* Pipeline Details (warmup)
  + In single-cycle CPU design, we have a control unit that generates control signals
  + In a pipeline, that control unit “lives” in the decode stage, but most signals won’t be needed until a later stage
  + Use the pipeline buffers to “delay” the control signals until the stage they are needed
* Hazards (recap)
  + **Structural Hazard** – different stages might need simultaneous access to the same hardware units
    - Note that the example has separate memory for instructions and data
  + **Control Hazard** – current instruction determines the address of the next instruction
  + **Data Hazard** – current instruction needs results from a previous instruction before that previous instruction is finished
    - We want to read from a certain register, but the value we actually wanted isn’t there yet b/c there’s an instruction further down the pipeline that will update it
* Resolving Hazards
  + Stall or insert “bubbles”
    - Wait for the offending instruction to get through the pipeline
    - Defeats the purpose of pipelining, but often necessary
  + Add extra hardware to work around the hazards
    - If the data exists somewhere down the pipeline but just hasn’t made it back yet
  + Guess and Hope (speculation)
    - Not covering as much in this class, but huge in modern processor design
* Hazard Detection
  + How do we know when there is a hazard in the data?
    - Add $0, $1, $2
    - Add $3, $0, $0
  + Destination register of one instruction is the same as a source register of another instruction
  + The information for a particular instruction is available from the stage that instruction currently occupies
* Hazard Avoidance
  + Sometimes, a hazard is avoidable, because the value needed to resolve the dependency does exist somewhere in the pipeline
    - Just not in the register file where we expected it to be
  + Forwarding is the logic needed to choose between the actual register file contents and the contents of a later pipeline stage
* Can’t avoid everything
  + Sometimes, a hazard is unavoidable, because the value needed to resolve the dependency doesn’t exist
* Pipelined instruction sets
  + Strategy 1: let programmers/compilers know how your pipeline is built so they can optimize and use stalls to handle hazards in hardware
  + Strategy 2: Change your ISA so that pipeline stalls are impossible, and force programmers/compilers to deal with it
* MIPS strategy
  + In “real” MIPS processors, branches come one instruction late (strategy 2)
    - The instruction immediately after the branch is always executed, whether or not the branch is taken
    - To enable in MARS, go to options 🡪 Branch delay
  + All load instructions are assumed to cause a stall if you use the result in the immediately next instruction
* Time for 1000 iterations for single- and multi-cycle
  + Single cycle
    - Every instruction takes 800 picoseconds (ps)
    - Time = 5\*800\*1000 = 4001600 ps = 4001.6 ns
  + Pipeline
    - 200 ps per cycle, variable number of CPI (due to stalls)
    - Cycles = (1\*3 + 3\*4 + 1\*5)\*1000 + 2\*4 = 20008
    - Time = 20,008 \* 200 ps = 4001.6 ns
* Time for simple pipeline
  + 200 ps per cycle, 1 CPI (including nops)
  + Time = 7\*200\*1000 + 2\*200 = 1400.4 ns
* What is an “Exception”?
  + A mechanism for transitioning between user code and operating system code
    - Operating system code has privileges not available to normal user code for managing system sharing between users and applications
  + Exceptions happen for a variety of reasons
    - Program errors (exceptions)
    - Explicit requests for OS services (system calls)
    - Responding to external devices (interrupts)
  + More on this in 301
* What happens next?
  + When the OS is finished handling an exception one of three things can happen
    - The original program receives an error signal from the OS and dies
    - We come back right where we left off to resume the program
    - The OS swaps your program out for another one (because we learned to share)
  + Again, much more in 301
* What does an exception mean for our pipeline?
  + To avoid utter chaos, exceptions must be *precise*
    - Clear and consistent spot in the program where the exception “happened”
* What needs to happen?
  + Instruction A is in the execute stage, and causes a divide by 0 error
  + Instructions Y and Z are in the M and WB stages
  + Instructions B and C are in the Decode and Fetch Stages
  + New Scenario
    - Instructions A B C D and E are in the WB, M, E, D and F stages respectively
    - The processor receives an interrupt from an external device that needs attention
      * Nothing wrong with your program, the OS just needs the processor for a bit to handle something else
    - Where does the interrupt happen and what is the return address?
* Discussion question 1
  + What is the “calling convention” for an exception?
    - What values/registers need to be caller-saved before the exception happens, and what registers need to be saved by the OS after it gets control
* Discussion question 2
  + Where can we put the exception return address? What are the arguments to the exception?
