 1_4 matter
- Performance is most likely going to be tested
- TA's write the exams
- Define performance = 1/execution time
- X is n times faster than Y
- Performance_x / performance_y = n = Execution_time_x / Execution_time_y

cpu_Time
cpu_Time = Number_of_cpu_cycles * clock_cycle_time

- clock_cycle_time is a fixed time used to calculate CPU time
- number_of_cpu_cycles = num_of_instructions

Problem 1: Consider the performance of a processor with a clock frequency of 2.4 GHz on a suite of benchmarks

Problem 2: When a processor executes program PA, the execution time is 3.8 seconds and the CPI is 2.9. When the same processor executes program PB, the execution time is 8.5 seconds and the CPI is 3.7. A typical workload consists of executing PA, then executing PB. What is the CPI for this typical workload.

Problem 3: Consider a processor that achieves an overall average CPI of 3.6 when executing a program P1. 23% of the instructions of P1 perform floating point operations. For these floating point instructions, the average CPI is 5. By improving the design of the floating point ALU, the average CPI for floating point instructions is reduced from 5 to 3.
A) What is the overall average CPI for the improved processor?
B) If the original processor executed P1 in 190 seconds, how long will it take the improved processor to execute P1.

MIPS = meaningless indicator of processor speed

W 2 M Lec 1-11-16

- ISA will write data into a location in memory
- Model the microarchitecture's participation in memory
- Located separately for now and look for better techniques to integrate them on a die
- Some latency here
- Think about it as the microprocessor which has to grab something from memory
- View of memory with some reserved segment at the bottom
- Global data is persistent throughout the program
- Stack is used for function calls and place a frame
- Heap is used for dynamic allocation (malloc, calloc)
- Text segment deals with things from the instructions
- Used to tell useful work
- Instructions like this for MIPS
- add, lw
- Stored in text segment of memory
- The processor has one stage of IF (Instruction fetch)
- Goes to the text segment of memory, grab an instruction out, and that is the instruction we will execute next
- In x86, how many bits of memory do I have to grab?
- ID (instruction decode)
- Determines if you access memory or not, what variables you use, etc.
- RO (read operands)
- If you have an add instruction, you have two operands
- RO determines the # of operands you have
- EX (execute)
- Do whatever the operation is supposed to perform
- WB (write back)
- If you have an output like store instruction or add instruction
- This is the general flow the processor has to take
- Some steps can be skipped and this is just a general stage (not everything uses all five of these steps)
- Ignore caching for now (assume that it is just stored in memory)
- Programmer writes in a high level language (machine expects something in the form of that ISA)
- Instructions are stored in memory! (REMEMBER THIS)
- Global
- int x;
- Not limited to some procedure scope, it is global
- ISA - Instruction Set Architecture
- Specifies how the machine should behave, what type of operands and operations should we support

RF vs MEM

- z = x + y
- Operands
- How many?
- Where?
- How specified?
- Components that are operated on
- x, y, z
- Source
- x, y
- Destination
- z
- Operations
- How many?/which?
- Can impact a # of things, a tradeoff we should embrace
- CISC vs RISC
- Allows compiler to enable compact code
- x += y;
- Less memory is needed since it has two operands total
- Put operands in registers
- registers are small, fast memory structures
- Held in the processor
- Cache differs from a register file in that a cache is bigger than a register file, so it is slower
- Compiler does NOT always know about the cache
- Cell processor does have scratchpad memory and thus the programmer needs to know about migration into those.
- We do NOT plan for the cache
- The programmer can plan on optimizing and breaking things into chunks
- RF (register file) is part of the ISA
- Part of the agreement
- Instruction itself can include a value and two of the instruction types contain values in them
- Zero register
- Comparison in x86
- jal
- Using ra
- Case of MIPS with implicit instructions

RISC vs CISC

- Reduced Instruction Set Computer (RISC) vs Complex Instruction Set Computer (CISC)
- We wanted code that would fit compactly into memory and would NOT take up a lot of space

- How many instructions, you would want few instructions in terms of the ISA
- We want more instruction types but fewer instructions in your memory
- Modularize things
- Going towards CISC
- You have different operations that are very complex types of instructions (this compactly represents it in the ISA)
- More instruction types means less instructions in memory
- CISC has increased instruction types because of the memory requirements we have to deal with
- In terms of instructions in memory, there are two kinds of alternatives with how we can represent instructions
- Fixed length
- Instructions have a fixed length and do NOT vary
- Decoding is much easier
- Variable length
- Instructions vary based on the size of instructions
- More space-efficient
- Auto increment i.e. ++
- x + y + z = a: Requires a lot more space
- In line with having not a lot of memory
- You figure out where you are in the cache line
- Figure out where your instructions start and end

RISC

- Embraces philosophy that instead of a lot of instruction types, we have smaller instruction types
- Simplifies the hardware design and puts burden on sophisticated compilers
- ARM
- Can handle 16-bit and 32-bit instructions
- Dealing with two types -> 16-bit vs 32-bit can help deal with a lot of things that aren't possible
- Simpler hardware design -> It is more limited to common modules that can be reused over and over rather than creating operations that perform complex formats at one time
- +, -, * rather than *+*, /-+, etc
- With a smaller set of operations, hardware is required to become simpler
- Less things to implement so by nature, it becomes simpler
- Memory vs hardware that implements any particular instruction in the ISA
- Pay for it with a little more complexity
- Get around some of the bad parts of small #'s of instructions
- If you have a high-level language, it adds complexity in that you have to do more sophisticated register coloring
- More aggressive compiler optimization is a big advantage
- In terms of MIPS, which is the candidate ISA

- Has a reduced set of instructions
- R-type
- I-type
- J-type
- Each format is 32-bit, so there are 32-bits of space for each of these instructions
- Program Counter (PC) tells which part of memory that you want
- Go to next 4 bytes of memory after the PC that you accessed
- Does NOT have to figure out how many bytes of instruction, just figure out the next sequential instruction
- Operands
- If I have more operands, you could support instructions like the address code
- Store things like two address code
- Fewer operands where we compact space and not store as much
- Store fewer operands to have more compact instructions
- CISC
- Has lower # of operands
- RISC
- Has larger # of operands
- You have fixed length instruction sets, so if you have something too large, you may get to the point where the fixed length size has to increase
- Leads to bloating and waste of space
- It often cannot grow that much in terms of # of operands
- Large # of operands where you specify 4 input values
- Not as clear of a tradeoff
- Benefits more from a low # of operands because it reduces complexity
- Register-Memory machine
- Operands can be either in a register or memory
- If instruction is performing an add, it can come from the source or from registers
- Try to do everything in one shot
- Makes instructions more complex for the architecture to implement
- More powerful and more complexed
- Found in CISC because it is more complex
- Execute them as opposed to a simpler design
- Load/Store Machine
- Operate on registers only
- Takes values from memory and acts on registers
- Load/store cannot do memory add unlike register-memory machines
- Data movement instructions
- Solely exist to place things in registers
- Arithmetic operations cannot go to memory
- More reduced in what you can do
- Found in RISC because it is more reduced
- A + B are stored in memory

- C = A + B
- Look at notes!
- With RISC, you can handle only a certain # of operands
- Does NOT deal with space compaction well

Look at different types of addressing

- Imm -> Immediate addressing
- Using a literal value from the instruction
- Use the same register specifier depending on the value
- Flexibility of usage is a big difference here

LW and SW

- Stumbling points
- Data movement operation for MIPS
- Take values from memory and store them into memory
- Load will load a register (LW)
- Loads from memory
- R[RT] = M[R[RS] + SE[I]]
- Don't use that value as input to the register file
- To get value that we want, use it as an index into memory
- Huge sentence to parse
- Hopefully, this syntax makes sense after a while when we implement the data path
- Whatever 6 bits would be the instruction

SW

- First reads from R[RT] and then stores something

J-Type

- Has 6 bit op code
- Has 26 bit immediate
- In the jump instruction, you can tell if it is a jump by the op code
- Changes the PC
- Unconditional branch
- Like a goto statement in certain programming languages
- Changes from the original value
- Optimization you can make is that you have to encode 32 bits
- Take the current PC: 32-bit value
- 32 registers and you have to either grow the size of the register to hold a longer instruction or you satisfy something else
- Tradeoffs are how much of 32 bits we use and what we use for various things

W 2 W Lec 1-13-16

- R-types all share a common op code
- You know you will have an R-type instruction

- R[RD] is the destination register
- R[RS] is the register specifier
- It is the address in the register file
- Place it in the contents pointed to by RD
- LW
- I-type instruction
- Base + Displacement
- SE(): Sign Extension, immediate is 16-bit, you need to pad on to get 32-bits
- Two different functions that are getting performed
- Fields of 32-bit instruction are interpreted in the same way
- All this data is going to be flowing around the processor
- Used based on the type of instruction that it is
- BEQ (branch equal)
- Data that gets driven out is R[RS] and R[RT]
- Base + Displacement addressing -> displacement is of byte displacement, could have significant bits on the lower bits
- Load byte and store bytes
- Does NOT have to start at a lower boundary
- An implicit shifting that is being done to it: word offset rather than a byte offset
- Deal with overflow detection
- Load instruction is a separate issue
- This is a linker and there is some target address
- If you shrink opcode down, is there a benefit to restricting the set of instructions?
- Starting with J, we have op code and an immediate
- LW and SW
- It may be that IMM helps reduce instruction count, it can look at a larger displacement
- rs and rt
- rs and rt can be bigger: more registers
- If an RF increases, how does this benefit?
- Not as much spilling, deal with a data migration
- Keep track of more values at once in the faster memory
- In MIPS, r-types can NOT access memory directly
- The more registers you have, the less loading and storing you have to do
- Exchanging values to and from memory: **spilling**
- Creates more loads and possibly more stores if the values are changing
- Could help to drop to instruction count and minimize loads and stores
- Physical impact so longer latency
- How does this translate?
- Cycle time is improved
- Takes longer to improve cycle time
- Cycles are usually a mesh of wire
- The more the size of the structure grows, the larger the entry gets

- Longer wires increase the delay for the register file
- LW
- Generally the worst performance
- It is involved in so many components because it touches the ALU and immediate
- If we increase register file size, this means fewer loads and possibly fewer stores
- This would drop instruction count and the CPI is the weighted average and cycles it takes for each instruction
- Changing instruction count
- May drop CPI since it affects the types of instructions that get received
- Disk access
- Is this under virtual memory?
- What if we didn't change the register file?
- For i and j, the Instruction Count might drop because of less branching and less jumping
- Usually better than BEQ
- In this case, if we look at IMM increase, IC can drop
- You may have to jump less far OR you jump further but jump less often
- Cycle time
- This could have an impact on cycle time
- Register reads are a lot more costly than sign extension
- It is more than likely that it won't change, there could be some change
- Likely that CT will stay the same
- Just increasing the immediate won't change much if the instruction size does NOT grow that much
- Likely be dwarfed by the bigger instructions
- The new mix of up and down can change CPI by both increasing and reducing
- The beq would piggy back across another jump
- Add and store reductions
- Your CPI could increase potentially
- Can a cycle time take longer based on the input?
- Yes, but in the general case, the processor does NOT always look at the size of the input
- Things move in stages one after the other, if you have instructions that try to pass each other in the pipeline, there will be increased heterogeneity
- Larger register file will have more delay, and there is a way of dropping cycle time
- A larger structure will have more delay and it could reduce the cycle time
- The question mark here is that we may NOT have enough register spilling to get rid of it.
- It may or may not have the slack to deal with register links
- Given more complete specs, this increase in register file size can give us a more accurate representation of reduction.
- Shift Amount - could reduce shift amount

- If you had R-type op code reduced by certain amount, you can induce that in rs, rt, and rd
- Could have same set of challenges for the I-type
- The function fields could have more sophisticated R-type instructions
- Could have been performed by a subroutine call
- Reduce instruction count by reducing the function size
- CISC vs RISC debate
- Add complexity and drop the IC
- You could have longer cycle operations with the same cycle time
- CPI -> more R-types in general
- It could influence your CPI
- Depends on how heavily it is weighted towards load words
- If you increase cycles of R-time, then your CPI will only increase
- It is possible that it increases and you don't really know where it is going to go
- Could go up or down
- ALU
- When you look at function field, it is going to influence the ALU and tell what operation it will perform
- Could cause a longer delay, which impacts cycle time
- Why don't we shorten the instruction length (fixed length instruction)?
- Less memory is used overall
- You are getting less bits out since you have a shorter instruction
- When we get to caching with a small fast memory that has a working set of instructions
- This reduces the instruction footprint
- A smaller cache could be used to improve the cycle time
- Does CPI get affected if you have a shorter instruction length?
- Assuming they can do an instruction fetch in a single cycle, it is possible that it could take less time, so CPI could be **fewer cycles per instruction**
- This is if the cache helps reduce the cycle time
- Instruction Count affected?
- No, count is independent of the length (if it is shorter, it can still be the same # of instructions!)
- More operations instead of less operations
- By having more instructions, you have more functionality and you have to do things that would NOT require as many of these simpler instructions
- If you do NOT have a BNE (branch not equal), you have to do some preprocessing
- All this can be combinations of instructions if you have another instruction available to you
- More instructions -> more accurately capture what you want to do
- All at the cost of implementing these new instructions
- Still want to stay fixed length and you want to make these components smaller
- Potentially more jumps because you have to piggy back more times

- Be careful about requiring multiple jumps
- More adds and more stores
- These cannot branch as far
- Claiming immediate space will have the opposite effect
- Shrinking register size
- Cycle time could be better, but more spilling
- Less latency since cycle time is better
- IC goes up since more spilling
- Have a long jump instruction, shift its immediate so it can cover a greater space
- You could never cover the lower 4 bits
- This could mitigate the instruction count decrease
- This is a fantastic tradeoff
- The imprecision could require a decrease in instruction count
- Up to the design decisions
- Is it possible that it increases, yes or no?
- These can be open-ended questions as long as you can reasonably explain things
- Jumps can go backwards because they replace what is in the Program Count (PC) -> i.e. if you jump to all 0's from a different value

Fixed-length vs Variable-length

- Make a long jump by making a longer instruction overall
- Short jump and not having a full 32-bit instruction
- Tradeoff in this space of variable length?
- It could save memory and there is potential for reduced CPI
- The problem is that decode becomes more complex
- I don't know now that PC = PC + 4
- Less control over how much the Program Counter jumps
- This creates a critical path problem
- Too many instruction set sizes can hurt cycle time since it now impacts the critical path of the processor
- Much more complicated path that has to be done
- Fixed length is better in that you have more control since you know you will jump PC = PC + 4!
- Variable length saves space but now you have to pay closer attention to how much you jump in your addressing on your PC!
- Instructions can preload and you have to use possibly two instructions to create a 32-bit value

Single cycle data path

- Design requirement is that we come up with a new instruction
- Take that instruction and extend the data path
- Include the top 4 instructions and whatever mix of R-type we have
- Data path on an exam that is a full specification
- Control for those 4

- Can we add a new instruction type for the data path we controlled?
- How can we specify this instruction?
- Can make things worse on the implementation side

What is x86?

- CISC ISA
- Heinously complex
- Why is Intel sticking with x86?
- Legacy and backwards compatibility
- How does Intel maintain dominance in market?
- Converts CISC and RISC
- Decoding is still extremely complicated
- Trace cache and dealing with micro-op form
- Power efficiency
- Trace cache where they could save the conversion and NOT do it so many times
- Crusoe processor
- Terrible and not well-known because it was bad
- Tried to convert x86 into an ISA
- Awesome power savings but terrible performance
- Absolutely tanks performance (downsides of tradeoffs)
- Software efforts of this have not been successful in improving performance while saving power

W 2 Dis 1-15-16

Basic MIPS ISA Review

- 32 General-purporse registers + PC (32 bits)
- $0: Always zero
- load-store architecture: All operators of instructions
- MIPS
- load-store architecture (all operators of instructions need to be registers)
- big endian
- byte-addressable
- All instructions 32-bits
- Instructions format
- R-type
- I-type
- J-type
- Look at handwritten notes

Arithmetic and logical instruction

- How do we distinguish between the different R-types?
- Computer first reads opcode.
- All of the opcodes are 6 bits
- This is used to distinguish between R-type, I-type, and J-type commands

- You want the format of the instructions to be the same, but you want them to do different things
- All these instructions are going to the ALU, which does the execution
- Opcode 0 tells the processor that all the instructions will go to the ALU
- Control logic is almost the same but there is a slight difference for each
- ALU can do add, sub for different functional codes

load-store instruction

W 3 W Lec 1-20-16
32-bit ALU

- Left side is a 1-bit ALU
- a and b are coming in
- A and B will have n 1-bit components put into n amount of ALU's
- These are going to 3 different components right now
- AND gate such as performing an AND operation
- OR gate in the middle
- Full adder at the bottom
- Ripple Carry (RC): take CarryOut of N-1 bits, you can chain it to the next set of instructions
- Multiplexer: Selects one of the results as output
- There is going to be some result that comes out
- Assume A and B are stable at time i
- After all these permutations, this result is going to be stable, but it's going to oscillate
- Compute the CarryOut at some time stamp and see if it stabilizes at some correct value
- Looking at adder latency: ripple carry form
- If every logic gate has delay t, at time 0, what would be the stability of this line?
- Time T because every AND gate took time T
- Add 1 to the entire chain and gives you the subtraction that you want
- With the MUX and the inverter, this does NOT mean the Cin of every chain has to be 1
- Overflow
- Look at most significant bit and compare the Cin and the Cout to determine if there is overflow
- If you have 1 propagating across, you want to make sure there is enough space to represent that.
- Understand that there are other things being measured that makes sense here
- Zero Detection
- Looks at every output bit of the ALU and says if any of them are 1, then we will have a 0
- Gigantic NOR gate
- This could be useful for branching i.e. beq

Set-On-Less-Than (SLT)

- R-type instruction that takes in 2 register inputs and outputs 1 register
- RS < RT
- Dump out a 1
- RS > RT
- Dump out a 0
- Use the Most Significant Bit!

