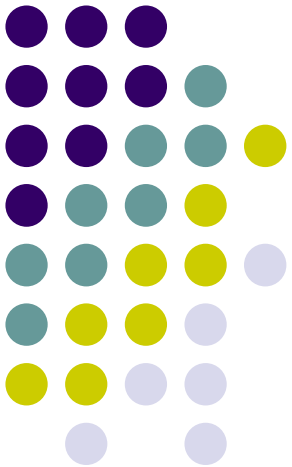
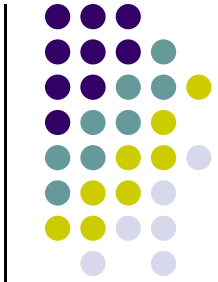
ECE/ME/EMA/CS 759
High Performance Computing for Engineering Applications

Pipelining Hazards
Timing Program Execution
Memory Aspects: The Cache



September 11, 2015

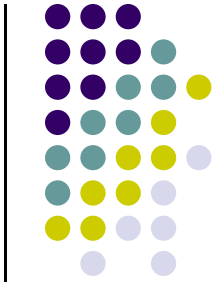
Quote of the Day



“Inspiration exists, but it has to find you working”

Pablo Picasso (1881-1973)

Before We Get Started

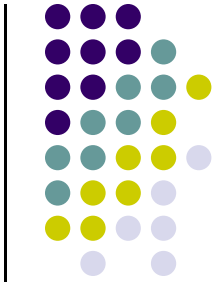


- Issues covered last time:
  - CMake (Hammad)
  - Registers
  - Pipelining

- Today's menu
  - Pipelining hazards
  - Timing program execution
  - Memory aspects: the cache

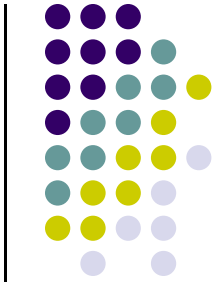
- Assignment:
  - HW02 – Due next Wd at 11:59 PM, use dropbox at Learn@UW

Before We Get Started



- Other issues:
  - Late homework – I'm sorry but I can't help you on this one
    - Not fair to me and the rest of your colleagues to bend the rules
    - Becomes a slippery slope
    - Recall that you can drop two homeworks this semester
      - Don't worry, it'll be fine (just work hard)
  - Accessing the video recording of a lecture
    - You need a CAE account
    - If you are not an Engineering student you can get an account by registering here: http://www.cae.wisc.edu
  - Calling me Dan is swell

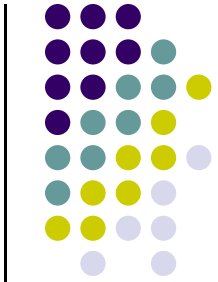
Pipelining Hazards



- Q: if deep pipelines are so good, why not have them deeper and deeper?
- A: deep pipelines plagued by “pipelining hazard”
  - These “hazards” come in three more common flavors
    - **Structural hazards**
    - **Data hazards**
    - **Control hazards**

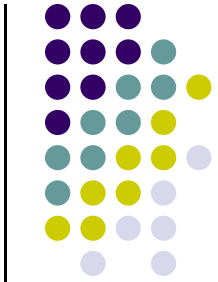
[Patterson, 4th edition]→

Pipeline Structural Hazards [1/2]



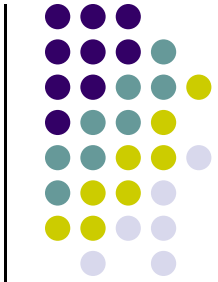
- The instruction pipelining analogy w/ the vehicle assembly line breaks down at the following point:
  - A real world assembly line assembles the same product for a period of time
  - Might be quickly reconfigured to assemble a different product
  - Instruction pipelining must process a broad spectrum of instructions that come one after another
    - Example: A J-type instruction coming after a R-type instruction, which comes after three I-Type instructions
    - If they were the same instructions (vehicles), designing a pipeline (assembly line) is straightforward
- A structural hazard refers to the possibility of having a combination of instructions in the pipeline that are contending for the same piece of hardware
  - Not encountered when you assembly the same car model (things are deterministic in this case)

Pipeline Structural Hazards [2/2]



- Possible Scenario: you have a six stage pipeline and the instruction in stage 2 calls for a `sin(x)` function evaluation while instruction currently in stage 3 computes a `log(x)` function
  - Possible solutions:
    - Add one more SFU (Special Function Unit)
    - Introduce a bubble in the pipeline
  - Note:
    - Adding one more SFU is daunting (requires a chip design change)
    - Stalling the execution of an instruction via a pipeline bubble at run time: is a run-time solution that is inexpensive but slows down the execution

Pipeline Data Hazards [1/2]

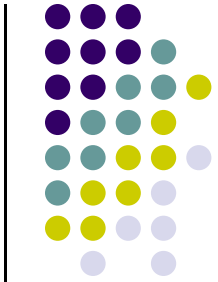


- Consider the following example in a five stage pipeline setup:

```
add  $t0, $t2, $t4    #  $t0 = $t2 + $t4
addi $t3, $t0, 16     #  $t3 = $t0 + 16 (“add immediate”)
```

- The first instruction is processed in five stages
- Its output (value stored in register `$t0`) needed in the very next instruction
- Data hazard: unavailability of `$t0` to the second instruction, which references this register
- Resolution (less than ideal)
  - Pipeline stalls to wait for the first instruction to fully complete