    - Pretty much any register
* MIPS exception handling
  + Save address of “current” instruction
    - Save in EPC register
  + Save cause of exception
    - Save in cause register
  + Give control to appropriate address
    - Usually 0x8000 0180
    - For TLB miss use 0x8000 0000
* 4.1: Introduction
  + Subset of core MIPS instruction set
    - The memory-reference instructions *load word* (lw) and *store word* (sw)
    - The arithmetic-logical instructions, add, sub, AND, OR and slt
    - The instructions branch equal (beq) and jump (j), which we add last
  + For every instruction
    - Send the program counter (PC) to memory that contains the code and fetch the instruction from that memory
    - Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, we need to read only one register, but most other instructions require reading two registers
  + **Multiplexer** – selects source line (input) to use as output based on control lines
* 4.2: Logic Design Conventions
  + **Combinational Element** – an operational element (operates on data values) whose outputs depend only on the current input
    - i.e. AND gate or ALU
  + **State Element** – a memory element that contains state, such as a register or a memory
    - Stored even in the event of power loss
  + **Clocking Methodology** – defines when signals can be read and when they can be written
    - The approach used to determine when data is valid and stable relative to the clock
  + **Edge-triggered clocking** – any values stored in a sequential logic element are updated only on a clock edge (transition from low to high or vice versa)
* Where do our transistors go?
  + Data storage
    - Memory, register file, pipeline buffers, paging TLBs
  + Instruction control
    - Instruction dependency logic
    - Operating system enforcement, address translation
    - Writeback/precise exception unit
  + ALUs
    - 1…
* Why so lopsided?
  + Consequences of two system-level issues
    - Off-chip memory is (miserably) slow
      * Need to hold as much data on chip as possible
  + In-order, atomic instruction execution
    - Actually doing this would drastically limit performance
    - Other techniques (pipelining and out-of-order) need a lot of overhead to maintain the illusion and still get performance benefits
    - Other models of computation are much harder to support with compilers, operating systems, debuggers, handwritten assembly…
* Single instruction multiple data (SIMD)
  + Why do one addition per instruction when you can do more?
  + Supersized SIMD registers can hold multiple data elements, depending on their size
  + Instructions can operate on all elements at once
* Some x86 SSE Instructions
  + Move data between registers or between registers and memory
    - MOVUPS $xmm0, address
  + Add the floating-point values in the whole vector
    - ADDPS $xmm1, $xmm0, $xmm0
  + Add only the first elements in the vector (copy the rest from $xmm0)
    - ADDSS $xmm1, $xmm0, $xmm1
* Alignment
  + It is easier for hardware to access multiple bytes of memory if those bytes are all in one cache line
  + Best way to ensure this is to enforce alignment
    - Require address to be a multiple of X
    - Design cache line size to be a multiple of X
  + Variation of the vector move instruction that executes faster, but will cause an exception if address is not a multiple of 16
    - MOVAPS $xmm0, address
* Remember Endianness
  + X86 is a little-endian system
  + Float array[16];
  + MOVAPS $xmm0, array
* Who wants to write assembly?
  + Nobody\*
  + So, generate some vectors from high level code
  + All of our high level code uses scalars and loops
  + Compiler has to translate scalars and loops into vector operations
    - Loop vectorization
* Problem Solved!
  + No, or we wouldn’t be in this workshop
  + Compilers don’t automatically vectorize everything
    - Programmer often knows more about what is really going on than the compiler can figure out
    - Loop has to be reorderable, with no loop-carried dependencies
  + Compilers sometimes vectorize poorly
    - Alignment, in particular, is very hard to prove if the program does not help you
* How can we help the compiler?
  + Alignment attributes on arrays
  + Remove obvious loop-carried dependencies
  + Remove non-obvious, possible loop carried dependencies
    - Array aliasing and pointer restrictions
  + Make sure loop counts are obviously a multiple of the vector size
  + Even then, hard to tell how well the compiler will really do
* How can the compiler help us?
  + Intrinsics – allow us to explicitly mix vector instruction into the rest of our code without having to write assembly
  + Vector Types – allow standard C operators on vector types, and let the compiler figure out the right instructions
* <xmmintrin.h>
  + standard header file that declares types and functions
  + each function corresponds directly to a particular x86 SSE vector instruction that can only operate on vector types
* Vector types
  + \_m128 items are opaque
    - no way to examine their contents without casting
  + Compilers are actually pretty good at instruction selection
    - Why not let them do it, based on high level operations on vectors?