SLT Implementation

- This modification to every ALU
- All but MSB
- Most Significant Bit
- What should Set hold if you can set subtraction for ALU
- Set will hold this value and we have to make sure we do subtraction as our goal
- We have a hard-coded 0 going into the Less input:b Goes straight to that part
- We will propagate 0 to the output and this goes for R_1 … R_31 = 0
- Most Significant Bit -> goes to Set Input
- Set of MSB links back to the Less of LSB: It is sort of like a cycle and it provides constant feedback into the LSB

Q. a_0 and b_0 … a_31 and b_31, will you want to have all these values going in?
A. Regardless of what you choose for the operation, you are still churning out the AND's and the Or's. It does NOT affect what happens internally

- Choose AND/OR operation with SLT being 3

Can We Make a Faster Adder?

- Ripple-carry means you serially propagate Cin and Cout
- If you look at the formation of this sum, it becomes less critical than the propagation of the carry chain
- You can hopefully do all of these sums more in parallel
- Waiting on Cin in order to do the sum
- Maybe we can do something with a and b ahead of time to reduce the amount of logic in the critical path
- generate: if both are 1, if you add those numbers together, you would get a carry for sure
- propagate: Cout = Cin
- kill: Cout = 0

If you have a generate, you definitely have a carry
$C_1 = G_0 + C_0 * P_0$

- Area cost because you have to add more modules
- There is a speed up but at the expense of area!

Carry Look Ahead

- For CLA, we can add more logic to the side and see that the Fan-In has increased significantly
- Different approaches for larger and larger circuits

Hierarchical CLA

- You can add more 4-bit look ahead logic
- With the same assumption from before, the generates and propagates are NOT related!
- They will all be at Time T and the AND gate would be there
- As a whole, we figured that the CarryOut has a value of 5T and the sum will be 8T
- Sum is 8T because we had 5T coming in from the CarryIn and we wanted it to go to a 4-bit Carry Lookahead Adder

W 3 Dis 1-22-16

Delay Analysis on Adders

- Give more complicated examples
- Assume the delay for the Gate is kT if fan-in is k. 1-bit Ripple Adder
- How do we get CarryOut?
- (b * CarryIn) + (a * CarryIn) + (a * b)
- 2T to get the products, 3T to get the sum
- Total delay is 2T + 3T = 5T
- How do we get the Sum?
- A Xor B Xor CarryIn
- Can use two XOR gates or one three-input XOR gate

16-bit ripple adder

- What's the delay of the following?
- result 15 : 15 * 5T + 3T = 78T
- cout: 16 * 5T = 80T

4-bit Carry Look Ahead Adder (CLA)

- Review the notes

16-bit Ripple Adder built with 4 CLA Adders

- Critical path is from the 5-input AND gate
- Review the notes

16-bit HCLA (Hierarchical Carry Look Ahead)

- P_1 and G_1 are independent of the carries (so just look at a and b). These will share the values of P_0 and G_0

Write a piece of MIPS code that will yield different results for different endianness?

W 4 M Lec 1-25-16

- CSA (Carry Select Adder): You can run these in parallel which is the big advantage of using this type of design.
- You can construct a 16-bit adder equivalent by combining 8-bit, 4-bit, 2-bit, and 1-bit adders
- Test question examples
- Equations for delay i.e. table lookup such as Fan-In, diagrams
- All of these things should be components for labels that we have
- Find examples to give partial credit and it will be based on some delay number for one or more designs
- Similar to the one on the HW
- Critical Path is the one with the longest time latency, so you always look down this path
- Each OR gate has an input of T, so sum them all up!
- Break down these Carry Lookahead problems into different pieces
- Most common mistake is someone figures out the C_3 and uses that as the Sum
- These paths for a hierarchical version has to look at the highest path and go back to the original sum.
- You still have this doubling of area
- The gap you have will allow you to put more levels on top
- There could be a point later on when ripple carrying is enough

Multiplication

- Adds and shifts
- You will be adding, but you may NOT be writing (it is not guaranteed)
- Booth's Algorithm
- Replaces numerous adds with more frequent subtractions if necessary

Review Data Paths

- Watch the video for Wednesdays

W 4 W Lec 1-27-16

- PC points to some location and grabs it from memory
- IF (Instruction fetch): Instruction memory
- ID (decode): Determine it is a load and read its registers
- EX: ALU -> We have an immediate, so we have an op code, rs, rt, and Imm. We want to get value of rs and add that to the sign extended IMM
- MEM: Data Memory
- WB: Reg Write

Load word does all of these operations

Store word would do all of the above except WB (Register Write)
Add would skip MEM but perform the rest of the operations

Different functionality for each component

- Data Memory contains the heap, stack, etc.
- Right now we are thinking about them as memories that exist
- Both memories share an ADDR specified to memory as an input
- For Instruction Memory, you drive out 4 bytes of information
- Eventually, you will put out the PC (Program Counter)
- 32 bit input is NOT necessarily the same the 32 bit output
- You are reading the value at an address, it drives out a memory location
- Data Memory does NOT require arithmetic operations i.e. add
- Load would want to read from DM, and store would want to write to DM
- MEMREAD and MEMWRITE: At most one of these two should be 1, but both could be 0. It can NEVER be both 1
- Either we drive out a location from memory on a load, or we store a value into memory at that address depending on which is easier
- We assume we have 32 bit VALUES and 32 bit addresses coming in
- It is slower to read from memory than it is to grab it from a register!
- When you eventually get to caches, we don't want the memory hierarchy to pull in every chunk.
- We want to throttle when we access memory and when we don't.

RF: Each will input will have 5 bits

- We've got multiple instructions, but the Control lines determine what is being use and what is NOT used
- There will be a 32-bit value coming in but RegWrite is the flag that determines if a write can be performed or not
- Register file is controlled by the software side, unlike the cache, so you won't have any irregular misses

ALU: Has two inputs x and y coming in as 32-bits, we want to get it into 32-bit form

- We will have a MUX that specifies what operations will be dumped out
- X OP Y is the actual 32-bit output

We have gone through instruction fetch and instruction decode

- For an I-type like load and store, we have to take the Imm and sign extend it.

Data path With Control

- If 0 bit is one, it tells you if you should branch
- Conditionally apply the BEQ statement: if(R[RS] == R[RT]) PC = PC + 4 + SES(I)
- Test Question: Start with this data path and then extend it afterward

ALU Control

- ALU can perform a number of functions
- AND
- OR
- add

- subtract
- set-on-less-than
- NOR
- These are 4 bit controls
- Select on output depending on the control operation
- There will be an output from the control called ALUOp that goes into the ALU CTRL
- This is a 2-bit signal and it indicates whether you have a LW, SW, BEQ, or an R-type
- BEQ gets 01 and R-type gets 10
- If we get ALUOp 01, we always tell it to subtract
funct
- What do all the X's mean
- This is the last 6 bits of the instruction and for loads and stores, and you don't want to use the fields, so all the X's means don't care (they don't matter)
- Load word and store word share ALUOp because they both use an add
- There are 4 possible options and a limited set for our ALU
R-type
- There are different functions that we support
- add
- subtract
- AND
- OR
- set-on-less-than
- These correspond to specific ALU controls and the R-type just tells it to consult the function field and this will tell you what the mapping should be.
- ALU control dumps things based on the instruction
- ALU comes from the main control unit
- RegDst: Controls which register destination you want. Picks the C port file for usage in this scenario
- Try to trace through and see how information flows throughout the structure
- Try to rebuild it on your own from the data path.

- The address will always include the upper 5 bits and it would be a nasty setting up process
- Load words would be broken on a RegDst
- You could try to do new instructions
- LAW: I-type
- In terms of efficiency, as long as you don't crazy about adding things that don't make sense, you will be fine
- Just make sure NOT to write and corrupt the file
- Use the port on register file twice and there are definitely things that are taboo to this implementation
- It doesn't need to be the most complicated implementation but don't use the port twice

- Don’t write to registers where it is NOT supposed to.
- As far as simple efficiency, it doesn’t have to be the most optimal, BUT make sure it is readable

LAW - I-Type

- There should be three major changes we need to make to get this to work
- R[RT] and allow memory output to be input into the ALU
- This comes out of the ALU and it is RegDst set up in the right way
- When you find the new instruction, find what is on the data path and what is NOT
- Try not to break current instructions
- Make sure every other instruction works properly so that this makes sense
- Fix R[RT] so it can go into memory
- Add another MUX which has a 0 and a 1, and it selects whether you want to use the ALU or the register file RT

Reroute it to this port here, and this output from data memory can be tapped

- Reinmann will give out the drawing and you have to extend the data path

LAW - I-Type

- R[RS] = M[R[RT]] + SE(IMM)
- Figure out how to reroute it to the memory address that you want
- Previously, you could have only used each port once, so you had to be smart with how you built your design
- That length of time is how long we wait for sign extension
- Make sure we tune our clock to suit this particular data path.
- It is a question of which stabilizes first and the order may change in this case.
- Unofficial homework
- Try to think about how to do the zero

Office Hours

0. Performance
0. ISA Tradeoffs
0. ALU
0. Look into lw, sw, MIPS language,
0. Datapath

It is possible for C_15 to be faster than C_16 or vice versa

W 4 Dis 1-29-16

0. Datapath of Single Cycle Implementation

How Control Signals Work

0. RegDst: Controls the input to the write register here

MemRead: You want the clock cycle time to be large enough so that the rest of the clock time is not wasted

- Don't waste time setting the clock cycle to 1
- It is don't cares on MemRead for sw, beq, and R-type
- Depends on the implementation which could be like z or a random value
- Can be 0 or don't cares

MemtoReg: Controls what we write to data

- Difference between MemtoReg and Memread is that MemtoReg is a signal to a register, while MemtoRig is a signal to memory
- In this case, you need 2 bits and you only have one bit for this. This simplifies the future design and you can make them the same.

ALUSrc

- MUX performs a check and we use the Zero ALU to subtract if necessary
- For R-type, we want ALU to perform the operation, which ends up as 0

Read up on how each instruction works on the data path

- lw rt rs(imm)

Read data 2 is the value of rt, and the MemWrite is 1, so it writes the value into memory

- Operand is PC + 4

R-type:

- Add rs rt rd

Q. What is the advantage of single cycle instruction?
A. Easy to learn and implement, but there is no main advantage

Q. Is it a realistic assumption that data memory access is a single cycle?
A. No, if it is NOT cached, it can take hundreds of cycles.

If you have a motherboard and memory slot, you won't see any components in the CPU. He is accessing the cache faster in memory. The second reason is that for the register file it is implementing using Flip-Flops, which is kind of fast.

- Assume you have a fault, how would this affect the execution of the CPU?
- Write to the RT instead of the RD
- How would this affect the execution of the CPU here?
- What time of instruction would it affect?
- lw -> it affects it because you write to the wrong destination register, which probably isn't the one you want.

W 5 M Lec 2-1-16

- Pay attention to the examples he has done in class
- The test will be open-note, open-book
- As long as you have been practicing the problems, you will be good!
- The first two should be relatively fast.
- Third and fourth are more challenging
- Single cycle data path is harder.
- Try these practice problems
- If you get it, it is a matter of engineering the answer
- It isn't that simple to figure it out on the fly on the exam
- Won't ask what the Opcodes or the function numbers are.
- He would give you type, data path, both controllers, and Reinmann would ask you to extend it.

SAW

- RegDst is used to choose between RT or RD
- These are your two possible destination addresses
- Those are the two options you have and choose it as 00, which forces it to do addition
- Instruction[25-21] is first RS
- We may need to have something else read out that is NOT RS or RT
- These two specifiers are 5 bit addressees that tell us which register we want to read out

CMOV

- Zero flag is conditioned on the operation, NOT the result
- Get inputs set up to have 0 and RS coming into ALU
- Have R[RT] coming in as write data
- Get the conditional write into R[RD] working
- R-types assume the same OPCode and we see this as a cleaner way
- Doing this in the main controller avoids this issue of the MUX
- We still need some kind of MUX, and we don't want to add any other thing.
- The ALU has its op coming in: output of ALU controller
- CMOVC could have been explicit
- Conditionals using MUX's

BEQM

- We don't need a data path extension, we just need to check the control
- For the PC, we have to get the memory contents at R[$t0]

WTF

- You always want SWTF to be 0 in this case
- Write to the PC a certain value and also write to the register file
- Connect it to multiple places

Timing

- Worst case path will set the clock cycle
- Execution time will refer to single cycle data path or a mythic cycle data path and we have to interpret based on the equation
- Find CPI of a certain path?

W 5 Dis 2-5-16

- Open book, open notes
- Review examples that the Professor did in the class
- Understand the sample exam
- Will teach the most important stuff in the class will be emphasized.

Performance: Review this online
Adder
Single cycle data path

- Exam should be slightly easier than the practice exam

W 5 Dis (Uen-Tao) 2-5-16

- 4 Topics/Question:
- ISA Trade-offs
- Execution Time Calculation (ET = IC * CPI * CT)
- Adder Timing
- Single cycle Datapath and Extending it
- This test is not terribly difficult. However, it is a bit long.

Midterm

- If you are familiar with the homework assignments, you shouldn't see anything totally new.
- Advice:
- Revisit/redo the homework questions, sample problems, and various class examples. All of them!
- Get comfortable with the question types that you have faced with so far, specifically the previous midterm ones and the ones from Reinmann.
- Go over these things and get to a point where you can quickly make accurate, correct distinctions.

Midterm Review: Trade-offs

- You have System A and it performs like this. Change it to System B.
- What is the effect on performance?
- You need to focus on the change we made and see how it affects CPI and clock time.
- There are two different flavors:
- Software
- Hardware
- Whether or not instruction count goes up or down is vague. Not enough information to tell if it increases or decrease.

- **Software can affect IC and CPI**
- IC is of a particular executable you are dealing with.
- Of course, the instruction count will change.
- CPI: More complicated instructions might take more cycles.
- If you have multiple versions it will be affected. Term for a particular instruction class.
- CPI is a weighted average over all of the instructions
- Every time you should composition, you will likely change the CPI
- **Hardware can affect CPI and CT**
- It can change CT when you decide the hardware and make a fixed clock path.
- Part of the hardware is defined in the clock time.
- The complexity of the hardware affects the clock time as well.
- If you want a particular task to be done, it has to be tuned to the critical path.
- If the critical path becomes longer, the clock time might have to change in a particular time.
- CPI is a weighted average, so how exactly can it change that bit.
- Depending on how you orient the hardware, you can reduce the # of cycles based on instruction type.
- In the pipeline, we have split up and divided the single cycle data path into 5 different stages
- 5 cycles that changes with CPI overall. Change how long each particular instruction takes

Ex.

- Say you have the ALU of a process and it is changed from a pure Ripple Carry Adder (Ripple Carry Adder) to an Hierarchical Carry Lookahead Adder (HCLA), where we assume a non-specific data path.
- What effect could this have on IC, CPI, CT?
- Hardware change that cannot affect software (cannot affect IC)
- Could potentially have a smaller critical path.
- Could possibly affect CPI.
- CT: possibly decrease. The hardware is simplified
- CPI: possibly decrease. The hardware is simplified.
- IC will not be affected. The different adder does NOT have an impact on what instructions get executed.
- There is no guarantee that any of these factors can change or stay the same if our information is limited.
- Is every block a cycle?
- Analogous into splitting it up into stages.
- This is for one instruction.
- Which (of IC, CT, and CPI) can be changed by the ISA's influence?
- IC and CPI are software influenced, while CPI and CT are hardware influenced
- ISA can change many things

- # of instructions defined
- Instruction encodings
- # of registers
- The ISA can change everything.
- The ISA exists as an interface between the hardware and the software.
- ISA can make changes and it will be defined as both hardware and software.
- Do specific ISA's only work on specific hardwares.
- It has to be a machine-designed for a particular ISA.
- This is why the ISA can change the clock time.
- Define in the assembly language that you want hardware to implement all of these things.
- Hardware implements the ISA -> a set of rules
- The processor has to be able to implement that.
- MIPS ISA can have one processor that does it one way, and a different processor doing a different thing
- CPI could be affected by software
- CPI is the average over all the instructions executed
- CPI could be affected by hardware
- Cycle is an average over executed instructions.
- Complicating hardware and adding another stage to complete would increase cycle time.

Tradeoffs Example Question

- Consider J-Type instructions
- [6 bits] [26 bits]
- oppcode addr
- What if we change it so we only have 25 bits for add. What effect could this have?
- MIPS is fixed length, so this means the opcode is 7 bits instead.
- More instructions defined for the opcode
- You cannot jump as far
- This is the new format
- [7 bits] [25 bits]
- IC -> If you rely on instructions to let you jump as far, you have to jump twice for a faraway address.
- Usually, you have to load first, or you jump twice.
- Pay the cost of jumping multiple instructions.
- You can answer both increase or decrease but you have to understand why it's correct
- Now you have 2^7 => 128 j-type instructions
- When you have more instructions defined by the ISA, you use fewer instructions
- If we are given the freedom, try to give reasons for both sides!
- This is to showcase your knowledge.
- CPI?