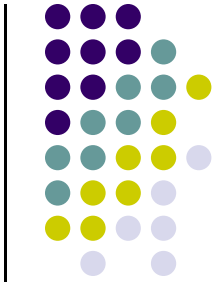
Pipeline Data Hazards [2/2]



```
add  $t0, $t2, $t4    #  $t0 = $t2 + $t4
addi $t3, $t0, 16     #  $t3 = $t0 + 16 (“add immediate”)
```

- Alternative [the good] Resolution: use “forwarding” or “bypassing”
- Key observation: the value that will eventually be placed in $t0 is available after stage 3 of the pipeline (where the ALU actually computes this value)
- Provide the means for that value in the ALU to be made available to other stages of the pipeline right away
  - Nice thing: avoids stalling - don’t have to wait several other cycles before the value made its way in $t0
  - This process is called a forwarding of the value
- Supporting forwarding does not guarantee resolution of all scenarios
  - In relatively rare occasions the pipeline ends up stalled for a couple of cycles
- Note that the compiler can sometimes help by re-ordering instructions
  - Not always possible

Side Trip: Register Renaming



- In addition to register forwarding, “register renaming” is another trick used if the resource that is contended is a register

- What’s the trick?
  - There are registers that are regarded as wild cards. When available, they can become whatever is needed

  - A wild-card register can be temporarily named exactly what’s needed and be used to advance the execution of the instructions in the pipeline

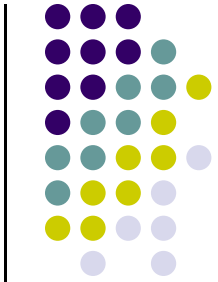
  - Temporary name, gets changed to something else soon thereafter

```
1: regA = 4
2: regB = memory[regA]
3: regA = 5
4: regC = 2
5: regD = regA + regC
```

register renaming

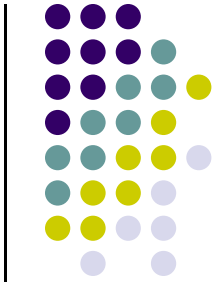
```
1: regA = 4
2: regB = memory[regA]
3: regW = 5
4: regC = 2
5: regD = regW + regC
```

Pipeline Control Hazards [Setup]



- What happens when there is an "`if`" statement in a piece of C code?
- A corresponding machine instruction decides the program flow
  - Specifically: should the "`if`" branch be taken or not?
- Processing this very instruction to figure out the next instruction (branch or no-branch) will take a number of cycles
- Should the pipeline stall while this instruction is fully processed and the branching decision becomes clear?
  - If yes: approach works, but it is slow
  - If no: you rely on branch prediction and proceed fast but cautiously

Pipeline Control Hazards: Branch Prediction

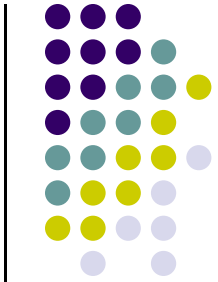


- Note that when you predict wrong you have to discard instruction[s] executed speculatively and take the correct execution path

- Static Branch Prediction (1st strategy out of two):
  - Always predict that the branch will not be taken and schedule accordingly
  - There are other heuristics for proceeding: for instance, for a do-while construct it makes sense to always be jumping back at the beginning of the loop
    - Similar heuristics can be produced in other scenarios (a “for” loop, for instance)

- Dynamic Branch Prediction (2nd strategy out of two):
  - At a branching point, the branch/no-branch decision can change during the life of a program based on recent history
  - In some cases branch prediction accuracy hits 90%

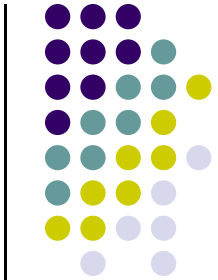
Pipelining vs. Multiple-Issue



- Multiple-Issue: just like pipelining, yet another approach to speeding up execution
  - It's different than pipelining
- A Multiple-Issue processor core is capable, *on average*, of issuing for execution more than one instruction per cycle
- Two examples to show when multiple-issue can kick in
  - Example1: performing an integer operation while performing an unrelated log function evaluation – they require different resources and therefore can proceed simultaneously
  - Example2: the two lines of C code below lead to a set of instructions that can be executed at the same time

```
int a, b;
float c, d;
//some code to compute a, b, c, d
a += b;
c += log(fabs(d)+1);
```

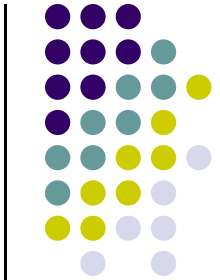
Pipelining vs. Multiple-Issue



- On average, more than one instruction is issued for execution on the same CPU core in the same clock cycle possibly from different instruction streams

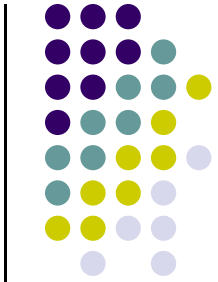
- Multiple-Issue can be done statically or dynamically
  - Static multiple-issue of instructions:
    - Predefined, doesn't change at run time
    - Who uses it: NVIDIA - very common in parallel computing on the GPU
      - One warp of threads executes in locked-step fashion
  - Dynamic multiple-issue of instructions:
    - When and what instructions can be paired is figured out at run-time based on hardware resources that can take additional work
    - Who uses it: Intel, very common