  + This is the major transition point from generic operations to a specific platform, so letting tools do it makes our code much more portable
* Summary
  + Vectorization is work
    - How much benefit do you need?
    - How much benefit might you expect from making some fraction of your instructions faster?
  + Vectorization used to be platform-specific work
    - X86 SSE, ARM NEON, MIPS SIMD Architecture (MSA)
  + With more interest, things are moving toward supporting manual, portable SIMD programming
    - Increasing importance, decreasing idolization of compiler capabilities
* SRAM (Static Random Access Memory)
  + Basically registers
  + Speed mostly depends on size
  + Needs active power
* DRAM
  + Capacitor storage
  + Write select = 1
    - “pressurize” or “vaccum” to store a bit
  + Read: select = 1, use data to sense “pressure” (voltage)
  + Slow
  + Technically no active power is needed
    - Needs regular refresh
* Pipeline
  + Registers – small SRAM
  + Ll Cache – small SRAM (64-128 KB)
  + Last-level cache – large SRAM (8-16 MB)
* Off-chip
  + DRAM
  + Hard Disk
* Locality – caches work because programs do not access memory randomly
  + Spatial locality – if I access address A I am likely to access addresses near A in the future
  + Temporal locality – accesses are repeated
* Caches in hardware
  + Hash memory addresses into a buffer, and check whether the item stored in that buffer matches the address you want
    - Simplest caches just take a few address bits as the hash function
  + Access procedure
    - Determine where in the cache the address should be
    - Check the tag for that cache line
      * Match, cache hit, perform the access
      * Match, cache miss, load new data from memory and update the tag
* Optimizations
  + In the previous example, we have 32 bits of “overhead” for every address being stored
  + We can minimize overhead in two ways:
    - Don’t store the full address: some of the bits are predetermined just by the hash function
    - Store multiple words of data with the same tag
* Problems with direct caches
  + Same problems as with all hash tables
    - Yes, 102 is relevant to hardware
    - Conflicts: multiple things we care about might hash to the same location
    - Bad hash functions means it’s really easy to come up with worse-case conflict patterns
      * Taking a few bits of the address is a bad hash function
  + So, how do we solve it?
    - Sequential lookups are bad. Remember we’re on the clock! Plus, we’d rather have the cache fully utilized
    - Allow multiple items (tags) per bucket (line)
* Cache replacement policies
  + In an associative cache, we have choices about which line to evict when we miss and need to bring new data in
  + The perfect eviction choice is to throw out the line that will be accessed furthest in the future
  + We can’t know the future
  + So, we try to predict the future behavior based on past events
* More replacement policies
  + Least recently used
    - Keep track of the line that was accessed longest ago, and evict that one
    - Bookkeeping gets hard with a large number of sets
  + Random
    - Just pick one
    - Reasonable chance of evicting exactly the thing we need most
  + Not most recently used
    - Just keep track of what’s important and throw out something else
* Design Space for caches
  + Fully associative
    - Check all tags at once (no line bits)
    - Very expensive in hardware
  + Direct cache
    - Simple hash, one entry per line allowed
    - Bad conflicts easy without a good hash function, and some conflicts are inevitable
  + Set-associative cache
    - Split the difference, reduce conflicts at some access-time expense

Don’t need to be able to construct this in hardware

* + Just understand the concepts in modern processing
* What we’re not talking about
  + With OS support, all programs on a single-core processor share that processor
  + However, only one thread is active at a time
    - The hardware has only one “active context” for a currently running thread
    - The operating system is responsible for switching the active thread in the single active context
  + This is not hardware multithreading
    - Hardware multithreading means supporting more than one thread context on a single core
* Present a view to the operating system as if there are multiple cores
  + OS treats the system as if there were two active processors
  + Both threads share same execution units
  + Essentially one core that runs multiple threads
* Why multiple threads?
  + We want our execution units to be fully occupied with useful work 100% of the time
    - No stalls
  + The programmer/compiler can’t fill all stalls itself
    - Cache and TLB misses are hard to predict
      * Hardware will not be fully utilized unless it figures out a new approach to handling stalls
    - Dynamically scheduled pipelines take scheduling control away from the compiler/programmer
* First, ground rules
  + When we support two threads, they need to be supported as completely independent threads in the operating system
    - Different programs/instructions
    - Shared memory, private registers
    - Different memory maps
  + If we are going to support multiple threads, which of the following components need to be copied, and which can be shared/used by instructions from any thread?