- It could go up since you have more complicated instructions and we are using these more complicated instructions.
- This is the most vague one.
- If we don't change the instruction count and we don't make use of the instructions, it won't change.
- Since we don't know the CPI of each instruction type, we can't really make any judgments.
- Increasing the opcode meant I have 120 instructions, will this affect hardware?
- Decoding would be harder since there are more possible instructions to handle.
- You have to download new hardware to handle add and load simultaneously.
- The new instructions can inc. complexity of the microarchitecture.
- CT?
- If we split the bottleneck into two cycles, the CT may have decreased.
- If we made a more complicated cycle, the CT may have increased.
- Given the proportion of instructions, you could give more precise decisions. Let's you make a more concrete decision.
- Jump instruction could be more expensive and this could increase CPI.
- If the CPI is lower than the average CPI, then we might want to make a more careful consideration.
- Increasing complexity of the hardware means you can no longer reasonably implement everything into a single cycle.
- Each cycle will be less than the original.
- It is hard to say for sure.

Midterm Review: Extending a Datapath for ???

- Let's define a nonsensical instruction for testing purposes
- PC = R[M[R[rs]][4:0]]
- This is a definitely unusable memory branch, or "dumb" for short.
- This is a dumb instruction.
- Can be an R-type or I-type, but let's go with I-Type to avoid the whole opcode issue.
- What connections need to be made?

Synchronization purposes -> we don't want to do this to maintain stability and correctness

- We will have a multiplexer that selects between the original RT and it comes from the data memory.
- You can get this into PC and we can use a MUX
- MUX will handle all this stuff
- Handle either PC + 4 + SES(I) or PC + 4 [Your choice!]
- Read register 1 was RS, and what we want to do is read from register again.

- We want to first read from RS, then use that value as an address into memory.
- Use that value as a 5 bit register specifier.
- That value is what is going to be set to the PC.
- Use a new value for the multiplier.
- Does the immediate need to be 0, and this is a good point!
- We don't have a way of directing R[RS] into data memory.
- Only works if we define the I-type instruction as 0.
- How do you get the least a 5 bits in this diagram?
- What this is is actually 32 wires. Take 5 wires and leave it off to a different place.
- 5 bits does NOT need to be sign extended!
- We want to redirect the RS into the ALU. Take the output from memory and be able to specify it. Have it MUX in front of the read register. This is the output you want to pass to the PC.
- Controlled by an additional control signal.
- Green is just a control signal.
- Controls these two new multiplexers we have added in.
- It is tangent to the Control.
- Don't care if you are using the dumb instruction, both work but one is a little simpler than the other.
- If something is a don't care but it is actually a 0, you won't lose points.
- If my reg write is 0, I know I am not writing to registers.
- This means I am not writing to anything.
- The register write looks like an enable button of this block.
- If you zero it out, you disable the write functionality.
- Memory should be 1 because I am reading from memory.
- MemToReg is don't care.

Sample Midterm (Datapath Question)

- R types have the same op code, so they cannot come out of the same module.
- If we look at the Control unit, it is not talented enough to generate this particular instruction
- ALU Control takes in ALUop -> you just need the output
- Don't need K-maps, thank god!
- Use ALU to check if R[rs] < R[rt]
- Unlike bneq, we are using unusual control flow. Change PC to PC + 4 + R[rt]
- Set PC to PC + rt
- ALU controller will invert the rt signal
- We want it to subtract, and we need to extract one of the bits for something.
- Now that we know how to use this signal, we can determine if it is less than 0.
- Set less than and set the significant bit. What would this value be?

- It would be true if rs < rt
- ALU result is 1 if R[RS] < R[RT]
- We use this signal to control some multiplexers.
- Tell us if we do one thing or if we do the other.
- Generate a 6 bit output function that is NOT the same as any of the others

W 6 W Lec 2-10-16

- It is only one stage and it isn't broken apart for SW -> LW
- It isn't a hazard!
- Loads and stores have a problem for out-of-order reads in certain processors
- When one instruction can occur in a given pipe stage and we keep things in the order of execution, this is sufficient.

Tradeoffs of SW vs HW

- SW:
- Advantages
- ET = IC * CPI * CT
- Less complex HW
- More flexible
- HW:
- Advantages
- Microarchitecture of machine
- Generally faster
- Static vs Dynamic
- Try to optimize for that in terms of software
- If you wanted to optimize for a particular color pattern in the code, what would the data inputs be?
- Hardware would have a better understanding!
- The hardware might have a better idea of determining the direction of a branch
- Hardware keeps track of the history of taken or not taken behavior
- Has the real time information of what is happening
- The first time it runs the instruction, it has no idea of what the instruction history is
- Every subsequent run will contain the instruction history so we can predict where the BEQ will branch off to.
- Compiler statically will NOT really have much information about the execution
- Hardware is better for dynamic instructions
- Software is better for static instructions (hardware has a limited window for static instructions)

SW -> Reorder

NOPS

HW -> Stall

NOPS

- Most compilers try to avoid using NOPs
- Mostly sees **forwarding** but it also known as **register bypassing**
- Unless we are talking about simple embedded controllers

Stall

- Write address has to be equal to what?
- EX.RegWrite == 1
- EX.WrAddr == ID.RS OR ID.RT
- It is possible to have a match based on a sign-extended immediate
- It may be that we have a jump instruction and the jump instruction had a match for the lower 26 bits
- Make sure it actually writes to the register file
- If I have that alone, we would want to affect the stall by avoid writing to the IF/ID latch.
- Have another control signal that says only latch if it is NOT a stall
- This is functionally correct

Q. Why don't we just not write to the PC and run another subtract?
A. Not stall goes to a write enabled and we are NOT going to allow the OR to bring in another instruction to replace it.

- Put bubbles in the pipeline for the later stages
- We could have a bogus value if we don't store it correctly

Q. How do you know the MUX after == RS?
A. Run a line from a previous stage and don't put it through a latch

Forwarding Conditions

- Forwards from EX me fetch to the ALU
- Did my write address match one of the input operands for A?
- The way they do that is by saving RS and saving RT and these come along in the pipeline latches for the comparison we make in our forwarding logic
- By MIPS convention, register 0 is always 0
- It should never be writing to the register value
- You may think it is redundant but in this case, it is just an additional check for this class
- Make sure destination is NOT zero and we want to write to the register file and ensure it is an input to our EX stage
- EX hazard
- Is NOT complete

Q. When does it get forwarded?
A. It depends on the latch we are talking about, but if we are referring to A, B, and C being read out, before A gets executed, we need to have read in the value for C

- As I read out everything, we need the forwarding logic and we need the MUX input for ALU

Pipeline deeper so you can create more hazards

- You get the cycle time to drop more but it has a greater impact on hazards

Data path with Hazard Detection

- Forwarding unit does that by figuring out when it has to do forwarding
- We are forwarding from this location to whichever location matches the load instruction
- Happens in general and happens just based on register patterns
- Unlike the stall logic we had before, we only stall when we have a load and then another operation after
- Hazard detection unit: This path is RT where we bring it to the Hazard detection unit
- This will be RT anyway and this is what we care about.
- Check if RT matches either RS or RT
- Check destination to the source in the code
- If one of the sources maps to the destination or we have a MemRead, we can force in ID/EX
- Penalty for having a load followed by an instruction gets worse as well.
- This is considerably more complex than what we have talked about it more.

W 6 Dis 2-12-16

0. Pipeline Datapath
0. Data Hazards
   - Stall & forwarding

Understand how the two data path diagrams work

- RegDst is 1 whenever you have add because you are writing to a value
- lw also writes to a value so that is when your RegDst == 1
- For R-types, RegDst == 1 when we have the add instruction
- Apparently, RegDst is 1 at the beginning of each cycle
- In ID stage, it reads the register and checks the values stored at each latch
- Half the two bigs to store it as this two latch en
- Different stages need different controls
- Value in latch should be 1 in this case

Q. Why did we calculate all the bits in the latch?
A. This is for understanding our the latch works. # of bits does NOT matter in this case.

value of RegWrite from $t to 9th
0000 1

- Stall: You don't do anything for a single cycle
- Put off the execution of the instruction

- Implicitly requires the forwarding after the stall

Mem hazard

- Forwarding is done in the memory stage
- Forwarding is done to the EX stage and then you have the result in the MUX from Data memory
- We can do the forwarding back to the ALU

W 6 Dis Uen-Tao 2-12-16

- The more instructions you have, the more work you would have to do in a cycle
- If we did something more realistic, then the cycle time becomes very unreasonable
- We would still have to incur the cost of doing a multiply
- Not a very extensible model for creating a data path.
- Pipeline Principles
- For each clock cycle, we have completed one instruction
- For a single cycle, we are using a bunch of different components one at a time, but you can only use each resource once.
- When you are reading from registers, you aren't using data path etc.
- The work can be split up in a very organic fashion
- We create a pipelined data path and split things up effectively into stages.
- Stage 1: Instruction Fetch
-
- PC (unconditionally): feeds instruction memory with the address.
- PC + 4
- PC reads from MUX from the next PC.
- Signal for multiplexer comes from a much later stage and we know it can become a source of issues.
- The default behavior is that every cycle will be incremented by 4, and that is our next PC
- Stage 2: Instruction Decode
- Take the instruction and they are based on the stages that they straddle
- This register is in between instruction fetch stage and instruction decode stage.
- Stage 3: Execution
- Compute sign extended and shifted left immediate
- In the first stage, PC + 4 was computed and in that stage, it is also propagated forward in order to compute the branch target.
- Zero signal is propagated forward and it isn't until we reach the next stage to figure out with certainty whether or not we are branching.
- It isn't until this instruction reaches the Mem stage that we are actually able to go to the next branching target.
- Stage 4:
- Stage 5: Write back

- The only weird thing is that Write register needs to be propagated from all of our previous instructions
- This would be decoded and the write register has been known
- We have to propagate the register until the write back stage.
- Register memory is continually reading or writing.
- Does the work of two different instructions:
- Write back
- Instruction decode

Pipelined Datapath

- Main issue is that it won't work
- Idealized case where the pipeline can achieve an approximate CPI of 1, but it's CT is much reduced
- This is the result we should be able to get.
- Hazards that prevent this from happening

Data Dependencies

- True Dependency (Read-After-Write or RAW)
- This means that an earlier instruction has to complete or reach some level of completion.
- Anti-Dependency (Write-After-Read or WAR)
- This could be a problem before if the write occurs before the read because there will be an incorrect value that is read

Data Hazard: Dependencies

- We can see why in some instances these dependencies will become an issue
- Reading from $t0 and writing back to $t0
- Only dependent on which registers we use
- Load will use ALU to calculate the address and read in the data from that particular address
- This is where the add reaches the Reg decode stage.
- It isn't until load reaches the write back stage where it can write back to the register file.
- This ordering of instruction will NOT work.
- **We can write to a register and read from that newly updated register in the same cycle**
- We don't have to wait until after the clock cycle is done to read the new value.
- We can resolve all our issues after reading too early.
- Ensure that it doesn't read as early as it does.

Data Hazard: Software Solutions

- If instructions don't work in this pipeline, we can have a software solution.
- We can only control the instructions that go into the pipeline
- We can insert instructions in between these two instructions

- Use NOPS (if it knows how the microarchitecture works)
- We need two NOPS in Uen-Tao's example
- NOPS do NOT do anything, they act as filler space
- This can be a conflict for reading and writing to registers
- Do we have this same issue for reading and writing to memory?
- We would store into memory at CC4
- In a load, we would read from memory at CC5
- The previous instructions will always be done with its memory options before we reach our current stage.
- Doing loads and stores at the same page, and because we stagger the next stage, they can only serve as one instruction at a time.
- Tradeoffs of using nops?
- Almost any data hazards can be avoided by using nops.
- If you did that, the downside would be that you have more instruction count because nops count as instructions.
- Overall CPI would NOT change, but CPI is an average of all cycles.
- For single cycle, it will stay the same even with more cycles to do stuff.
- We are still increasing the instruction count by a significant amount.
- Less portable
- You have to know the # of nops to insert and it probably won't have a 20 stage pipeline.
- In order to have anything reasonable, your compiler needs to know how many stages your pipeline has.
- Previously, we wanted the compiler to just fit the ISA.
- Arbitrates instructions run at that particular time.
- NOT going to work for another architecture.
- **Reordering**: Without nops, this would NOT work as is because we have a dependencies ($t0)
- Our instruction count ultimately would be 5
- We need 2 instructions in between these because of how these things were set up.
- If we have stores and loads, we don't know if we can reorder them unless we know the amount of data within each register.
- We have a case where we can optimize it further if we have more information.
- Between almost every instruction, we need a pair of nops
- This nop solution is NOT really effective.

gcc: Does not necessarily know the microarchitecture, but rather focuses on the ISA

- Emphasizes taking the code as is and trying to make it faster
- It places a very strong burden on the microarchitecture and limits portability and other microarchitectures
- Generally, hardware will take care of handling the reordering
- It doesn't know how the microarchitecture is organized
- Because it is making decisions during runtime, it can get dynamic runtime information from the registers

- Generally reserved for the microarchitecture itself.
- We want to focus on hardware ways of doing this: issues of hardware/ software methods
- If we wanted to make it look like this, we have to insert nops between any two independent instructions
- In order to have an equivalent solution, the hardware can perform stalls
- Purely using stalls and purely using nops will be exactly the same
- Both try to line up the dependent stages
- The add will enter the pipeline and the add will just stay in the Reg file until the value it needs until it is written
- Given a series of instructions, and a specification of how the pipeline operates, draw a diagram for which instruction is at what stage at what time
- Multiple ways of implementing the stalls
- It does enter the pipeline at CC3, it just doesn't do anything!
- We won't save anything to the register and in CC4, the load will enter the memory stage.
- Because we still need to stall, we need to continue backing up the pipeline
- The load is going to enter the write back stage and this time, we have backed it up enough where the result is correct.
- We will have this add instruction enter the decode instruction three times
- Keep having it reenter this same stage.
- This is known as **bubbling**
- It enters and we don't want it to pass forward, how can we ensure this is true?
- The values are going to be read here but they are going to be wrong values.
- We don't latch them. However, we could possibly duplicate the load instruction if we only don't latch them.
- We need to do more!
- The add will effectively enter this next stage but the control value will be 0 for the next stage
- For the subtract, we would have to repeat the instruction fetch stage.
- The add is being passed forward, so just make sure NOT to save to the register
- This allows the add to repeat the instruction decode stage
- We will need a new control signal that is just PC
- The add will repeat this stage over and over until we get the correct value.
- If you perform a sage and don't do anything, just draw a bubble around it.
- This can be accurate for what happens in CC3.
- Subtract should NOT repeat the instruction fetch stage.
- It is only when the add is in the Instruction Decode stage that we can determine what is going on

Data Hazard: Hardware Solution

- The add would be in this stage so it wouldn't be one of the depended upon instructions
- The add is the dependent instruction and when it is in the ID stage, we can determine if we need to stall or not.
- When comparing info between add and load, compare information with ID stage and EX stage
- ID stage has information regarding dependent instructions
- Instructions that we depend on either belong to the EX stage or the MEM stage
- Under what conditions do we stall?
- Figure out what register I am writing to and what signal am I interested in.
- It is going to belong to the WriteToReg signal
- The same thing happens in the memory stage and we want it to belong to the Mem stage
- We only need to worry about this if this is an instruction that writes to a register
- All of these signals need to come together and form logic that looks like the following

Stall (PCWrite = 0, IF/ID.Write = 0, ID/EX.Control = 0)
if:

- This comparison is the condition A, there will be a dependency and that instruction in the execution stage will write.
- If subtract dependent on the load, then we are fine.

Pipelined Datapath

- Those bits are going to be as part of the Read Registers
- If we have a jump, it is possible for it to match the write register field.
- For instructions that we have so far, it is enough (branch, R-type, lw, and sw)
- However, this is NOT sufficient for more complicated systems that have things like J-Type instruction

Forwarding: Uses a register to save a value and pass it in at a later stage

- We wouldn't need to stall because that would be immensely inefficent
- What if I wanted to forward to the instruction decode stage?
- Focus on forwarding to the execution stage.

W 7 W Lec 2-17-16

- We just need to see that the instructions we pull in don't affect the architectural state.
- Doesn't affect memory in terms of storing or loading
- By trying the NOT taken path, we will either be right and no penalty, or we will be wrong and we have to fix the path
- Just make sure recovery time is NOT significant
- With stalling, we **ALWAYS** see a penalty, which is bad!
- In the next cycle, we want to be able to fetch the SLT from memory

- This is known as a pipeline flush and this latch will be stored at the end of the cycle
- This is how they would progress to the next stage if we were to continue executing down this path.
- We are killing off each instruction and turning it into a no-op

STAR label in notes

- This will be an array of two values where we increment and check the initial component of those two values
- Place it relative to T2 and this will be consumed by the ADD and build up a value that does NOT match a certain key

Example:

- In the next stage after memory (CC 12), we can fetch the correct value of lw
- Once the memory detects to take this stage, we should be able to send it a signal to give it information
- Additional MUXing required with potential zeros padded in.
- Avoid a complete stall on every single brach
- In the MEM stage, you will write the correct PC anyways after flushing
- jr has different jump targets so it doesn't work too well
- BTB is NOT going to be on the test but it is used to determine if jump is taken
- Procedure call when you return to different locations, so you want to be able to jump based on the context in which is was invoked
- Something that isn't known statically
- Branches should be indirect because they rely on a level of indirection that is much more difficult to predict.