Comments on Dynamic Multiple-Issue



- Checking for dependencies in the same instruction stream is complex, requires high cost in time and energy
- Checks must always be in place to make sure result is ok and nothing goes south owing to a multiple issue event
- NOTES:
  - Multiple issue an attribute of what is called a “superscalar architecture”
  - Both pipelining and multiple-issue are presentations of what is called “Instruction-Level Parallelism” (ILP)

Intel's HTT Feature



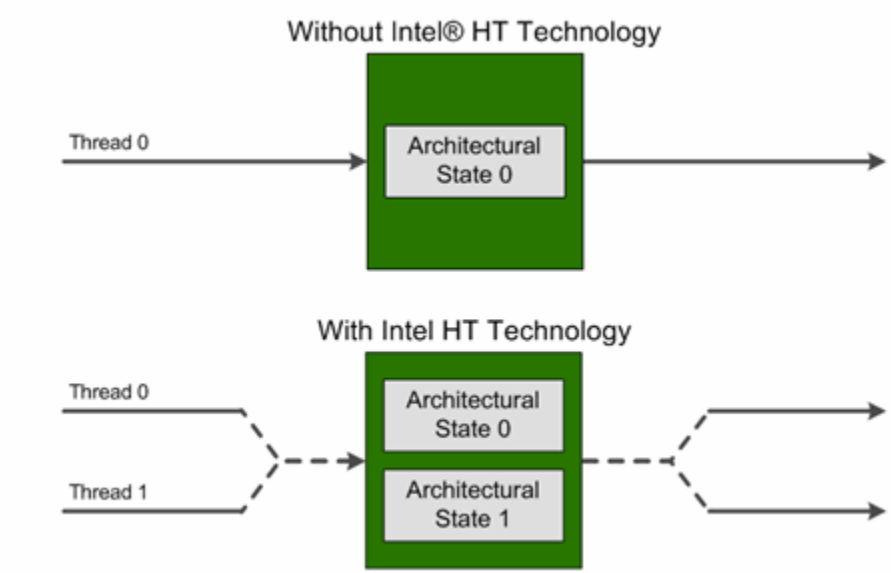
- Typical Intel core supports the "Hyper-Threading Technology" (HTT)

- HTT: two threads, each with its own machine instruction stream, active at any given time
- The scheduler tries to issue instructions from both instruction streams
  - The HW upgrades to support HTT are relatively minor
    - Required to save execution state of a thread when it's idle and not running for lack of data of lack of functional units available

- Why a big deal?
  - Dependencies between different streams of instructions: in theory, there should be no dependency

[Intel]→

HTT, Nuts and Bolts

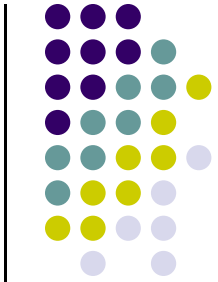


- A processor core to maintain two architectural states, each of which can support its own execution thread

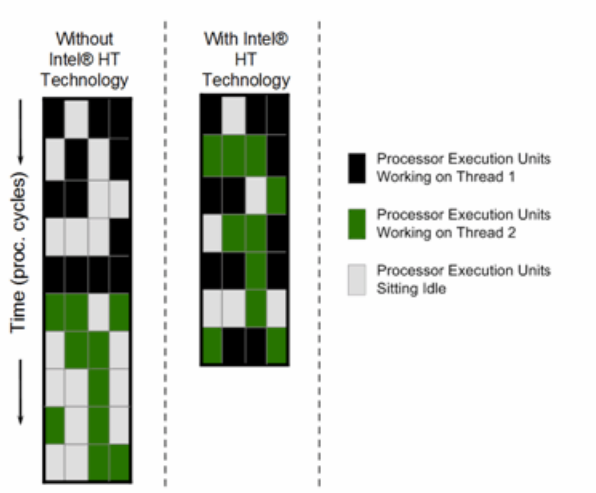
- Many of the internal microarchitectural hardware resources are shared between the two threads

[Intel]→

HTT: The Bottom Line

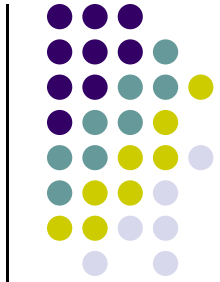


- Essentially, one physical core shows up as two virtual cores



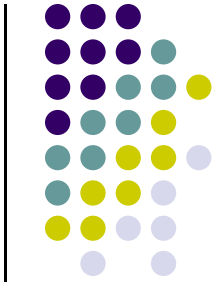
- Why is it good?
  - More functional units are used at any given time
    - Higher utilization rate of the hardware assets
  - When one execution thread stalls (has a cache miss, branch mispredict, pipeline bubble, etc.), the other execution thread continues processing instructions at nearly the same rate as a single thread running on the core

[Intel]→



Measuring Computing Performance

Nomenclature



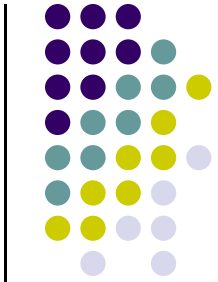
- Program Execution Time – sometimes called wall clock time, elapsed time, response time
  - Most meaningful indicator of performance
  - Amount of time from the beginning of a program to the end of the program
  - Includes all the housekeeping (running other programs, OS tasks, etc.) that the CPU has to do while running the said program