    - PC – can’t be shared (need 2)
    - Caches – can be shared
    - TLBs – could potentially be shared but better off with 2
    - Register File – can’t be shared (need 2)
    - ALUs – can be shared
      * Whole point is to keep ALUs as busy as possible (enhances efficiency)
* Sample pipeline
  + Pc (2) 🡪 TLB (2) 🡪 Instruction cache (1) 🡪 decode (1) 🡪 regfile 🡪 execute (1) 🡪 TLB (2) 🡪 Data cache (1)
* Priorities for mixing threads
  + Increase instruction throughput in the hardware
    - Reduce stalls and unused instruction slots
  + Avoid increasing an individual thread’s latency where possible
    - Where does one thread start and finish?
  + Keep the hardware simple
  + These goals are in tension with each other, but we will evaluate different solutions based on how well they achieve these goals
* Swap-on-stall
  + Fetch using the PC for one thread until it hits a stall, then switch to another thread until the stall has passed
  + Hardware needed
    - Stall detection
    - Per-cycle selection in each fetch
  + How well does this meet our goals?
    - Increases throughput somewhat (not amazing); 2 instructions per cycle whereas 4 is ideal
    - Latency – thread 1 isn’t slowed down
      * Thread 2 is pretty bad but better than it would’ve been in a single thread design
* Rotating schedule
  + Each cycle, fetch from alternate threads, skipping one if it is already stalled
  + Hardware needed
    - Stall detection
    - Per-cycle selection in each fetch
  + Goals?
    - Throughput – about the same as swap-on-stall
      * Main difference is that interleaving allows for pipeline design without worrying about stalls for an individual thread
      * Ends up being better than swap-on-stall if there are a large number of small stalls
    - Decreases latency for thread 1
    - Hardware is the same as swap-on-stall
* Interleaved
  + Each cycle, mix instructions from different threads
  + Hardware needed
    - Same as first two
  + How well do we meet goals?
    - Throughput – better than others: best you can do for these threads --- the more threads you have the easier/better this works
    - Latency – no increase for thread 1
    - Hardware – worse
      * Need to deal with multiple instructions for multiple threads simultaneously
      * For every instruction, which thread does it come from?
  + Still ends up being worthwhile, despite the hardware

Real Hardware examples

* + AMD Radeon 7000 GPUs
    - Swap on stall (with compiler support to “plan” stalls)
  + NVIDIA GeForce 400 GPUs
    - Mixture of swap-on-stall and rotating
  + Most modern, out of order CPUs
    - Interleaved schedule
* There is no one correct solution
  + This is a general principle of system design!
  + The “correct” way to implement multithreading depends on context
    - How important is each of the goals?
    - How many threads will be actively running?
  + More threads make swap-on-stall and rotating schedules more effective
  + Interleaving isn’t needed if you use SIMD instead of superscalar hardware to increase throughput
* Experiments
  + Methodology
    - What is the procedure?
  + Metrics
    - What are we measuring?
  + Reporting
* Benchmark considerations
  + Measurement error/variation
* What might effect our performance?
  + CPU strength – processor
  + ALU
  + Cache (part of processor)
  + RAM
    - DRAM
    - SRAM
  + Registers
  + Hard Disks
  + Operating System
  + Interfering applications
  + Compiler/interpreter
  + Libraries
  + Cooling system
  + Ambient room temperature
  + GPU
  + Network card/ISP
* Network download throughput
* Video game performance
* Scientific simulation
* Webserver workload (facebook)
* Database
* Cache effectiveness
* 1.) think about: for the situation given
  + define metric(s), including units, that you would care about
* 2.) talk about what experiment you would run to measure that value in a meaningful way

Network download throughput

* + metrics
    - mbps
    - pages/files downloaded per second
  + Download a bunch of files; see how long each one takes
    - Different groupings of files
    - Multiple files at the same time
    - Try 1 gb, 2 gb, 3gb etc.
    - Try 1 file, 2 files, 3 files etc.
      * Same size
    - Do torrents allow for faster speeds?
      * File vs. torrented file
  + Need to consider
    - Internet speed
    - Number of users on the network
* Problems
* Algorithms
* Software
* Operating Systems
* Assembly/Machine Code
* Architectures
* Transistors
* Shield/dielectric
* photoresist