Branch is very dynamic: Has bias (can be frequently taken or NOT taken)

- Instead of trying to guess it statically, we have **dynamic branch prediction**
- Can be purely hardware or software/hardware
- We can make predictions more intelligently based on this behavior

1-Bit Branch Predictor

- Shift PC by 2 to get rid of lower two 0's and module by 4
- This gives a particular entry of 1 or 0 in the hash table
- 0: Not Taken (NT)
- 1: Taken (T)
- We index in instruction fetch to make a prediction or we index in memory if we know the direction it is going to take.
- Table is being populated by previous branches

Chart on the right helps us understand, and it is NOT kept in hardware

- We are probing the table in IF and updating it in M (trying to see if the actual result stored in the hash table is Correct [C] or Missing [M])

Pattern history table

- Small hardware table typically set in the PC
- This is totally invisible to the software header

The % of times you are right is heavily biased depending on your implementation

- A dynamic branch prediction that is biased towards Taken will have increased odds if the majority of the paths are Taken

2-bit Dynamic Branch Predictor

- There some hysteresis in our prediction
- You will be stuck in unstable states and you will keep making wrong predictions
- You need two consecutive not taken in order to choose the “Not taken” route
- We need a confirmation to be able to follow this way,

Test Question

- Guessing C or M given bubbles and instructions.

Predictor

- It is either taken or NOT taken
- NOT taken because 0 represents NOT taken in my FSM
- What would this entry be updated to be?
- 01 because we aren’t confident of our prediction anymore.
- 10 indicates weakly predicted taken

Get a branch predictor designed

- Get information based on history and it would be some FSM to tell you how transitions occur and we can track data in this table as a result of the correct or incorrect predictions.
- Initial conditions need to be added in.

Is Stalling and Bubbling interchangeable?

- When we want to have a stall, keep instructions in pipeline from making forward progress
- Prevents instructions from ID and before from moving forward
- The mechanism by which you do that is by NOT latching IF/ID and PC
- Prevents forward progress
- You need to put something into the ID/EX latch
- **Stalls create bubbles**
- Adds to the CPI, NOT The IC!
- No-op is fetched from IF and is software provided
- No-op is one way of achieving a stall and it is done through software.

- Flush gets rid of instructions in the pipeline because we don't want it to execute.
- Turn instructions into bubbles
- Flushing and stalling have similar rend results, leading to bubbles in your pipeline

Since flushing sets control to all 0's to determine no-ops, are those bubbles no-ops getting passed through while the instruction passes through the IF or ID stage?

- Yes, they are treated like instructions (the way no-ops are treated).
- They are two sides of the same coin, depending on software or hardware.

When an instruction has made it past execution, can it no longer be flushed?

- In a real pipeline, there is another stage after write back and it can still be flushed.

Branch Delay Slots

- Agreement between hardware and software
- Compiler can inject in three independent instructions
- Don't have an impact regardless of the branch's direction
- Take from after the branch if necessary
- Hammock in the control flow

It is NOT easy to find independent instructions to stick into no-ops

Q. Is filling in slots only in software?
A. We don't need to worry about control hazards anymore.

Q. Why does it need hardware at all?
A. Delay setting of the beq to a later point in time. Usually, it supports branches because we assume there are independent instructions afterwards.

- In a regular ISA, I would never have to specify how many delays exist. In this case, the depth of the pipeline factors into the ISA
- My branch delay slots have a certain length in them.
- In software, we have to find enough independent instructions to fill this or my IC will increase.
- CPI will be kept closer to 1

Delay slot makes you have 100% accuracy

- Never wrong anymore if we set up our delay slots to work all the time

Predict Not Taken
Predict Taken?
Dynamic Branch Prediction

- All three of these are done in hardware so we cannot confirm that it will always be correct

Branch Delay Slots -> Verify that it is always correct because it is done in software

Shortening pipeline penalty and reduce the # of cycles lost to the flush

- It has a great impact on the processor by reducing the wasted # of cycles and wasted pipe depth.

If we can do it even earlier and compare the two register files, we can accomplish this faster

- Tell you if there is full forwarding or not
- If not, you would have to stall for data hazards
- If it resolved in ID, we would require a stall
- Given all of this, you can analyze particular patterns of instructions

Forwarding logic doesn't always work

- Feed it to the value right after it, so we need to inject another stall
- Normally, if you have instruction A and A is not a load, write to EX/MEM latch and forward to it.
- BEQ cannot wait until execution
- We cannot let BEQ read in the same cycle that A is executed.
- Read out the correct value from the register file
- We would have been able to forward and this second stall would need a longer penalty for incorrect branch operations

Strategy:

- Figure out which instruction to do first, and adjust the path that is going to be correct
- Find the time to redirect the code from instruction fetch
- If you prefer to put the instruction in that will actually flush, put in other types of hazard stalls]

W 7 Dis 2-19-16

- What is the prediction made in that pipeline whether a branch is taken or not?
- Once you are in an earlier stage, you waste less cycles
- You would need additional components in the pipeline like a comparator to do a comparison. This is the downside of resolving earlier. The hardware is more complex, but you do save your CPI quite a bit!
- Key tradeoff: hardware complexity vs CPI
- Downside of resolving earlier is that it won't necessarily improve performance!
- If you add a comparator, it can possibly increase cycle time

Example

- Assume we have full forwarding and we predict NOT taken

W 7 Dis (Uen-Tao) 2-19-16

- Forwarding is done at either end of the EX stage or end of the MEM stage

- Branch enters the instruction decode stage or the instruction fetch stage
- Instruction gets a CPI approaching 1, but it executes over the course of several stages
- In the Instruction Fetch stage, it gets it from memory, and in the Instruction Decode, it gets it from the EX stage
- Zero signal is used in the MEM stage to be combined with the AND and this is where the branch is resolved
- It isn't until the MEM stage until we know when we need to branch

Control Hazards

- Writing the PC at the end of the MEM stage
- We are still executing all the unknown instructions
- Is this true for all pipelines?
- Branch target is actually entering in the pipeline
- What is taking so long far this next instruction?
- At the 4th stage, it isn't until practically the very end that we know when we need to branch
- Specific design change, not a natural property of the world
- Set it up such that the branch AND is in the MEM stage

- What enters the Instruction Fetch stage?
- We don't know at this point what enters the IF stage
- By default, everything is going to assume NOT taken since we don't know anything about the branch
- Next branch will be PC + 4
- Over the course of these three instructions, some instructions are going to fill in until the 4th clock cycle when the branch is resolved in the MEM stage
- For this particular pipeline, it isn't until the branches in the pipeline stage that the actual branch target enters the pipeline
- Instructions 1, 2, 3, 4, ….
- What is going to be in the pipeline?
- Going to be in the IF/ID, EX stage, etc.
- They are NOT going to be in the pipeline if the branch wasn't taken
- We have a case where every single time we hit the branch, this will happen.
- Occasionally this is right; occasionally this is wrong

Control Hazards:

- beq can enter the write back stage
- Branch target is instruction 9, and instruction 3 is going to advance
- If an instruction is in the pipeline, we need to flush them
- We cannot just let them pass through because they can write stuff you don't want
- Have they done all these things to change the state of the program?
- They haven't done any of these yet, so we can turn these into no-ops

- These will pass through the pipeline as normal but they won't do anything.
- We want to make it such that as it passes forward, it becomes a no-op
- It has to pass through the ID/EX register
- The control signals also pass through the ID/EX signals
- Unconditionally, this instruction cannot do anything
- Just using the ALU doesn't change the state of execution

Control Hazards: Resolving Incorrect Guesses

- Between control signals, there is an option of choosing between the normal control signal and 0
- In the case where we want to flush or turn it into a no-op, trigger the instruction fetch signal and choose 0
- It will pass through and become nothing.
- Before we can flush it, instruction 1 has gone through the ALU with potentially the wrong value
- We haven't saved it but we want to nullify that effect.
- Key focus is the IF Flush
- This latch is special; no control signals going into the IF/ID signal
- The IF Flush that enters this register does NOT control a MUX
- It will tell this particular register to turn the instruction going into it into a no-op
- inst3 becomes a bubble
- We have instructions but we predicted wrong.
- This is okay because we just forget about it.

Control Hazards:

- We need this to be fast!
- It isn't fast enough.
- It costs us three instructions and we didn't need to do them.
- In the pipelined case, at the end of each cycle, a new instruction is complete.
- We finished up the instruction that was bubbled so we wasted three cycles before we actually get a result that we actually executed.
- 3 cycle penalty: Considered slow apparently
- Even though it is technically correct, it is **inadequate**
- Hardware is simple enough where we can guess NOT taken
- Guess and sometimes it is right, sometimes it is wrong
- When we guess right, we don't pay any penalty at all.
- Obviously, when we are wrong, we pay the penalty
- Reduce the # of mispredictions/reducing misprediction frequency
- "Decreasing how often you pay the cost of a branch delay"
- You need to look at how the branch is executed
- Look at frequency of branches in the code
- If I had code that was purely inline with no branching, what would my misprediction penalty going to be.

- It would cost 0 (reducing overall penalty of execution), more of a software optimization.
- Branch Penalty
- “What it costs when you’re wrong about a prediction"
- Influenced by
- How the branch is executed
- Which stage that the branch is actually taken (AKA “Pipeline branch resolution case”)
- In this particular pipeline, the branch is resolved in the MEM stage

Control Hazards: Reducing Penalty

- If the branch resolves in the MEM stage, if we know we want to branch here, we have fewer instructions lost.
- As you mentioned, it’s going to be two cycles because we have to flush instructions in the IF/ID stage
- Is it possible to resolve branches in the earlier stage?
- Yes! The important part is the Zero signal
- In order to do that, we need to pass along PC + 4 + SES(I)
- We need to figure out if we need to use the ALU
- Use the ALU to take two values and subtract
- Use the zero signal which we could use to branch
- If we had a special comparator unit in the ID stage, we can figure out if they are equal
- We can have a redundant operation and change the critical path
- It is possible
- The branch target enters the pipeline in the EX stage
- If the branch relies on something from earlier, you have to worry about this one?
- We need to forward from here (need to go back in time)
- We have paid a cost, and this cost reintroduced data hazards)
- Everything is a tradeoffs
- He is NOT going to ask you to implement forwarding; just know when it is possible to resolve something purely forwarding
- We can only start forwarding it at a point in CC4 or CC5
- We need to resolve the branch in the ID stage, so we need it in CC4
- Discussion of trying to reduce the branch penalty to 1 and we are hammering this point home.
- The path of reading something from the register file includes the entire work of the ALU stage
- Incur cost of the clock time
- Is NOT possible without stalling.

- Pipeline Diagram Questions on the final
- Memorize that branches are constructed in a simple way
- A simplified version as to what this comparator looks like when we draw it out.

- Forwarding is represented in the textbook by the blue line

Control Hazards: Reducing Penalty

- Resolving branch in the EX stage:
- 2 cycles
- Seems to be a better choice but we need a more complex hardware operation.
- What if the EX stage is the critical path (Primary delay of the clock time)
- It could extend the clock time
- Resolving branch in the MEM stage:
- 3 cycles
- If data was the critical path, it is fine (we have to be wary of how long each stage takes)
- If we need a value of $t0 in the ID stage, what do we need to do?
- Bubble twice!
- If we don't have forwarding, that has to align with the stage where we write back to the register
- This is the same as if we didn't have forwarding.
- Just stall twice!
- What if we had forwarding to the ID stage?
- We would only need one stall
- Save it in the EX/MEM latch and it should align with the ID stage
- Receive the new $t0 value by forwarding
- What if instead, the beq depended on a load?
- If that were true, the value would be retrieved here and it would be needed in the next cycle.
- We just need to stall twice.
- If it doesn't work, just figure out the # of times you need to stall.
- Stalled it enough to align ID stage with WB stage
- If this was the only dependency, it would help to have forwarding.
- Forwarding will help to make this case better.
- Only one stall compared to the load case.

Full forwarding

- Includes only the following:
- Forwarding from EX/MEM latch to EX stage
- Forwarding from MEM/WB latch to EX stage
- Pipeline hazard strategy
- Determine which hazards actually enter the pipeline

Control Hazards: Reducing Penalty

- Say we had a 5-stage pipeline whose branches were resolved in the WB stage.
- Flush the unused instructions from the IF, ID, EX, MEM stage
- At this point, one of the instructions has already finished the MEM stage
- We will have prevented any instructions from writing to register

- However, one of the instructions could have flooded the MEM stage, and we would NOT have undone the store word.

Control Hazards: Prediction

- Hardware by default has predicted NOT taken
- This is just the normal behavior but this may NOT be the best way.
- We could always stall but and this would maintain correctness
- We would never have the chance of predicting correctly.
- Potentially, he could ask if you have a pipeline that always stalls
- Prediction
- Software based static prediction
- Simple operation that is just based on the static code.
- Always taken/not taken
- Something that needs to be done in hardware
- Doesn't take into account dynamic execution
- Branch direction to a higher memory address is likely a loop
- A loop that iterates over 1,000 entry array
- In this case, we have a branch that branches upwards
- If we just predict that almost all the time that the branch is taken, it will be fairly correct.
- You have to consider the software/application
- Based on displacements and op-codes
- Consistent, not going to change based on the execution
- Static code is likely going to be very different from what happens during execution.
- Differs greatly from dynamic instruction count.
- Hardware based dynamic branch prediction
- Branch will tell you what your prediction needs to be.
- Each branch is going to have a different address
- Going to be subject to collision
- Whether you actually branched is going to be a tradeoff on the software side.

Control Hazards: Dynamic Prediction

- 1-bit predictor
- Hash table
- When you look up the table address 0x10, it will have a result
- The entry for this particular branch is 1, and we will predict that the value is taken if the table entry is 1
- 1-bit predictor trade-offs?
- Simple, easy to maintain
- NOT very effective because you cannot really bias it to check if something is mostly taken.
- Whichever option you take will be assumed to be taken next time.
- This isn't a good algorithm!
- Use of a hash table implies there will be collisions

- In the case of a collision, you get aliasing
- This affects execution by making predictions that are NOT actually valid
- branch1 is executed and influences the branch table entry
- branch2 will look at its own entry, and you will predict based on branch2's behavior
- You can affect this based on the previous action, which is NOT enough to make a good prediction
- 2-bit predictor
- Look at the last two moves
- This works the same as the 1-bit predictor, but now you have more states
- 1-bit can have 2 entries, 2-bit can have 4 entries
- Creates a state diagram that allows you to update the entry
- This update is affected by the 4 state transition that we have.
- Example
- Look at his slides
- The assumption is that you will be consistent in your choice of the majority of states
- 2-bit predictors have many versions but they all have 4 states!

Control Hazards: Example 1

- One strategy for doing pipeline diagrams:
- Determine/list the instructions that actually enter the pipeline
- Enumerate the instructions that enter the pipeline

```
lw $t0,
lw $t2,
beq $t2,
lw $tt1,
bne $t1
sw $t0, -> flush
insn1 -> flush
insn2 -> flush
lw $t0
lw $t2
beq $t2
lw $tt1
bne $t1
sw $t0
```

W 8 M Lec 2-22-16

- Things emphasized in class will be put on the final!
- Finish pipelining and introduce a performance metric that finds a way to compute CPI heavily on the final

ET = IC * CPI * CT

- Ways to improve data path

- Look at ways to drop CPI and CT even further!
- Super pipelining: allows us to drop the pipe depth even deeper
- Super scalar aka multi-issue: Multiple instructions in a single stage of the pipeline
- Example is “2-wide” superscalar: You can have two instructions per pipeline
- More parallelism which will drop your CPI even further
- They don’t usually use CPI as a term, they use IPC (which is the inverse of CPI)
- Higher IPC is better
- Stick with CPI because the book uses it and look for CPI’s that go lower than 1

IF ID EX MEM WB
- BP = 3 cycles
- Load-use = 1 cycle
- LW -> ADD

If ID EX/M WB
- BP = 2 cycles
- Longer latency of combined section
- Going with shallower hazards exposes less of the pipeline

Load loop vs branchless prediction
- As soon as you start MEM1, you write to the PC, so extending the Memory stage won’t affect branch prediction
- If I extend my execution, it WILL Impact load-use!
- freq * Penalty
- In this case, we have something more “insidious”
- LW has increased in penalty AND frequency

This creates more hazards for load use but doesn’t affect branch prediction

Questions to ask about calculations:
- Branching is based on how often we have a beq and how often it is wrong

Superscalar
- The density of adding ports has a serious impact on cycle time

ILP
- Finding instructions you can run concurrently
- Has to have parallelism you can exploit
- You need to have code where there are independent instructions you can put together

- When they go through two at a time, there is NO forwarding between those two instructions
- You want to go through it and assume them to be independent
- Find an independent instruction

Static

- Expose all the pipelining details as far as hazards
- You don't have to build hardware that builds that kind of structure

Bundles

- Dynamic instruction execution: Processor tries to pull out instructions and have them execute together
- Go through an example of how the compiler can schedule this code
- R-Type branches with one slot, LW/SW in another slot
- VLIW - Very long instruction word because you have multiple instructions in that bundle

Loop Unrolling

- Grab a value from memory, store it according to some fixed amount
- Add instruction because of load use penalty can go down
- Store word will go down
- 4 instruction bundles that we will have on this particular code on this machine
- There is really only one cycle in which you will have occupancy in that path
- 3 out of 4 cycles won't be taking advantage of super scalar
- Take this and create more parallelism
- Across different iterations, there are NO dependencies
- Do loop unrolling and unroll it twice

Need register renaming

- In order to distinguish each
- Store values before entering loop
- At least these loops don't use large registers
- In cases where we don't have enough registers, there is an added cost of spilling
- Creates a memory dependency issue
- Do some compaction and ADDI's can all go away if you change the offsets

In-order vs Out-of-order

- Architecture needs OS support
- Raise the white flag and ask for fetch to be redirected where the handler comes in and makes any instruction potentially branch
- In the slides and watch the videos
- Pick and choose what Reinman wants to cover

- If there is some issue that flagged up, the machine state would still show things as occurring in order

MEM
REGs

- You may have values that occur within the out of order machine, and we want it visible to the software
- It adds a new stage to the end of the processor called **COMMIT**

One approach is to use a reservation station

- For each functional unit, they have a waiting station where instructions come in, in order
- The result bus coming into the ALU gets snooped by the reservation station
- Each one of these stations will lead to a different type
- The danger here is we don't rename things in the compiler

32 LR

- The ones the compiler knows about, and they can be a lot larger

32 LR => 128 PR

- Usually use that in the internal part of the machine
- The way they handle this is via a mapping
- RAT (register aliasing table)
- If it uses $t0, the RAT will look at the physical mapping and give the current use for it
- If there is a write to $t1, the RAT will pick a free physical register and give a mapping
- Given we have hardware that can actually achieve all of this, there is design complexity?
- Finds parallelism that finds instructions and finds where it can execute and package things up into bundles so it can execute and find parallelism that way.
- Portability
- Stuck to a particular machine organization
- Go from 2-wide issue to 4-wide issue
- Different pipe depth and we need another compiler
- Expose a lot of architectural details to the compiler
- Other tradeoffs
- The compiler has more depth and it can see the entire program flow
- Hoist code beyond branch boundaries
- A lot more flexibility in the distance it can see than the hardware does
- Hardware is constrained by the # of physical registers
- Takes a lot of power, timing delays
- Has a better ability to understanding what is happening in the now
- Can make predictions based on that
- More flexibility in terms of real data

- What are the tradeoffs between branch predictions in hardware and branch predictions in software

W 8 M O.H.