- CPU Execution Time
  - Like “Program Execution Time” but counting only the amount of time that is effectively dedicated to the said program
  - Requires a profiling tool to assess (like `gprof`, for instance)

- On a dedicated machine; i.e., a quiet machine, Program Execution Time and CPU Execution Time would virtually be identical

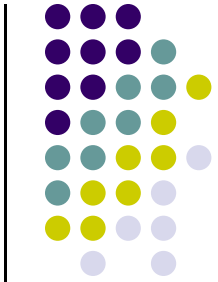
[Patterson, 4th edition]→

Nomenclature [Cntd.]



- Qualifying CPU Execution Time further:
  - User time – the time spent processing instructions compiled out of code generated by the user or in libraries that are directly called by user code
  - System time – time spent in support of the user's program but in instructions that were not generated out of code written by the user
    - OS support: open file for writing/reading, throw an exception, etc.
  - The line between the user time and system time is somewhat blurred, hard to delineate these two times at times
- Clock cycle, clock, cycle, tick – the length of the period for the processor clock; typically a constant value dictated by the frequency at which the processor operates
  - Example: 2 GHz processor has clock cycle of 500 picoseconds

The CPU Performance Equation



- The three ingredients of the CPU Performance Equation:
  - Number of instructions that your program executes (Instruction Count)
  - Average number of clock-cycles per instructions (CPI)
  - Clock Cycle Time

- The CPU Performance Equation reads:

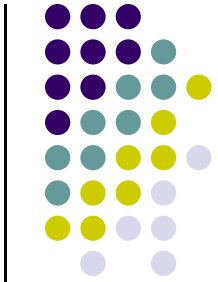
$$\text{CPU Exec. Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

- Alternatively, using the clock rate

$$\text{CPU Exec. Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

[Patterson, 4th edition]→

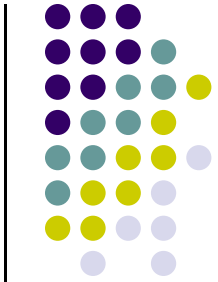
CPU Performance: How can we improve it?



- To improve performance the product of three factors should be reduced
- For a long time, we surfed the wave of “let’s increase the frequency”; i.e., reduce clock cycle time
  - We eventually hit a wall this way (the “Power Wall”)
- As repeatedly demonstrated in practice, reducing the Instruction Count (IC) often times leads to an increase in CPI. And the other way around.
  - Ongoing argument: whether RISC or CISC is the better ISA
    - The former is simple and therefore can be optimized easily. Yet it requires a large number of instructions to accomplish something in your C code
    - The latter is mind boggling complex but instructions are very expressive. Leads to few but expensive instructions
    - Specific example: ARM vs. x86

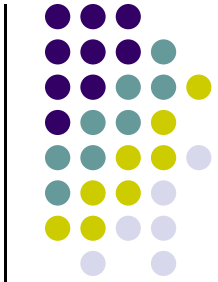
[You believe you improved performance of your chip. Prove it!]

SPEC CPU Benchmarks



- Several benchmarks used to gauge the performance of a chip
- Idea: gather a collection of programs that use a good mix of instructions and flex the muscles of the chip
- These programs are meant to be representative of a class of applications that people are commonly using and not favor a chip manufacturer at the expense of another one
- Example: a compiler is a program that is used extensively, so it makes sense to have it included in the benchmark
- Two common benchmarks:
  - For programs that are dominated by floating point operations (CFP2006)
  - A second one is meant to be a representative sample of programs that are dominated by integer arithmetic (CINT2006) – see next slide

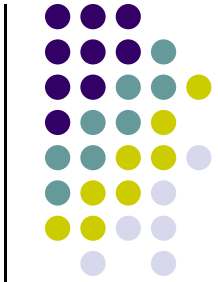
SPEC CPU Benchmark:
Example, highlights AMD performance



CINT2006 Programs		AMD Opteron X4 – 2356 (Barcelona)			
Description	**Name**	**Instruction count [$\times 10^9$]**	**CPI**	**Clock Cycle Time [seconds $\times 10^{-9}$]**	**CPU Exec. Time [seconds]**
Interpreted string processing	perl	2118	0.75	0.4	637
Block-sorting compression	bzip2	2389	0.85	0.4	817
GNU C compiler	gcc	1050	1.72	0.4	724
Combinational optimization	mcf	336	10.00	0.4	1,345
Go game (AI)	go	1658	1.09	0.4	721
Search gene sequence	hmmer	2783	0.80	0.4	890
Chess game (AI)	sjeng	2176	0.96	0.4	837
Quantum computer simulation	libquantum	1623	1.61	0.4	1,047
Video compression	h264avc	3102	0.80	0.4	993
Discrete event simulation library	omnitpp	587	2.94	0.4	690
Games/path finding	aster	1082	1.79	0.4	773
XML parsing	xatancbmk	1058	2.70	0.4	1,143

[Patterson, 4th edition]→

SPEC CPU Benchmark:
Example, highlights AMD performance

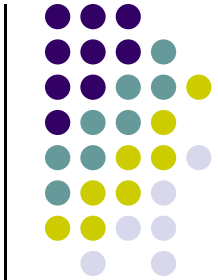


- Comments:
  - There are programs for which the CPI is less than 1.
    - Suggests that multiple issue is at play
  - Why are there programs with CPI of 10?
    - The pipeline stalls a lot, most likely due to repeated cache misses and system memory transactions

[Patterson, 4th edition]→

The ME759 Most Important Lesson

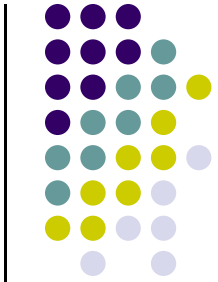
[You can now drop this class and miss nothing.]



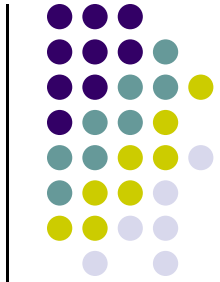
- The cost of memory transactions trumps by far the cost of computation

- Computation is free, the art is in providing data fast to sustain computation

Summary of Topics Covered

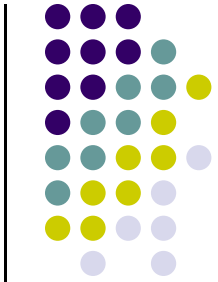


- Von Neumann computational model
- From code to machine instructions
- Instruction Set Architecture (ISA)
- Transistors as building blocks for control/arithmetic/logic units
- Microarchitecture
- Registers
- Pipelining
  - Structural hazard
  - Data hazard
  - Control hazard
- Performance metrics for program execution



Memory Aspects

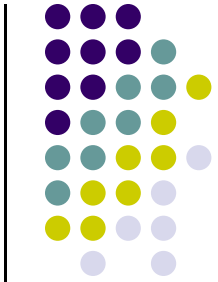
SRAM



- SRAM – Static Random Access Memory
  - Integrated circuit whose elements combine to make up memory arrays
  - “Element”: is a special circuit, called flip-flop
  - One flip-flop requires four to six transistors
  - Each of these elements stores on bit of information
  - Very short access time: ≈1 ns (order of magnitude)
  - Uniform access time of any element in the array (yet it’s different to write than to read)
  - “Static” refers to the fact that once set, the element stores the value set as long as the element is powered
  - Bulky, since a storing element if “fat”; problematic to store a lot per unit area (compared to DRAM)
  - Expensive, since it requires four to six more transistors and different layout and support requirements

[Patterson & H]→

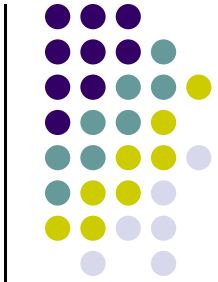
DRAM



- DRAM type memory: the signal is stored as a charge in a capacitor
  - No charge: 0 signal
  - Some charge: 1 signal
- The good: cheap, requires only one capacitor and one transistor
- The bad: capacitors leak, so the charge or lack of charge should be reinforced every so often, from where the name “dynamic” RAM
  - State of the capacitor should be refreshed every millisecond or so
  - Refreshing requires a small delay in memory accesses
- Is this delay incurred often? (first order approximation answer)
  - Given frequency at which memory is accessed, refreshing every millisecond means issues might appear once every million cycles
  - Turns out that 99% of memory cycles are useful; refresh operations consume 1% of DRAM memory cycles

[Patterson & H]→

SRAM vs. DRAM

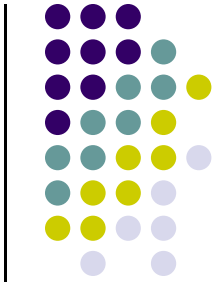


- Order of the SRAM access time: 0.5ns
  - Expensive but fast
  - Mostly used on chip for caches
  - Needs no refresh
- Order of the DRAM access time: 50ns
  - Less expensive but slow
  - Mostly used off chip (system memory)
  - Higher capacity per unit area
  - Needs refresh every 10-100 ms
  - Sensitive to disturbances
- Limit case: a 100X speedup if you can work off the SRAM

	Transistors per bit	Access Time	Persistent?	Sensitive?	Price	Applications
SRAM	6	1X	Yes	No	100X	Cache memories
DRAM	1	10X	No	Yes	1X	Main Memory

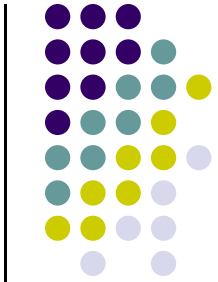
[Rauber&Runger]→

Feature Comparison Between Memory Types



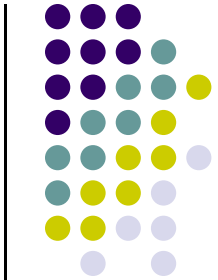
	SRAM	DRAM	Flash
Speed	Very fast	Fast	Very slow
Density	Low	High	Very high
Power	Low	High	Very low
Refresh	No	Yes	No
Retention	Volatile	Volatile	Non-volatile
Mechanism	Bi-stable Latch	Capacitor	Fowler-Nordheim tunneling