- Compiler would have the potential to see the whole part of the code
- It is good for compilers
- Limited in terms of basic block boundaries or cannot go past certain procedural boundaries
- Tradeoff is that it doesn't look at the dynamic execution of the code
- There are feedback directed optimizations and you have to use the same execution that occurs dynamically
- Hardware is in the moment trying to adapt, compiler can see further since it doesn't care about the details
- 5 stage and 6 stage pipeline,

Scratch pad

- COMMIT: put it into an official ledger
- Out of order creates tentative result and COMMIT allows it to affect the actual architectural result
- Stage in which out of order should affect processor state
- Return to in-order semantics
- Rename it a 2nd time and get the logical process back
- Second RAT in terms of recovery
- We didn't talk about what happens with flushing, but we can control the state and flush the state before it enters the commit
- COMMIT usually affects things in which the order it comes in
- Use non-speculative RAT to get speculative-RAT

Loop Unrolling:

- More register utilization because of increase of instructions
- Benefit is that you have more independent instructions you can use
- Loop, add extra branch statements so you can check and create more control hazards
- With the ISA we have, it may inhibit parallelism but it can have predicated execution
- Predicate and turn control hazards back into data hazards
- Extra registers that are 1-bit and can use guard instructions
- Way of changing control dependencies to data dependencies
- Skipping this 1 instruction in branch and convert it to predicate that is executed conditionally
- No control hazards
- Negligible because you have to check predicates and convert a control hazard into the data hazard
- TCPI is a large # of instructions
- Small examples of two iterations to execute, you can use that formula to try to come up with it without having to come up with a pipeline diagram

- Consider the warmup time of the pipeline
- Use that formula instead of a pipeline diagram

Equation is not accurate for a small # of instructions

- Use pipeline diagram at all times

W 8 W Lec 2-24-16

- Capacity grows very nicely but we don't know a way to develop memory that is both fast and high-capacity
- Register files that have the highest demand
- Caches are typically SRAM that are high speed but low capacity
- Can have multiple levels of these caches
- This is all motivation for physical design
- If we didn't have this issue of building large, fast memory, many of the things today in terms of hardware would be moot

USC anecdote

- Ask about a Professor working at a tenure case
- Pulls out material from his car, forces his out of his car with the shotgun
- The guy with the gun puts the gun, and starts the car
- The guy pushes the robber out of the car
- Sounds like a bad deal at USC and he talks to the dean

- Exploit it in the cache and in terms of accesses to the cache
- In the memory stage, we either have accesses or we don't and this accesses the memory hierarchy
- We have L1 caches that could satisfy a request, or we may have to go all the way to disk
- Assume we don't have disk, and we want to hold the full capacity of DRAM
- 2^32 B = 4 GB

Multitasking and bringing in more applications

- Everything is in DRAM and we are starting there
- Exploit the principles of locality

Temporal: accessing the same thing close in time
Spatial: accessing things close together in space

- Analogy to books in the library
- Basic idea is that you have a large space of stuff that you want to access and you want to look at a small window of it at a time and exploit locality as much as possible

We could have an instruction cache

- If both stages try to access memory at the same time, you can get a hazard

- The reality is that there is a single point of contention and this is a non-deterministic latency
- Built into system of latency where we can build anything

Cache has a few parameters of interest

- Total size: caches cannot hold the entirety of memory
- 32 KB
- 128 KB
- 2 MB
- Caches are somewhere in this range typically
- 3 components that define the makeup of a cache
- Block size:
- How much data is drawn out on a particular access and can take advantage of spatial locality
- Access things close together in contiguous memory
- If we have a cache and we want to go to memory to get it, we have to pull in the block size from memory and pay the expensive penalty out of the cycle
- Growing the size means we are growing more data on the cache miss and exploiting locality
- Spatial locality has been exhausted and huge block size will limit how many you have in your cache
- Latency of how long it takes to pull the block into the cache
- Power drawback for the cache size for pulling out a word each time
- Having some block size is good for spatial locality
- Associativity
- Total cache size is 2^10 B
- Not a big cache size, not a large overall fragment
- Cannot put entire contents of memory here
- If you have 2^10 bytes stored in cache and we hold the data, how do we know where the data came from?
- Cannot rely on the address alone!
- You need tags
- Make a hash table and check that one tag
- For a given address, where do we put it into the cache?
- Use associativity via a direct hashed table!
- Address should only go to one location
- Fully Associativity
- Check every possible location for fully associative (in contrast to direct mapped)

Organization

- 64 sets
- 8 way associative cache
- Look at notes

Difference between a set and index

- Sets and indexes are kind of the same
- The index is which one we are going to be accessing
- See which one we should check
- The way we will access this is to access the row based on the address we are trying to access this with.
- Associativity is how many columns we have to look at.
- Fully associative is checking everything
- Direct mapped is like a hash table

Fully Associativity

- For a hash, we have potential issues for thrashing if items keep colliding when they map to the same bucket
- More associativity, generally means we have more accesses for the cache
- More timing, power, and area constraints

8-way set associative is more complex

Q. Are we building the tags from where the hash address is coming into?
A. We have an address coming in. The address is broken into 3 pieces. Based on the physical characteristics of the cache. Two way set associative cache

- Index tells you which row to access
- Offset tells you where to start my request in the data

For instruction memory, we can still use cache blocks from byte-addressable memory and this is something you do with the instruction cache

- In general, we will look at everything as being byte-addressable
- Are all the tags being evaluated simultaneously?
- There are different ways of designing the cache
- Do a comparison and pick the data that would be appropriate
- More power efficient and less performance efficient
- Where did we get each of the bits for the offset
- These are the 4 factors we care about for the cache
- Associativity tells me how many ways we have for my cache, and we can tell our # of indices
- The block size alone will tell you how much I want for the offset
- Tag is the leftover bits

Bring things in a block-level granularity

- Get a faster return on the block
- Figure out which block to remove
- Eviction policies for a cache
- How do we kick something out of a cache?
- It could have garbage values
- Valid bit
- Usually associated with the tag
- This V bit is the valid bit!

- If V is 0, block is empty
- If V is 1, block is from memory
- Once we have a policy to kick things out, we can choose replacement in the cache
- For a set associative cache, we need a replacement policy
- FIFO/LIFO
- LRU
- If you pick something used first in time, that has the least temporal locality and you are exploiting temporal locality to the extent that you can
- Can be difficult to maintain above 4-bits (very unreasonable and cumbersome for fully associative)
- For class purposes, we can generally assume LRU for the replacement policy
- Valid bit
- The valid bit can tell you if it is a value brought in from memory or a garbage, uninitialized value
- We want to make sure that the data is real and NOT just garbage
- We can clear garbage bits easily

Q. DM is one way basically, so we made it sound like we can only have one block in one way
A. DM cache only has 4 entries. The pattern of access states that we have the lower significant bits coming in.

Q. In order to have collisions (same index but different tag) on the same row, would it have to be stripped by the tag + offset?
A. Yes! It can have the same index but different tag

Q. Go from 0000 to 1111?
A. Moving things at a block granularity and there is this notion as memory being a patchwork of blocks. The cache's view of memory can be broken down into 16 B chunks and each can be aligned so that the lower blocks can have addresses 0000. Even if you have a miss, you are lining it up and this sets every calculation up.

- You can have accesses to the same block even if they have different addresses
- Labeling of A is taking the upper portion of this address and saying this is the range of lower bits contained in A
- Notion of address and cache block address
- Starting point of A all the way to cache block of A
- Can be decreasing or increasing the same, and they can be in the same set and there would NOT necessarily be a relationship between elements in the same set
- There could be something that happens or they could all have the same index
- The tag + index will uniquely identify every block

Q. Would there be an instance when two tags are the same and the map to different indices?
A. If we took this address with the same tag, they can be in different sets. If you find a match, you don't want a duplicate to the same set.

Q. How do we determine the starting point?
A. Pointer into memory that is the location of one byte. The location can be a load byte or load word. Starting point that we want to pull things from. Pull in 4 bytes starting from that 4 byte location. If we are modeling the cache, it is NOT as interesting to us. Offset will be ignored for that purpose anyway. We just want to know the bounds of the offset.

Q. Where do we store hits and misses?
A. Say this is a load word, we will stall the pipeline until it is ready. We would then know we have to stall.

- Usually some decoder that determines how MUX'ing happens.
- There is more memory contained since we have metadata

Q. How do we determine the cycle time if the critical path is variable?
A. Determine if the instruction fetch has an I-cache and this will be defined in our cycle time. Let's say for argument's sake we want everything to fit in IF. Our cache can determine our size of the 1st level structure. Might as well go into memory and pay extra penalty on top.

TCPI: BCPI + MCPI

- MCPI is additional penalty caused by a miss in the hierarchy

We have the next block access to a different index but it is a totally different block, should map to 01 and we can put that in there

- Very crappy data structure to use

Associativity: Tradeoff between complexity of the cache and a set of accesses

- Reason for this is choosing the most power efficient
- Took the tradeoff of looking at one place for the data but we cannot make use of our overall cache
- 4 unique blocks and these can fit into a 2-way set associative cache

Q. How do you tell which is the LRU?
A. You would keep track of the MRU with a flag bit and then select the opposite for eviction

- Least recently used should have been the first way, and this should the least recently used second component.
- If we update, the MRU value of the MRU bit

- We try to put an MRU for the entire set

This would be the valid bit and we are assuming everything is valid

- Pull in B and the memory contents of B and the memory contents of 1

Do all the tag comparisons first and then you drive out the data

- A parallel access should NOT go too high with associativity, but if you want a high L1 or L2, the tag comparison will be dwarfed by the power of the data
- Modern processor has low associativity on the next cache
- It depends on what you can sustain in the processor
- Extend idea of using bytes in memory to using blocks
- We may not need to worry about the block level once we have left the cache
- # of sets is usually given but you can figure it out based on the 3 quantities!
- With a block size of 16-bytes, you can tell the # of indices
- A fully-associative cache is generally a small cache i.e. 8 entries or 16 entries

Three types of misses

- If you know the cache behavior of your processor, and how do we remediate the cache misses?
- You have to know how your cache is getting screwed up
- Helps you better understand how to fix the architecture

0. Compulsory (cold start miss): you didn't have a choice i.e. an address you have never seen before. An infinitely large cache would NOT have helped you and if you context switched in, that's it.
0. Conflict: Fully associative cache would NOT have seen a miss.
0. Capacity: Regardless of what associativity, if you don't have enough space, there will be a miss. You have to have enough accesses or a small enough cache where even a fully associative cache could not have handled the problem.

- Changing your cache could NOT have solved this problem.

- Conflict misses can be solved by increasing associativity, but there comes a point where you fill up your fully associative cache and that becomes a capacity miss.

Stores create a problem for caches because of consistency

- Dirty bit: indicates whether a block has been written to or not
- Go through some pattern of accesses and after a block has been kicked out, you access the next value of the memory hierarchy
- Write-through: update immediately, write it back when it gets evicted
- Problem of coherence can be simplified through write-through, the problem with write-through is that you can have high traffic through the bus.
- It can simplify coherence

- If you weren't writing to memory, you can possibly save time if there was no bottleneck
- If I did a store to A and a load to C later on, if C is critical path
- Write-back will use an extra bit but you wait until the last possible moment.
- Avoid using a lot of writes to memory. Eventually, you will only write it back to memory once.
- Try to get the write done with and don't slow down critical loads

Q. When we do a write-through, why do we have to wait to write to memory?
A. It needs to be buffered somewhere, so we can have a separate write buffer and we can shortcut the bus and bring in a value and overwrite it. Update a memory every single time you write to the cache. We don't have to wait and if we store and see changes propagate through, you only need the memory bus to be occupied. To pull the block, it has to write first and that is the problem we are talking about.

- If we have a store of A, do we pull it into the cache or not?
- We have a **write around** vs **write allocate**
- These design decisions can be made at many points in the hierarchy.

Differences between Write-around and Write-allocate
Write-around: Don't allocate in the store miss, go to next level
Write-allocate: Bring the block into the cache if it misses on a store

- sw -> you don't necessarily have to wait anymore
- lw -> stall everything else since you are just waiting on memory

The pipeline says that the cache is part of the pipeline latency and that memory stage
- Memory BUS is occupied and you have to add the latency of the memory BUS
- Contention for the memory bus
- Write allocate and write-around does NOT influence if it goes to the next level
- If it is a miss, you have to write it somewhere, and L3 doesn't change.
- If you do a write allocate, you would need to bring the block in and write to it.

W 8 W Lec O.H.
- Making a cache that has multiple kilobytes and 1-2 cyclers of latency
- 32 KB
- Memory is a full 4 gigabytes with hundreds of cycle of latency
- Let's say 5% of the time, I have a miss, and I go to memory and it keeps my overall average memory cost low in terms of latency
- Capacity to back me up int terms of just having this and not missing.

Save latency by putting it one stage further and it makes more sense to put it in the EX stage

- EX would probably be a critical path and they don't want to put it in there.

If I had a loop and was taking 2 loops that were similar, we call that loop fusion

- Loop unrolling is a compiler trick to find parallelism in an out of order processor
- We don't have the register space but the hardware does this with much more efficiency
- With branches, we sometimes cannot go beyond basic block boundaries

We have a hard time moving a load above a store if it aliases each other.

- Hardware we can recover

Eventually, there is more code you have to interpret but it improves your portability

W 8 Dis 2-26-16

- VLIW
- Done by compiler
- Dual issue MIPS supports VLIW
- Loop Unrolling
- Compiler has to show how many times the loop goes through
- C: inline
- Hint that the function should be inline
- Loop unrolling is a similar hint using __attribute__ (loop_unroll 4)
- Compiler isn't clever enough to do this
- Register renaming
- Gets rid of dependencies
- Dynamic multiple issue
- Just know that it does the same thing as VLIW but in hardware

Caching
Mapping: Direct-mapping, set-associative, full-associative

Writing policy:
Write-hit policy: Write through / Write back
Write-miss policy: Write allocate / Write around

The processor access following memory address 2, 3, 11, 16, 21, 13, 64, 48, 19, 11, 3, 22, 4, 27, 6, 11

Cache
Number of blocks: 16
block size: 1

For a direct-mapping cache, what is the hit-rate?

For a 2-way set-associative cache, what is the hit-rate?

Assume LRU Replacement Policy:

W 9 M Lec 2-29-16

- One way to view the hierarchy is that in the pipeline itself, go from two caches to one caches and finally to memory
- Miss rate means that the % of cache accesses miss and they have to go to the next level of the memory hierarchy

BCPI vs MCPI

- BCPI handles only the 1st level of latency
- MCPI handles additional levels of latency when you have a miss!

SW

- Do NOT stall on miss
- In this case, 0.25 loads have a chance of stalling
- We have a miss 0.3 times
- This penalty is the same for L2 + MEM
- If you have a LW before a branch, it will be worse because you don't have forwarding logic
- If it was resolved in ID without forwarding, then any instruction that feeds a branch would have a stall
- General data hazard stall and a lot more information needs to be provided
- Full forwarding means you have to do it without stalls
- Additional delay after it leaves the memory, you can use forwarding to get a lot of data hazards
- In the 2nd part of the control hazard, what is the .2
- Misprediction 20% of the time
- Obtained by getting the accuracy of dynamic branch prediction and subtracting from 100
- 100% - <dynamic_branch_prediction>
- 1 cycle access say there is a latency involved in accessing my instruction cache
- Included in BCPI
- Memory accesses would take at least 1 cycle
- Will 2 cycle access affect MCPI in anyway?
- No! The 5% does though.
- What would have to change here?

Apple and Intel will run benchmarks and they will see some killer app performance and we want to make sure to perform well on that too.

- How does having a deeper pipeline lower MR?
- Let's say we started with a 32KB, 2-way set associative cache as our original design
- Move it to 32KB, 8-way set associative cache (reduce conflict misses)

- If we did this, the power went up considerably and we decide to make it serial access
- Pipeline this across 2 stages
- This is part of the design situation to accommodate the more associative cache
- 2 cycles are pipelined across the stage
- Even though we changed the ways for the 2nd example, could we reduce it such that the cycle time goes down?

Knowing for the test

- Pipelining for two cycles allows you to **reduce** the miss rate (MR)

Q. Could we have a 4-cycle and split it up into only 2 stages?
A. How would an instruction persist across those 2 cycles. You need to latch contents as it flows through. Another instruction can come in and its data needs to be stored somewhere too. We need latches for each L1 access. Need a 4 stage MEM and we need M1, M2, M3, M4.

- We may have pipelines that do NOT need latches.

Q. What if there was a 2 cycle instruction fetch in the L1$?
A, Look at notes marked (2 cycle access latency)

- If the stage happens to take 2 cycles, you have to use wave pipelining (off-topic for this class)
- One instruction per cycle coming out
- Limited access and saving energy from having a latch
- We aren't doing that for this class fortunately.

Q. Is MCPI not affected by this?
A. It could be, but you cannot tell directly from the board. Keep the same size cache with no difference in miss rate, but we pipelined our cache more aggressively. This doesn't influence MCPI

- The biggest hurdle for this question is figuring out what the design actually is.
- We did this extension to improve the cycle time of our overall processor
- What is the impact on the hazard?
- Look at this hierarchy and understand the latency at different points.

Every time you access memory, you populate the cache with data

Virtual Memory

- Much less concerned with page table design and evictions for pages
- Only concern is to make Virtual Memory fast
- TLB (translation look-aside buffer)
- What is virtual memory?

- Level of abstraction used to store information
- Granularity of data movement is difficult in a cache from virtual memory
- Virtual memory's granularity is a page

Page Table (PT)

- Stored in memory
- Can be paged.
- If we have this type of organization, let's use the same #'s
- Mapping from one to another
- Translates addresses of virtual pages to addresses of physical pages
- Map things at a granularity of pages
- Address at virtual address and discover what the physical address should be.
- That is the point of having two lw's for a single lw

- Organized so the block is treated as contiguous as it is brought to and from memory.
- Pages will always be bigger than blocks because of this design

Q. Since the 2nd lw is determined at execution, is there any way to do out-of-order?
A. The problem is that there is a RAW (read after write) dependency. Only possible solution is pre-fetching (grabbing a page ahead of time). If you have multiple faults, out-of-order execution will occur and you need to work with an individual case of the fault.

- lw miss vs a page fault is different!
- A page fault is much worse
- If the page is still in memory, we can still potentially do the translation.

Q. What is the benefit of this for virtual memory?
A. Allow each process to have the illusion it has the entire memory. It is easier for programmers because we can allocate addresses at linkage time. Take away virtual memory and deal with physical memory at any time. You would have to compile things to consider what is happening before you link things. This would be a horrible mess!

- Better to have a level of indirection so it can run.
- If you run on my machine that has 8 GB vs yours that has 4 GB, you want it to run consistently.

Q. How do you enable sharing?
A. Have them point to the same page!

Q. How do we have protections?
A. Keep addresses distinct in this regard.

Q. Is the Page Table cached as well?
A. Yes, it is.

Q. Is the linker using virtual addresses?
A. Yes, it doesn't know physical addresses until runtime. When application A or B is resonant, it will give it a few depending on the policy. Depending on priority levels and etc., it will vary depending on those policies

- This page table is a page table cache; if it is resonant in memory, we can use a TLB to cache it.
- Look at the drawing in the notes!
- Indexing
- Any address that goes to the same page should map to the same TLB
- You would want to use some part of the virtual page # in order to index
- Offset
- Based on the size of a page
- Offset in this case is 12 bits
- Index will be based on the # of entries; in this case, 2 bits
- Collectively, the tag and index usually make up the page #, while page offset is the offset

Q. What is the 2 bits for the instruction?
A. Based on the organization of the cache. What would happen is that you would check the index and you would look up for the 19 bits. Check for a tag hit or miss. Get out those 19 bits right away and go without a 2nd load

- Like a cache, where you need to tag

In this data set, we need 4 comparisons. Use the tag to compare against the 4 ways for which of the tags are in here and they drive up the data and we will be golden. Use those three bits to choose of the offset of the cache.

- Virtual address is used slightly differently for the data cache to fulfill the lw instruction.

Another possible organization:
VIVT (virtually indexed, virtually tagged): virtual index is used as both the index and the tag

- When we get to the 3rd alternative, we will see why we use this combination of levels
- Used to organize the data cache, and we have to see if it corresponds to application A vs application B
- If you access the cache fully, the TLB will indicate if it is the right location or not.
- Becomes tricky for protection or not.
- NOT the greatest alternative.

VIPT (virtual address where you use tag, index, and offset per the data cache.

- Access the data cache with that index here.

- If I have to pay this penalty to get to the comparator, if I have the address coming in and I take an index, I would drive out the two waves.
- Drive out these tags to the comparators to do a check, which would have some latency (delta)
- As long as latency is NOT greater than this path, we have NO impact to our performance

Use the same policy for TAG in the physical address

- If you are dealing with a data cache, this will be the cutoff for the page.
- If so, everything after this is all 0, and all of these would be part of the same TLB entry.
- When I look at it from the perspective of my data cache, I will strip off the lower 5 bits for my data cache and each data cache line would be the index to a cache

W 9 W Lec 3-2-16

- The division is the # of pages in physical memory
- Doesn't include the physical pages
- TLB has a subset of the page table and you need a tag to keep track of what part of the page table that the TLB refers to
- To find TIO, first get O (offset), then instruction (I), and T (tag) is left over
- Offset is the granularity of the data storage
- In general, we will ignore that amount of offset.
- We never use the offset and we will index it to get the physical page table.
- It has 16 entries (can hold 16 rows), and 4-way SA, so 16/4 = 4 indices
- If you were given extra size in the TLB, he would have to give more information.
- Status bits would give you 3 bytes per page table entry
- L1 is virtually indexed, physically tagged
- Each tag will be compared in parallel and for each index that we select, this tag will be used for comparison to determine a hit
- Best case is TLB hit but at least you have a translation in the TLB
- If the TLB misses, you have to go and look for the translation in the page table
- Not the best option because you have to do another access
- **PTW (Page table walk)**
- Where is the entry corresponding to the page table index I want
- If you have a fault, you have to go to the OS to retrieve the memory from disk, and then grab your data after.
- In this class, we are just dealing with the TLB access to make this fast.
- Fault: Access a page that hasn't come into memory yet
- Not necessarily an error
- If it is in the TLB, it is fine, and it doesn't have to be in the cache.
- The page has NOT been brought into memory and it is still on the disk.
- Could some of the page table entries be on the caches?
- Part of the benefit of the TLB is that it avoids some pollution.

- In the TLB, you can get potentially a page table entry.
- We can hold 5 page table entries in the cache.

L1 cache

- 64 KB => 2^16 bytes

TLB: hardware structure

- Hopefully NOT on the critical path

Page table: caching

- In the L1 cache
- Parse it in the page table and get out the entry that it wants

TLB size calculations, we were assuming the same data at one entry of the page table

- Physical page # in both and we have 6 bits extra
- When you hold the size of the cache, you usually ignore the tag and just quote the size of the payload.
- How many total bits required to implement the TLB?
- This is what people quote on specs you have a 32 KB cache
- 32 KB doesn't include the tag
- We didn't bias it by % of loads and the penalty

AMAT - average memory access time

Modern Processors

- Have multiple cores
- Cores vs CPUs
- It contains a PC with a thread of control and a set of registers, caches, ALU, etc.
- State that we need for one execution core
- Virtual memory space is allocated in the same physical storage
- Benefit of parallel programming is if A and B are threads, they can communicate through the cache since they have the same shared memory hierarchy
- Communicates through message passing
- We have a shared memory processor with multiple cores that share memory
- Sharing has some benefits and downsides
- Benefits
- More resources that you get access to
- Shared pool is larger
- Downsides
- You have to contend for resources

Cache consistency

- The problem of when to update the cache vs what is in memory
- Memory coherence: Problem occurs when one core doesn't update when another core has an updated value

- Talk about a few alternatives as part of making this class more relevant to modern processors

Q. Is consistency only between cache and memory? Does coherence refer to things between two caches?
A. Consistency within the memory hierarchy. Coherence between multiple cores. Just a term difference

2 approaches

- 1. Snoopy Coherence: snoop port listens to whatever is happening on the interconnect.
- Snarfing: pulls data directly from the BUS
- Automatically updates the cache value of A when B is changed
- Because there is a shared access point to this L2, we would 1st have to pass the data from the BUS and pull the data from the L2
- If you wrote code, how would you physically propagate the change over?
- Private memory vs shared memory
- If I write to address 512, I want the change to be seen
- Q. For snarfing, are the cores directly connected?
- A. Let's assume there is a bus, we have access to the bus with read and write ports. We can have intelligent ways of sharing and we just listen in on the bus.
- This component puts write X with address 512
- Goes to L2 cache and make modifications to its cache as well.

- 2. Directory Coherence
- Bottleneck as you go to larger cores using snoopy coherence
- We could distribute out and have multiple channels to memory
- Real aggressive designs that can become more scalable
- Works for small levels of systems and approaches on large scale networks
- In this class, snoopy coherence is NOT a scalable alternative
- There will be a different type of tracking done through directories

MSI Protocol:

- Modified: The block has been changed
- Shared: Two directories share the block
- Invalid: No one has a copy of the block

Other protocols:

- MESI
- MOESI
- ACE

Why do we have a bigger PA than VA?

- Each process only has addressable memory and it is a smaller addressable memory space.
- They would easily use more than the physical memory that I have.

- We would have some of the processes and these would be resonant in the available physical memory.
- If there was a fault, it would be the OS job for 2^18 pages I have in the physical memory
- The other alternative is if I have a 64-bit VA and 36-bit PA
- We could potentially have 2^64 bytes of addressable memory
- You could only fit 2^36 spots of paging
- Why could we have more physical than virtual memory?
- In terms of that, in a server architecture that is co-resonant, we can have a lot more physical memory so they can all be active at once to share the process space.
- We need a lot of them resonant at once.
- Having a lot of physical memory has more space you can allocate.
- Balancing act that is taking place where you have a pool of physical pages and in addition, you have the OS that is balancing that pool among the candidate virtual pages because of all the applications it is running

Chips

- There will be chips in the architecture on the board and you can support the processor to have a maximum capacity.
- It may not be something that is hardwired and it is capable of handling up to a certain size.

MSI

- Maintains state of all the different cache blocks and let's assume the L2 holds all the space
- Tracks whether that block is in modified, shared, or invalid state

Invalid: No one has the block

- Q. Every time B wants to use Y, it goes to the directory, so why doesn't this remove the necessity of using a cache?
- A. It can keep writing to Y as much as it wants to.
- Thrashing comes about for a contested item.
- Hope is that you have localized sharing but NOT the entire memory space
- In this context, you retain the value until I am told otherwise. Once I am in modified state, I would be fine.

MESI

- Only person has the block in the E stage (exclusive)
- These are further optimizations that try to make it better.
- Q. Once A obtains a modified permission, who keeps track of the permissions?
- A. The directories usually do, but A can set the dirty bit for this.

Q. Will the file ever change back from M to S?

A. Yes, it can transition to S after you force it to write it back. Otherwise, if there is an eviction, the eviction would have to go out and it would have to Write Y and this would change to an I since it would no longer be in anyone's cache.

- Tremendous amounts of conditions and especially race conditions.

Q. If 1 of the cores deletes a value, will it change it to I?
A. If it deletes or modifies something, you need the write permissions. There are silent evictions and noisy evictions.

- For noisy evictions, you would let the directory know you kicked something out.
- For silent evictions, you don't tell the directory and this can occasionally cause problems.

Directory is far more complicated than snoopy coherence.

- Fundamental tradeoff is scalability between directory coherence and snoopy coherence
- Snoopy coherence can create a bottleneck
- Directory coherence can have a bottleneck for sharing
- If you use the block many times, it won't be impacted too much.
- Hope is that most parallel programs do NOT share many things.
- Bad programming practice to have a lot of sharing in the first place.
- The other type of question to expect is that he could give you the state of a machine and if you have X and Y here, and there are some things that happen on each individual machine
- Give me the state of memory after the set of occurrences.
- What would happen if A did a store on Y?
- If I didn't have snarfing, how would I do this?
- This core did a store of Y
- Assuming the caches do a write-through and not a write-back.
- B would invalidate Y, A would modify Y and the store would be broadcast.
- It is write through so you go through to modify L2 and after that one step, you would have seen changes here.

More likely to give a snoopy coherence question on the final!

- More detailed question on snoopy coherence

Intel: core or cache hooked up to a router

- Replicate the same structure over a mesh topology
- Each of these routers can have a core and a cache
- Memory interfaces to reach out and this is designed to stamp out copies multiple times.

L2 has a 256 B block, so we have 8 bit offset.

- Index is 11 as well from the previous example
- This whole span is the index and whatever is above that is a tag

- They actually map to different indices because the 4 bits are the only difference across the block.
- Different indices and there isn't any conflict between those two blocks.
- This green offset is for the TLB and therefore, it is for these three bits and this would give us two more bits in order to have an index for the tag.
- All zeros and these would be one page and these would be a different page.
- All of these would have the same tags as 0
- We would have three L1 cache blocks and we could have two different pages.
- We have three L1 cache blocks
- We also have two actual pages since here, I ignore that part when I am trying to have particular pages.
- My TLB would only be two entries for this.

We can use the information we have from before to break up the stream of addresses and determine if we have a hit or miss.

- From here to here is my index, and from that line up is my tag.
- TLB would be my index and everything above would be all zeros for my tag.
- For L1 cache, why do we have three cache blocks?
- They all map to the same index and these bits are identical and they differ in their tags for their bit positions.
- Create three different blocks

For types of misses, they might ask what kind of miss (compulsory, etc.)

- Other types of questions deals with MCPI and TCPI
- These are the more challenging problems in this class, so they are intended to be a little tricky and we can get them out of the way so we are ready for the test.
- Understand what the architecture is and enumerate everything out.

W 9 W O.H.

- Does it have to encompass all of that memory?
- As a programmer, you have some bit width and when this maps to a machine, you could always get more disk than you could have possibly had in your code.
- No matter what, you could have picked some size and you could always add more disk space.
- There is really no reason why the decision you made has to map up with what I have physically on that piece of hardware.
- There is going to be a constraint based on the OS as well.
- Whatever the linker chooses on its data path
- If you look at x86-64, it can use up to 64 but it is constrained by the OS
- Disk would certainly be different, not impacted by the chip set and the RAM has some limitations.

- Disk you can keep adding more and more space
- Some limit to the virtually addressable space
- How much physical memory do you put in your machine vs how much you put in your disk.
- Put more virtual than more physical.
- I have 64-bit address space and they have to collectively fill my virtual address space.
- I can have multiple applications that consume a lot of that capacity.
- Q. Why can virtual memory be bigger? Why give it more than what is available?
- A. When a program asks for a certain amount of memory, it may NOT be using it all at once. Working set can be much smaller.
- The same physical memory can be reused for different virtual pages.
- I want to read a certain # of books, you cannot read more than one book at once.
- I can keep pulling in books but I can only read one at a time.
- We can access the overall volume and access one page at a time, and physical memory will represent that you won't access the page at the same time.
- The disk can start to get hammered and the disk just starts thrashing because of paging.
- Q. Does each page map to a unique place in physical memory?
- A. We are talking about RAM when we talk about actual location. When we refer to bit address, we are talking about RAM.
- Disk is handled by the OS since this refers to an I/O device
- Q. Could I possibly have more virtually allocated space than physical space available overall.
- A. They don't allocate spaces until it is needed. In some cases, you avoid allocating a page until it is touched. The stack or heap may not grow that big.

W 9 Dis 3-4-16

Problem 1

Cache Access: 9 nano seconds
Hit rate 94%

Main Memory Access Time: 110 nano seconds

TLB Access Time: 5 nano seconds
Hit Rate: 98%

Disk Access Time: 6 milliseconds

Assume 0.1% page fault rate
Assume all the page table entries are in memory

What's the AMAT (Average memory access time) of the system?

Notes:

- System wants to read something from memory, regardless of if it is data or not.
- Q. Why do we bring it into memory if it is a TLB miss?
- A. If the TLB miss, what will the system actually do?

If there is a TLB miss, it will go to the page table and in theory, this step can cause a page fault and we assume all the page tables are in memory.