Cost and Speed Implications



- SRAM: bulkier and expensive → can't have too much
  - Plagued by Space & Cost constraints
- Compromise:
  - Have some SRAM on-chip, making up what is called the "cache"
  - Have a lot of inexpensive DRAM off-chip, making up the "main memory"
    - Also called "system memory"
- Hopefully your program has a low "average memory access time" by hitting the cache repeatedly instead of taking costly trips to main memory

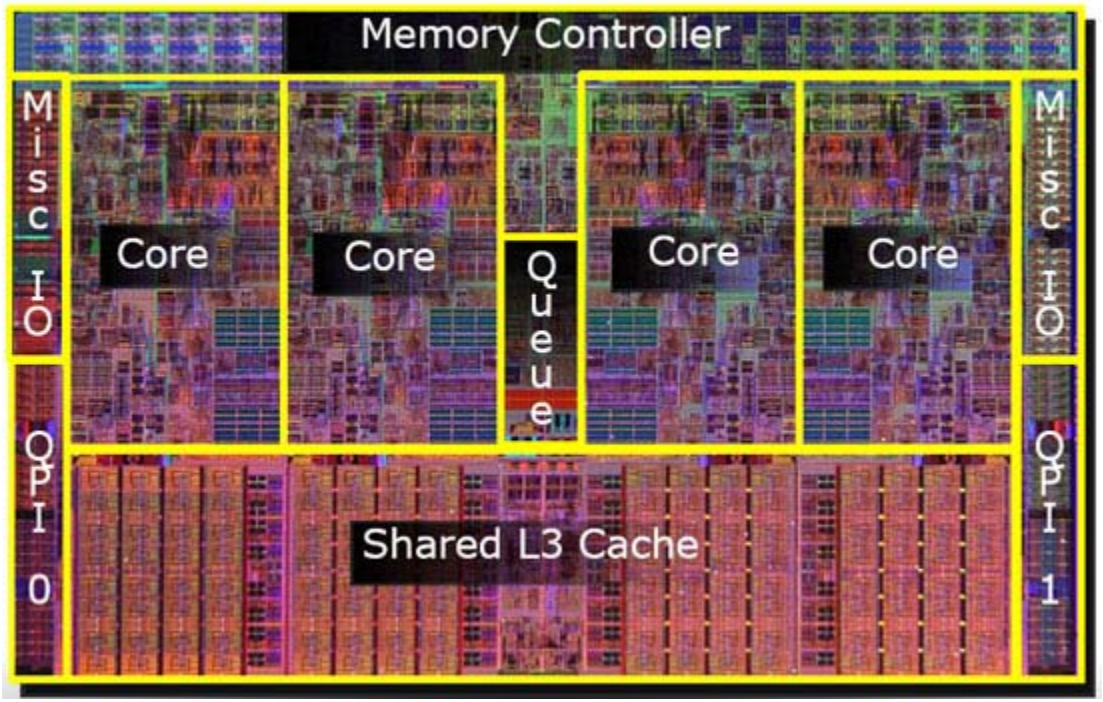
Fallout: Memory Hierarchy



- You now have a “memory hierarchy”
- Simplest memory hierarchy:
  - Main Memory + One Cache (typically called L1 cache)
- Today’s memory architectures typically have deeper hierarchy: L1+L2+L3
  - L1 faster and smaller than L2
  - L2 faster and smaller than L3
- Note that all caches are typically on the chip

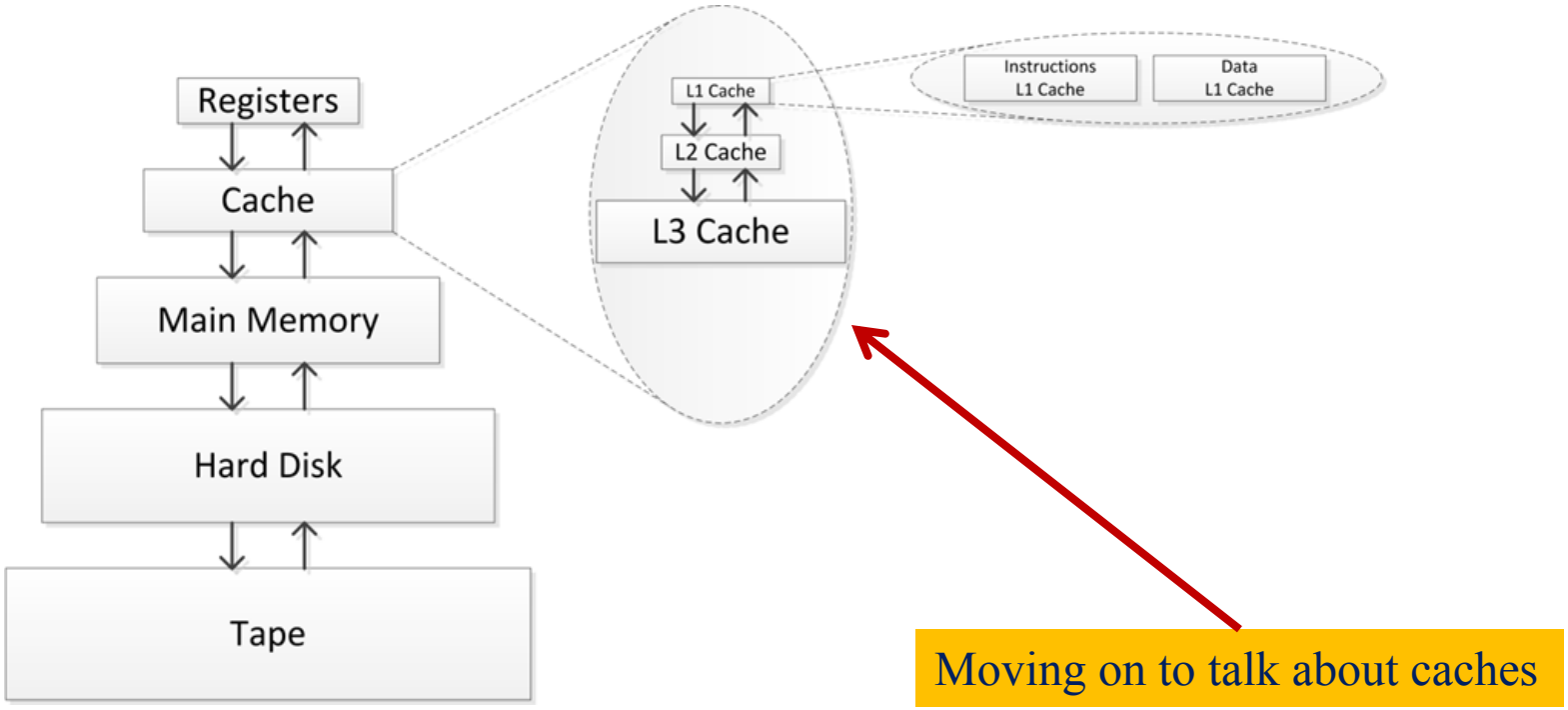
Example: Intel Chip Architecture

- Quad core Intel CPU die that illustrates L3 cache
- Intel Core I7 975 Extreme, cache hierarchy
  - 32 KB L1 cache / core
  - 256 KB L2 (Instruction & Data) cache / core
  - 8 MB L3 (Instruction & Data) shared by all cores

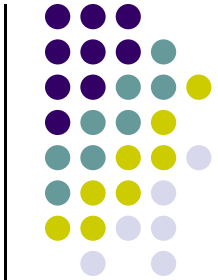


Memory Hierarchy

- Memory hierarchy is deep:



Cache Types



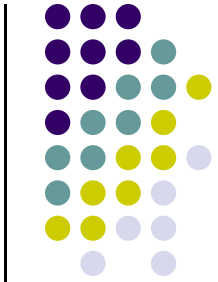
- Two main types of cache

- Data caches feed processor with data manipulated during execution
  - If processor would rely on data provided by main memory the execution would be pitifully slow
    - Processor Clock faster than the Memory Clock
    - Caches alleviate this memory pressure

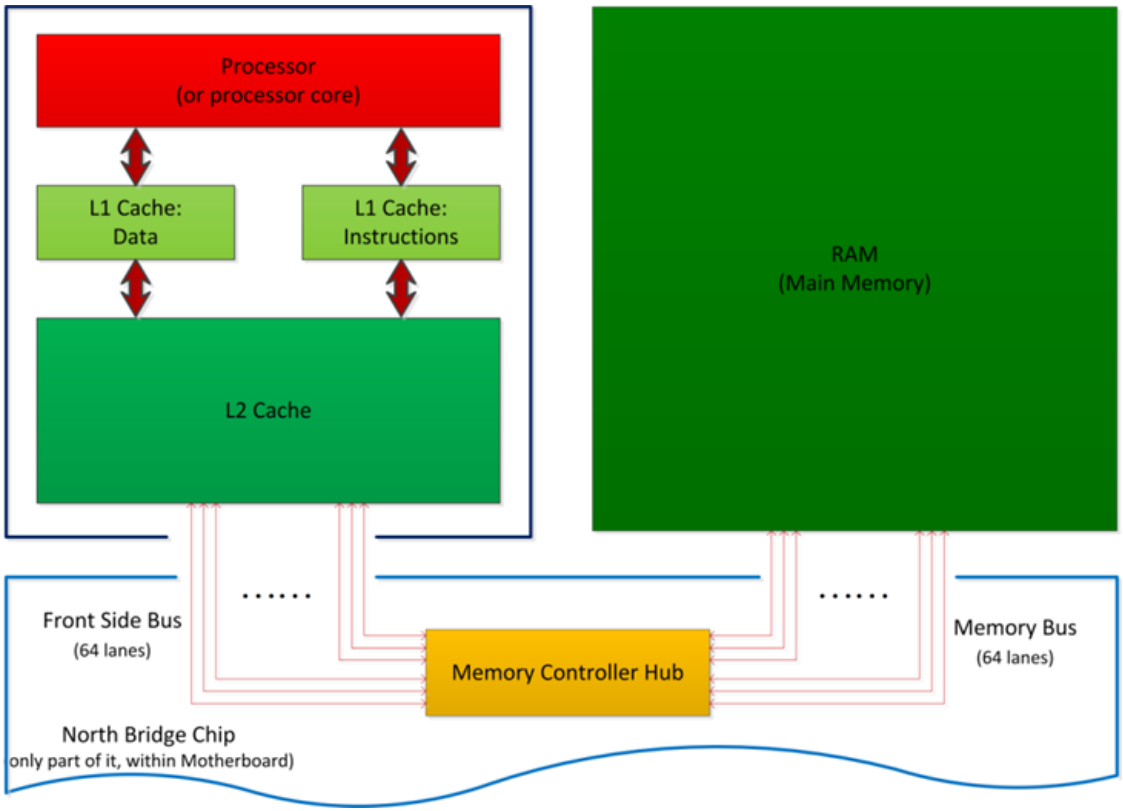
- Instruction caches: used to store instructions
  - Much simpler to deal with compared to the data caches
    - Instruction use is much more predictable than data use