- Assume it is NOT in a physical address
- Come to the cache, use the fetched memory
- All page table entries are in memory, so why do we go to disk?
- After we know the physical address, do we know if it is in disk or is it in memory?
- In this step, we finish the address translation
- We have to indicate if a virtual address is valid or not.
- Page fault is figured out after the address translation

Q. Is there a page fault that can happen twice after the TLB and if we draw memory and it goes into disk?
A. We don't have the branch in memory. In this point, we realize we have a Page Fault

- Page table entry with a physical address and here you have a valid bit.

We might need to go from Disk -> Memory -> Cache

Q. How can you say the table fits into the memory if the block size is that small.
A. Virtual: 33 bits
Physical: 32 bits
Page size: 4 kB
12 bits
21 20

Page fault -> virtual address maps to place on disk, so this occurs when it is not actually loaded into main memory

Q. Why do we include the write back time? We should't care because I already fetched ti and we don't need to get it from the cache?

Problem 2

Assume write-invalidate snoopy cache
write-through/write-around cache
Two processes: P1 and P2

M_1, M_2 -> L

Events

- P_1 writes 10 to M_1
- P_1 reads M_1

- P_2 reads M_1
- P_2 writes 20 to M_1
- P_2 writes 40 to M_2
- P_1 reads M_1
- P_1 writes 30 to M_1
- P_2 writes 50 to M_1
- P_1 reads M_1
- P_2 reads M_2
- P_1 writes 60 to M_2

L <- invalid
L <- 20

Snoopy cache
- Subsequent read, it will retrieve the appropriate value
- After the value of M_1, P_1 will deliver the value to P_2
- If it is a write-back cache, the value of M_1 should be correct.

Write-around cache:
- write hit: Write back/**write through**
- Write back: Just updates the cache
- Write through: Updates both cache and memory
- write miss: **Write around**/write allocate
- Write around: Just write to memory
- Write allocate: Bring the value from memory to cache and then use write-hit policy

Q. How do we know if it is a write-hit vs a write-miss
A. Here, previously M_1 is trying to write to M_2 and that means there would be a miss.

W 10 M Lec 3-7-16
Multiprocessors
- More conceptual nature

Ask questions on Wednesday about the final!

Dynamically Scheduled
- ILP: instruction level parallelism
- Hardware was responsible for going into that and finding parallelism
- Processor is finding parallelism within a single thread of execution
- How energy efficient is this?

- TLP
- Programmer produces multiple explicit threads and we see an improvement in performance
- These are not necessarily disjoint approaches

- Extract out TLP and try to find ILP within any of these threads
- You cannot always find parallelism because of branches, store loads
- Best balanced TLP job will still have issues (?)
- More energy efficient!
- Depends on TLP abstraction
- If it is amortized over multiple instructions, it is very efficient.
- Compared to static execution, the dynamic execution is much more flexible and versatile

Two ways of using multiple thread contexts

- Focus more on parallel computing in general but we will go into the basics

ILP

- Processors with really large windows of ILP with a lot of parallelism
- Power envelope was far more than they can tolerate
- As we started going to multicore revolution, we would have multiple processors on a single chip
- It could have an L2 cache with a pipeline covered in class this year
- Each would have a PC, so what would each of these run?
- Multiple threads on different processor cores working together
- Split things together into multiple pieces and that would be worked on.
- MapReduce is a great example
- Image processing
- Under the guise of multiple cores cooperating, the big question is where do we place the cores with respect to interconnect and memory?
- These would all work together with interconnect and they will reach a common memory
- Could be multiple units spread out and we pay different #'s of latency
- NUMA vs UMA (Non-uniform memory architecture vs Uniform memory architecture)
- With this idea, you have memory at each core and it is physically distributed
- Each node has its own memory and that can communicate over Interconnect later
- This type of approach is more amenable to a shared memory model
- You assume you have some kinds of shared memory that everyone can access
- Point to the same variable in memory vs a more distributed approach
- MPI is a good example
- You are doing message passing and creating buffers in each memory to allow different tasks to send messages to each other
- Data has to be explicitly communicated with each other.
- Tradeoffs
- Interconnect might be a bottleneck

- If I only had one of these memories and if we had a BUS, it might NOT be very scalable.
- Tiled architecture
- See handwritten notes
- This could repeat into more complex topologies
- You would have a way of continuing to scale this and provide bandwidth.
- If you aren't careful, you could have a single bottleneck, but the hope is that you continue to expand.
- Distributed memory can mean more synchronization!
- If they are really independent things, they are more scalable i.e. cluster communicating
- Separate disk -> I/O bandwidth that aggregates without it being dependent on other machines
- Great utilization of the aggregate bandwidth
- Communication costs get higher until you hit the interconnect
- At least DRAM but it can have its own disk in this spectrum of things
- Depends on what is shared and what is distributed
- Single motherboard with single cores and access to their own memory banks.
- CMP in general are chip multiprocessors
- Multiple cores on the chip
- Cores have become somewhat commodity components and we pushed from an era of homogeneous cores to heterogeneous cores
- Why would you want a less powerful processor?
- Save energy and consider the energy aspect
- Single, very powerful core with lots of out of order executions and one that is power hungry with a lot of ILP
- How do you want to make it load-programmer controlled or hardware-controlled?
- PS3 example
- Scratchpad: similar to a cache but completely programmer visible
- Programmers can lay things exactly out on the scratchpad
- The problem is that the programmer is inflexible and HAS to lay things out on the scratchpad.
- Transfer data explicit to the SPE
- This has to be done on separate threads to execute simultaneously
- How do you choose which core to run on?
- Static based approach where you have to program and figure this out.
- Varying amounts of parallelism and that scheduling problem has been looked at dynamically.
- Performance counters: actual counters in the hardware
- Components that can add up accesses to a certain variable
- How many times the cache in your pipeline flush?
- Apple makes this obfuscated.
- Can be used for tuning with a period of high cache misses with a lot of out-of-order complexity.

- Could be run-time based or hardware scheduled.
- We have this notion of chip multiprocessors in general
- Try to push this further to accelerated models
- Not just explicit cores but we can have different times of algorithms on different cores
- More extreme power but less efficiency
- More of these heterogeneous types of approaches

Interconnect
Mem: more and more cores => more and more workers

- You need to have enough on-chip cache to hide latency and bandwidth.
- Having these cache topologies and these can be used to try to reduce latency and provide a better cache topology.

He will only ask really high-level questions

- Tradeoffs between distributed multiprocessor and shared multiprocessor
- View that a system might have in the software perspective
- Multiple contexts for threads that exists

Fine Grain Multithreading (FGMT)

4-way superscalar

- Multiple instructions per cycle
- 4 issues slots you can fill
- Look at time
- If I study one program executing, I might have two instructions in one cycle and three instructions in another
- We might get a miss where the instruction pipeline stalls
- What could fine grain multithreading help with?
- Vertical waste because as soon as you get a long latency stall, you could switch to another thread.
- Idea with SMT is to help with both vertical AND horizontal waste
- Try to figure out true data dependencies and make these more available ILP and you have a lot of different solutions
- All sorts of reasons that you have this waste.

Physically speaking, the overhead for filling everything up means you have multiple copies of the register file

- You need to check register resources for X, Y, and Z

Multiple PC's because you need to track three threads of program execution

- Other instructions might have to be duplicated or partitioned

They had issues called the tornado effect that caused scheduling problems

- Multiple threads that execute execution slots
- Area overhead was somewhat minimal

- What could be the drawback?
- If you have two that are switching and access things out of memory, you will draw things out of memory and pull them back into the cache repeatedly (redundancy is high!)
- With these three threads trying to coexist, they will be fitting in the cache resource.
- If one of these has more vertical waste, we will have even more misses if we don't know if there is enough parallelism to supply what happens there.
- Throughput increases at the cost of single thread performance
- Any given thread will perform worse and some applications can be slowed down if it is running background applications
- Intel had a particular problem called the tornado effect
- You could have two streams of instructions coming in
- If your scheduler was wrong and you had a really deep pipeline, rather than waiting for an instruction to finish, you could guess when the instruction will be done.
- This would starve the other threads and deny them resources (we had denial of service attacks because of this)

Multiple threads compete for each other to get rid of factors like branch prediction, more complex cores, etc.

- Arguments that we need both types in our system.
- SIMD

Q. How did swapping out for another thread take care of hazards?
A. Control hazards -> if you hit a branch, you can try to predict that branch or you can have a signal that does thread swaps and pulls in threads until that branch resolves

- Look at notes example

Frontend and backend

- Instructions are frontend
- A lot of complexity that comes with fetching out instructions and all this overhead costs a lot of power
- Data are backend

Single instruction multiple data

- Single frontend shares multiple data to the backend
- Q. Does the frontend have to be faster than the backend?
- A. Not necessarily!

Nvidia GPU's

- Has anyone programmed in Kuda(?) or something similar?
- In something like Kuda(?), you maximize things when you have non-divergent branches
- Improving when you go in lock step because you embrace the SIMD models
- **We won't be tested on GPU's!**

SISD vs SIMD

- Multiple streams of instructions and hopefully I can improve my throughput.
- Had something with a distributed memory wall and hit a communication gap.
- My communication would become dominant!

SIMD vs MIMD

- Less bandwidth and loading less instructions from memory.
- Not going to have as many instructions in SIMD (helps with bandwidth)
- More parallelism and you want more similarity for types of operations so they can fit in the SIMD model
- If everyone is doing an add and there is NO shifting, then we want to have this sam flow of execution and this is amenable to a SIMD model.
- SIMD is a very amenable model and you have the improvement without the tradeoffs.
- MIMD
- We can extract parallelism when things are totally different
- We have to pay for the power on the frontend
- SIMD
- Extrem is a GPU and a smaller variable would be instructions that manipulate data in this type of way where a single instruction adds multiple pieces together
- MIMD has multiple fetch executions

Q. Are there any models that can do both?
A. Execute multiple SIMD instructions by moving to Intel many cores.

- Each of those would have SMT as well as SIMD extensions

Multiple lanes for precise and imprecise

- Area you have may be outweighed by porting requirements
- Can be used in multiple ways!

Q. What do you mean by bandwidth of instruction?
A. On the frontend, there is an instruction memory pulling in instructions. Amortize with a single frontend and reduce your instruction band since you would be pulling in once but you lose flexibility.

Conjoined cores

- Two cores that are physically close enough
- One core could use a floating point unit at a time, allowing you to reduce size of the core and reduce the amount of silicon used
- Tradeoff is trying to access the resource at the same time can cause conflicts.

Wednesday - Final Exam review

W 10 M O.H.
Exceptions

- Static speculation vs dynamic speculation
- Speculation: you don't know what the actual value is but you take an educated guess.
- Branch with a negative bias and you go backwards in the instruction string, guess that it is taken.
- Dynamic prediction is how many times you see it go backwards vs how many times you see it go forward.
- Based on what you've seen in either analyzing the code or a past run, but NOT the current run.
- How do we pass it into the architecture?
- Use extra bits to flag things to associate it with the architecture.
- The compiler does this and packs it in a format.
- Immediate field and a register field
- The compiler would set the bits based on the context of the instructions
- We talked about how we don't know if loads or stores alias each other.
- They don't know it, but they guess with a check load that gets verified by a real load
- Pre-fetching, instruction guesses that it will go down this path and it is impacted by ISA
- You would have to have extra code with the loop unrolling
- Speculating that the branch would be taken a multiple of 4 times and we have to fix things up if we were wrong.

Q. If I do a loop that goes 20 times, and we pack it into 5 instructions, will it hit 4 times?
A. If there is a multiple, then you don't have to put in the recovery code. If it is an evenly divisible multiple, the IC might go down because you encounter the branch. You can pack all the addi's together.

Q. In the context of the size of a cache, we have tag, index, and offset. Given an address, is it just a pointer into the cache? How is the cache data stored together?
A. There is only a tag that is stored with the block. The size in hardware specs is based in data. There are extra bits NOT accounted for because we have valid bits, tag bits, etc.

- Given size of cache, divide by the number of blocks to find unique entries, you would further to divide that by associativity.

What is 3C?

- Compulsory, cold, etc. (types of misses)

No need to worry about exceptions

Amortization: If you compile something, ever time you have to execute it. If you compile something once, and you run it 1000 times. The cost is amortized. The more amortization you can do, the less that cost because.

- Cost is spread out across multiple uses.
- Sounds like an averaging in terms of Big-O
- Usually in the engineering of a design, it is if it has multiple uses and it defrays a cost.

Good resources to practice?

- The most challenging would be with TCPI
- What if I went from a 5-stage pipeline to a 10-stage pipeline
- What would be the impact on control hazards?
- Enumerating out a frequency and penalty part.
- The hardest part is understanding the question Reinman will be asking.
- The instruction count will go either up or down.
- Register spilling
- This meant that we would have less loads or stores because of spilling.
- If I am increasing the register file size, I am decreasing my spilling.
- Fewer loads and stores improves miss rates and you would have to figure all that out.
- The book doesn't have good practice problems for TCPI
- Static compiler and multi-issue stuff with loop unrolling
- Loop carry dependencies
- When you unroll, and you have an array where each iteration of the loop took the value of the next element.
- Rearrange things to optimize the instruction.
- We only covered one so it will probably be loop-unrolling and figure out what can be unrolled.
- Snoopy coherence question

Dynamic scheduling

- Something called a RAW or WAR hazards
- lw -> r0
- If I have another subtract from r0, and we have a RAW, these will be independent, but you cannot execute the LW and the ADD at the same time.
- If ADD goes before the AND, that is the WAR
- Register renaming lets you get rid of false dependencies
- How do we know if they weren't false dependencies, we have to have a compiler analysis and register 0 will be dead at this point in time and if there was a branch, the compiler couldn't deal with it.

W 10 W Lec 3-9-16

Performance -> ET = IC * CPI * CT
ISA -> Tradeoffs!

ALU: Structure + tradeoffs
Adder
Multiplier
SLT

Single Cycle Datapath

Pipelining		Fundamental idea + theory
Data Hazards		Pipeline flow + analysis
Control Hazards	TCPI	
Super pipelining	Scheduling	
Superscalar	Implementation completion	

Caching	->	Structure + Tradeoffs
		Address Stream Hits/Misses - What kind of Miss?
		MCPI + AMAT

Virtual Memory	Structure
	TLB Modeling

Multiprocs	Cache Coherence
	Tradeoffs + Terminology

Left side: topics
Right side: potential questions

Bolded: On the midterm but not as emphasized on the final. Don't completely forget about the ALU. Three types of adders. Multipliers exist here. Go into SLT which has an interesting feedback.

- Focus on structure and tradeoffs.

Performance: instead of CPI, we will use TCPI
ISA: all about tradeoffs and limited # of bits, what would you get if you have more or less of those bits.

You would potentially have less spilling with less registers and do a more efficient job of register coloring.
Single cycle data path: CPI is always 1. Everything takes a single cycle and this design carries over to pipeline

Won't ask an extending single cycle question, but know parts of this.

After the midterm
Pipelining: Improves throughput. Latency could get worse due to pipeline latches. How long does it take to reach a steady state.

Data hazards, control hazards -> tradeoffs here

- Tell me how long a particular piece of code executes -> pipeline flow diagram, forwarding, stalling, etc.

- Draw the pipeline diagram and show me the flow diagram
- TCPI: does not take into account the amount of time to warm up the pipeline
- Bad for short sequences of instruction but ideal for longer sequences
- We can tell how often we have load use hazards and dependents
- Super pipelining
- Deeper design
- **NOT** expected to add additional stages to super pipelining and draw a new picture
- He could ask though something like this
- Check the notebook!
- Superscalar
- Static vs dynamic
- Static: unrolling and example of this in the homework
- Do a couple of examples of this one
- Compiler is doing the scheduling, so the compiler has to consider stalls and when you compose the schedules, you have to look at the time taken overall.
- Pull in registers to get rid of false dependencies, get things to overlap more.
- Dynamic: Know the basic tradeoffs
- **NOT** going to be asked about reservation stations, etc.
- Implementation completion
- Given part of the design and complete the rest of the design
- Completion should just be connecting wires or something along the lines of the complexity of single cycle datatpath.

Q. Why can't we put the latches before the comparators?
A. That should be okay. There are multiple solutions for each of these things.

Q. What are examples of latches that don't work?
A. If you did a double latch and a single latch, there will be a race condition. When you add latches, there are a few places it could go wrong but just be careful.

Caching

- Elements that compose a cache, associativity, # of sets
- Form the overall cache size
- Given an address stream and you know the size of the cache, give hit/miss behavior
- 3 C's
- Types of Misses

AMAT includes access latency

- Need to give AMAT with respect to what?

MCPI does not include access latency

Virtual memory

- # of bytes addressable in Virtual memory, page size, # of bytes addressable in Physical memory
- Break down the page table and ask how many entries it has.
- For the TLB, if it has 8-way set associativity and 4 sets, how many entries can it hold?
- All the analysis that we did on multi-boards
- TLB and Cache
- Check for TLB hits and misses like we did for catches

Multiprocessors

- Snoopy Coherence
- Will be asked a snoopy coherence question
- **NOT** going to be asked Directory Coherence
- Just understand tradeoffs in space
- We went over the difference between chip multiprocessors and simultaneous multithreading
- Terms that could be asked on a multiple choice question

Q. Snoopy coherence. Without snarfing, do caches have to be write-through?
A. You can imagine a case where you have a cache as follows (look at notebook)

Tradeoffs between directory coherence and snoopy coherence

- Directory coherence: advantage in scalability. Don't have to constrain bandwidth. Add more cores because it has a single place to go rather than the cores having to listen.
- Snoopy coherence: No extra traffic on the network when you want to check the load you have to access. Extra overhead because everyone listens to everything. Sharing is good because you can tell everyone what you are writing

Scalability is better for directory, but snoopy could be better for small structures

Snarfing: Instead of snoop invalidating, when you modify A with A', you put A' in the cache as well.