- In an ideal world, the processor's request would be met by data that already is in cache. Otherwise, a trip to main memory is in order

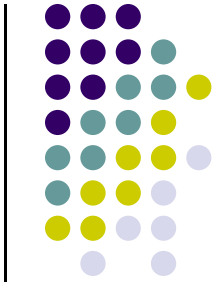
Split vs. Unified Caches



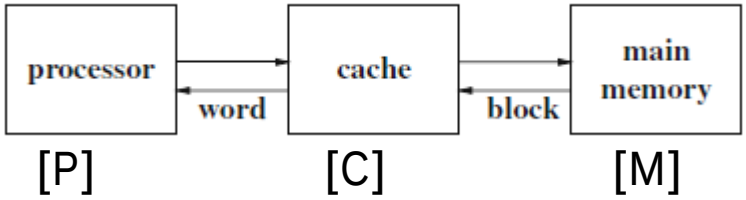
- Note that in the picture below L1 cache is split between data and instruction, which is typically the case
- L2 and L3 (when present) typically unified



How the Cache Works



- Assume simple setup with only one cache level L1

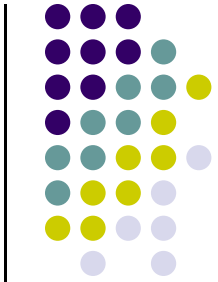


- Purpose of the cache: store for fast access a subset of the data stored in the main memory

- Data is moved at different resolutions between P → C and between C → M and
  - Between P and C: moved one word at a time
  - Between C and M: moved one block at a time (block called “cache line”)

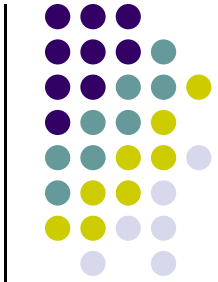
[Rauber&Runger]→

Cache Hit vs. Cache Miss



- The processor typically agnostic about memory organization
- Middle man is the cache controller, which is an independent entity: it enables the "agnostic" attribute of the P → M interaction
  - Processor requires data at some address
  - Cache Controller figures out if data is in a cache line
    - If yes: cache hit, processor served right away
    - If not: cache miss (data brought over from main memory – very slow!)
    - Difference between cache hit and cache miss:
      - Performance hit related to SRAM vs. DRAM memory access plus overhead
  - On more advanced architectures data can be "pre-fetched"

More on Cache Misses…



- A cache miss refers to a failed attempt to read/write a piece of data from/to the cache, which results in a main memory access with much longer latency

- There are three kinds of cache misses:
  - Cache read miss from an instruction cache: generally causes the most delay, because the processor, or at least the thread of execution, has to wait (stall) until the instruction is fetched from main memory

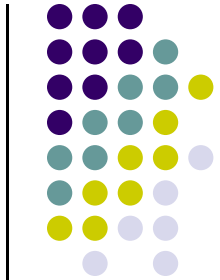
  - A cache read miss from a data cache: usually causes less delay, because instructions not dependent on the cache read can be issued and continue execution until the data is returned from main memory, and the dependent instructions can resume execution.

  - A cache write miss to a data cache: generally causes the least delay, because the write can be queued and there are few limitations on the execution of subsequent instructions. The processor can continue unless the queue is full and then it has to stall for the write buffer to partially drain.

[Wikipedia]→

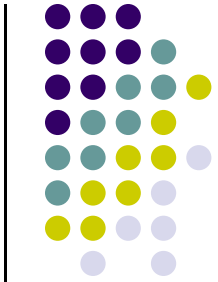
[It makes sense to ask this]

Question:



- Can you control what's in the cache and anticipate future memory requests?
  - Typically not…
    - Any serious system has a hardware implemented cache controller with a mind of its own
  - There are ways to increase your chances of cache hits by designing software for high degree of memory access locality
  - Two flavors of memory locality:
    - Spatial locality
    - Temporal locality

Spatial and Temporal Locality

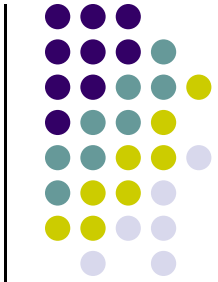


- Spatial Locality for memory access by a program
  - A memory access pattern characterized by bursts of repeated requests for data that is physically located within the same memory region
  - “Bursts” because this accesses should happen in a sufficiently short interval of time (otherwise the cache line gets evicted)

- Temporal Locality for memory access by a program
  - Idea: If you access a variable at some time, then you’ll probably keep accessing the same variable for a while
  - Example: have a `for` loop with some variables inside the loop → you keep accessing those variables as long as you loop

Cache Characteristics

[Not covered here]



- Size attributes: absolute cache size and cache line size
- Strategies for mapping memory blocks to cache lines
- Cache line replacement algorithms
- Write-back policies

- NOTE: these characteristics carry over and become more convoluted when dealing with multilevel cache hierarchies