- Extra overhead in the port because you write both tag and data
- Adds more overhead to the physical structure but can improve access time

If you see a problem worth 30 points vs one where it's a bunch of multiple choice questions worth 1 point, i.e you cannot remember snarfing, don't waste time on 1 point questions.

- Do multiple choice questions last!
- Focus on some of the questions like TCPI

Address Stream Hits/Misses (Will be asked on the final)
TCPI -> show all work and spend more time on these topics

- You need to add a MUX that takes this opcode and takes an equal SLT and pull the info

The TA's said it was comparable in complexity to the sample final

Practice Questions

- Snoop Inv
- If this is the state before, what are the states after these 3 stages execute
- Branch delay slots lets us resolve BNE earlier, but not in this case
- More code puts more pressure on the cache, and this may cause more misses assuming the cache can hold those additional instructions
- It would have contained 5 instructions and we should have done more manipulation
- 9 instructions that our compiler would have to place in memory.
- Static instructions go up!
- Just through code analysis, you still need to store and you cannot work around modifying the allocation
- Change instance of $t1 to $t0. What is the problem with this?
- Not only do you have to know if $t0 is used for other purposes, but you have to know if $t1 is also used outside of the loop and hence is useful.
- If I didn't know we could move a load above the store, we have to account for that.
- Why do we need the 2nd LW?

TLB holds 128 associations so we hope we can exploit locality to optimize our situation

- If the page table has a fault, then we have to go to disk, which is the worst case.
- There can be multiple memory requests just to get our memory out.
- Some of the page table can be on disk.

Virtual addressing is ambiguous, physical addressing is **NOT**

- You need some mechanism to disambiguate things
- Depends on the relative size of the page and the cache.
- The natural question to ask for this is the address stream
- Skip the first 6 bits and from the cache perspective, it doesn't matter.
- Index and tag together that give you the unique address and tells you if it's the same address or not.
- Partitioned it off into a tag.
- Figure out the unique blocks
- Contrived example

TCPI: Understand what the question is asking

- Keep track of things on the question so you understand the architecture

- Virtual address translation
- Virtual address -> physical address
- 32 bits
- 0xffff82c4

- For the cache, we feed a virtual address, and then we get the data of the virtual address
- For TLB, we feed into the virtual address and we get the physical address

Cache:
VA -> Data of the VA

TLB:
VA -> PA

Finally, for page table, we can view it as an array
- Page Table Array

PT[Index] = data

VIVT will probably not be on the test, VIPT will likely be on the test

W 10 Dis (Uen-Tao) 3-11-16
- Everything refers to Kikibytes on the test!
- KiB = 2^10
- MB = 2^20

Final Topics: Pre-midterm Material
- You were only expected to know the following
- Performance: ET = IC * CPI * CT
- ISA: Trade-offs
- Ex Making register files large => less spilling
- ALU: Structure and trade-offs
- Adder (no timing question)
- Multiplier - could be tested
- Be aware of Booth's algorithm, but you won't be asked about the Multiplier in conjunction with like a pipeline
- It will be an isolated topic
- SLT
- Single Cycle Datapath
- No extending the single cycle data path question

Final Topics: Post-midterm Material
- Pipelining Question Types
- Basic Implementation
- Pipeline Flow Diagram
- TCPI

- Scheduling
- Implementation Completion
- Pipelining Topics
- Data Hazards
- Control Hazards
- Super pipelining
- Static multi-issue (VLIW)
- Dynamic multi-issue (superscalar)
- Caching
- Structure
- Trade-offs
- Address stream hits/misses and what kinds of misses?
- MCPI/AMAT
- Virtual Memory
- Multi-processors
- Cache Coherence, Trade-offs + Terminology
- CMP (chip multiprocessors)
- Different cores on different processors
- Heterogeneous cores
- SMT (simultaneous multithreading)
- SIMD (single instruction multiple data)
- MIMD (multiple instruction multiple data)
- Every topic asked on these will be very high-level

FInal Review: Static Scheduling

- HW 7, Q5
- 2-way Superscalar (VLIW statically scheduled)
- 5 stages pipeline
- Specialized issue slot

- Schedule the following without unrolling.
- You may assume that the store never goes to the same address as the first load
- Use static scheduling and we need a gap between these since this is a load dependency)
- We need a gap between the issue slots
- R-type dependency so we don't need a gap between these.
- No true dependency
- addi can really go anywhere because these two use $s0 and it won't collide since they are in the same issue slot
- Read the original value of $s0 and write to it.
- bne needs to be after the addi
- When a branch enters the pipeline, what follows the branch in this case?
- Two ways of handling control hazards
- Branch prediction
- We don't know exactly what will come after the branch.

- What happens when the branch advances to the ID stage, what enters the pipeline?
- It will be the prediction that we are making.
- Predict the branch is going in one direction.
- Delay slots
- Don't let anything enter the pipeline until the branch is resolved.
- If you were doing the branch prediction and flushing, the branch has to be at the end of the loop.
- With branch prediction, effectively, this instruction becomes NOT taken.
- BNE has to be at the very end.

- With delay slots, instructions entering after the branch will be executed.
- NOPS enter the pipeline after the branch, but things need to be executed to completion.
- If we had delay slots, we could schedule it and if we use no prediction, this always enters the pipeline.
- Either don't take it, or we move back to the beginning
- **You don't need to know about branch delay slots**
- Don't need to schedule on the assumption
- **Assume branches are always at the end because they need to be handled by prediction!**
- Q. On the test, we will specify that branches will be handled using prediction
- Ignore the delay slot stuff and treat it as if it was using branch prediction

Final Review: Caching

- Cold/Compulsory: This block has never been accessed, so this miss is always going to happen.
- Capacity: The block has been cached before, but since the cache was full, the block has to be thrown out
- Only going to be a capacity miss, if the cache was fully-associative, you would NOT have the same sort of set issue
- Conflict: The block has been cached before and the cache is NOT full, but due to a conflict in the set, the block has to be thrown out
- Only going to happen if the cache was not fully set-associative.
- What can be done to reduce each type of miss?

Cold/Compulsory

- Each time we access a byte, we pull in one block (cold miss)
- If we move to another array with another block, that is another cold miss
- **Increasing block size**
- **Prefetch blocks**

Capacity:

- Increase the size of the cache.

Conflict:

- Cases where we have too many blocks in our set, and even though the cache is NOT full, we still had to evict some blocks prematurely.
- In the ideal case, direct-mapped caches would be king because that is the lowest amount of associativity
- Even then, you only have one block in that set, so you have to hope you are in the same block.
- Because your set can contain only one block, your cache can be completely empty and you can only hold one block at a time.
- How do we reduce the impact of a conflict miss?
- **Increase our associativity!**
- To reduce conflict misses, we can use a full associative cache, so we never have to evict a block until it is entirely full.
- Reduced block size => less locality

Final Review: Virtual Memory

- The Intel Haswell chip supports for three different page sizes, 4KB, 2 MB, and 1GB
- This suggests larger vs smaller pages.

What are the advantages of larger page sizes?

- Once you pull in the giant thing, you won't have to pull in as many from thread execution
- Copy a page from disk, if you have a larger page, you have pulled in more virtual address space at a time.
- Time advantage because there are fewer page faults
- If you copy more from disk, you don't have to incur from disk multiple times.
- The way virtual memory operates is that it is a fully-associative cache
- Now, physical RAM/memory acts as a fully-associative cache.

Fully set-associative cache so we reduce the impact of conflict misses.

- Operating system maintains the mapping and that would be ridiculous with physical memory.
- OS maintains where each page is and keeps track of everything.

Final Review: Cache Coherence

- Two different shared multiprocessors with different L1 caches, shared L2 cache, and main memory
- Write-Allocate: We have to go to the next level hierarchy and we will try to read B
- Evict A = 10, and instead, it will have B = 20
- Discuss weird alternative policies and how those would work.
- The cache policies are dealt with at 1 level at a time, then it depends on the next level's cache policy.

- We don't have to do anything to B of the other processor because we haven't written anything.
- Evict A and pull in B from A.
- Write-Around: Write miss policy where instead of copying data to the next page, we just go to the next level.
- Follow the miss policy whenever there is a miss.
- The read is straightforward, and there is no alternate policy.
- Sort of like a write-allocate because you are reading C to it.
- Comes in two steps
- If you miss, you perform a read to the next level:
- For snarfing, you need an additional database compared to snooping. Write-invalidate is much simpler because you don't have to update values

Memory

- Memory addresses range from 32, 48, and 64 (?) bits. If we consider the 32 bit case, the main memory was be at least 4 GB
- In the past 10 years, the computer could have been a 32 bit processor, but computers can function normally with less memory than that.
- With 4 GB of memory, it would be fine for every process.

Virtual Memory: Basics

- Virtual memory doesn't use "blocks", but rather "pages".
- Each page is just a contiguous chunk of memory of a set size.
- Uses a virtual address and we need to look into physical memory to see what it corresponds to.
- Page fault -> go to disk and copy that piece into main memory.
- Consider a case where the page size is 4 bytes per page.
- We need to organize it into a span of 4 bytes per page.
- If we access address 0x06 and we want to see which page it belongs to, it spans Page 1
- This address belongs to a specific page, in this case, Page 1
- Page # and page offset is the zeroth page in physical memory.
- Page offset is the offset from the beginning of the page to the actual address we are interested in.
- This access goes to page 1 and it is at an offset of 3.
- It would be an offset of 3 belonging to page 1.
- Our virtual memory is split up and this type of diagram could be replaced by block.

Physical memory consists of a scattered mess of page data from many processes -> this could be a page from some other process

- Each thing is going to be shared within that physical memory space.
- We need a way of understanding that Page 2, Address 9 corresponds to address 0x01 at the present time.
- This means that if we have an instruction such as lw $t0, 0($t0)
- The R[$t0] + 0 is a Virtual Address

- Physical memory is a “cache” for the virtual memory on disk.
- Find the location of a block on the cache, and it can only belong to a particular set.
- Looking at just the address, we can find the block on the cache.
- RAM is too critical to risk a conflict miss, so we don’t want to miss.
- We generally want a “fully set associative” cache in this case.

Virtual Memory: Page Table

- Virtual Address decomposition

[VPN][VPO]

- VPN: Virtual Page Number
- The index into the page table
- What you get from the page table is the physical page number!
- Decomposed in the same way from the physical address
- In both situations, the PPO and VPO are the same since page size is consistent in both virtual and physical address space
- Physical page number is synonymous to tags
- Protection bits are defined in the homework to make it an even # of bytes
- Don’t worry about alignment and other crap like that.
- For each virtual page, we need to map it to a location in physical memory.
- Because there are 2^6 pages, then we have 2^6 entries in our page table.
- Page table is an entity stored in memory, it’s not a hardware component.
- Memory resonant (always there at any point in time)
- Page table entry will indicate either:
- The virtual page is in memory at location [PPN: VPO]
- The virtual page is not in memory (valid bit = 0)
- Page is either allocated on disk
- Page is unallocated

With multiple processes running on the same machine, a process can only see its own memory space.

- How can it possibly see that Process 1 has a variable at a specific address?

A lot more complicated!

- For every memory access, you need to actually make two memory accesses!

Because the page table is simply a data structure in memory, it can be cached like anything else!

- Initially, the CPU would issue a virtual address and give it to the MMU (memory management unit)
- Instead of going straight to memory, we are going to the address in the page table entry.
- Figure out where the address is based on the size of the page table entry.
- Be able to get memory for the cache.

- If there is a hit for this particular block, it will return the page table entry for the MMU
- Reconstruct the physical address and have it for the L1 cache
- Cache is NOT super efficient!
- We need to do initial accesses to main memory to get it into the page table.

Virtual Memory: TLB

- Translation lookaside buffer is the transition state.
- Made it more complicated, but in the ideal case, we have made it faster!
- This TLB issued only for page table entry.
- Does NOT cache data; only page table entries!
- When you get a TLB hit, you are not done. You have to follow the pointer and get the data.
- Accessing the TLB is the same as accessing a cache.
- TLB is indexed like a normal cache.
- This is an example of a TLB with 4 sets (4-way set associative)
- Contain the PPN as well as the valid bit
- Payload is the PPN rather than a cache block.

Virtual address -> if you have a hit, you can get a PTE and convert it to a physical address.

- Get the data back from here.
- If there is a TLB miss, we have to issue the Page Table Entry to cache as before.
- Once we find the page table we are looking for, we pull it into the TLB.
- Find the TLB entry and pull it into the TLB to maintain a cache relationship.

Virtual Memory: Cache Indexing

- Cache only using physical addresses
- Best case scenario
- Accessing the cache can logically be broken down into two steps:
- Find cache line and 4 way set associative to where the data might be in the cache.
- Use the tag and compare it to see if any of the blocks are the ones you are looking for.
- If the blocks are two bytes, the page will consist of two blocks, and it will span several blocks.
- PIPT: Physically indexed physically tagged
- Almost no caches implemented in the real world are like this.
- The TLB adds to the critical path -> we have no choice but to access the physical address
- In the worst case, it would be so bad that we don't have to consider it.
- The cache just depends on whatever address we are indexing on to.
- VIVT: Virtually indexed, virtually tagged

- You would look at the virtual address that the processor originally issued and insert it into the cache.
- We can reach our best case with a hit in the cache.
- We have no choice but to go to the physical address mapping.

Urn-Tao Office Hours
Difference between addi and add?
- add immediate vs normal add

Full forwarding: Assumes forwarding a previous value to the EX stage at a later instruction
EX -> EX (one instruction down)
MEM -> EX (two instructions down with NO bubbles, one instruction down with 1 bubble)

Stage definitions
IF: Grabs the instruction
ID: Decodes the instruction's value
EX: Executes whatever the instruction is (most values are ready by then so we have forwarding here)
MEM: Memory does accesses for main memory for I-types (MEM -> EX two stages down)
WB: First half is writing to the destination register, second half is reading from the register

Transparent latch: We use a transparent latch in the implementation of the register file and when we perform a write, it performs it on the first half o the WB
- Allows the WB to have the functionality where it writes 1st half, reads 2nd half

Final
- Problem 4: Page table entry would both map to the same thing or it would be invalid, disk is the true backing storage
- Page table: we divide by virtual memory space (hypothetical virtual memory) / page size
- Page table maps virtual pages to physical addresses, where the pages you have are actually stored
- We need physical memory as a cache because disk is too slow
- If this is address 1000, we are looking for the bytes in physical memory of address 0
- Look into physical page 0 and that is where the data is actually stored
- Take virtual address and take this to look into a table
- Mapping is the table itself, and if we have 0, we go to index 0 of the table (at a high level)
- We find the physical address
- Mapping in the granularity of pages (NOT clean!)

- We first have to figure out what virtual page we are looking for

Virtual Page #

- Take 2 bits and the rest is the virtual page #
- Caching block, 2 bits; tag + index (based on how many sets there are in the cache)

Q. How does the virtual address know where to go?
A. Access virtual page 0, then we see what is index 0 in our page table. It is the virtual page #, what the mapping of the page table does is converts virtual page #'s to physical page #'s

TLB

- Cache page table entries
- Convert virtual address into a physical address
- The TLB is a hardware component
- Only one TLB for each processor
- Each processor is probably going to run more than one process
- TLB indexed using virtual address
- All processes have distinct virtual address spaces
- Another process might have something else at address X.
- These all map to the same location and we would have to flush the TLB to make sure we don't have any false hits.
- We have one TLB per processor, processor runs multiple processes
- TLB is like a cache (physical hardware component)
- Page table for each process, but only one TLB shared among processes on a single processor

Q. TLB, how does it get physical address?
A. Index into TLB using virtual address, and the payload is the physical address

- Instruction cache is virtually indexed and physically tagged?
- Assume reads on this problem and there are no policies of read hit vs read miss
- You will have to worry about policy if it is a cache coherence problem

Definitions
Virtually indexed and physically tagged:

- Caching, we hadn't talked about virtual addresses yet
- Assume you had addresses in physical memory (entirety of physical addresses)
- When we access into the cache, break it down into two steps
- Find index
- Find tag
- This is physically indexed, physically tagged (PIPT)

Introducing virtual memory (this is a little slow)

- Processor is slow and this issues a memory load (virtual)

- Code is only aware of virtual addresses
- Starting with a virtual address
- You have to need to convert virtual address into a physical address
- Where can I satisfy the mapping from VA -> PA
- TLB
- Page table
- A backing storage -> page table is stored in physical memory
- Can be cached
- We can cache page table entries specifically into TLB
- Cache for page table entries
- In the best case scenario, we have our virtual address

Snooping Cache with Write-Invalidate, Write-Through Policy

- Cache where we have L2 that is write-back
- L1 is NOT write-through
- You would need to know the FSM

Write-Update

- We don't need to know low-level details of snarfing, but we do conceptually

Booth's Multiplication Algorithm

- It wouldn't be low-level but we have to know how to use it
- It will be a tradeoff question

Diyu O.H.

- Dirty bit: Only deals with write-back or write-through
- You cannot just use write-allocate (write miss policy)
- You need to specify write-hits