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

Memory Aspects: The Cache

Virtual Memory

The Shift to Parallel Computing

September 14, 2015

Quote of the Day

“Computers are good at following instructions, but not at reading your mind.”

Donald Knuth

Before We Get Started

- Issues covered last time:
  - Pipelining hazards
  - Timing program execution

- Today's topics
  - Memory aspects: the cache
  - The virtual memory
  - The shift to parallel computing

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

Summary of Topics Covered

- Von Neumann computational model
- From code to machine instructions
- ISA and Microarchitecture aspects
- Transistors as building blocks for control/arithmetic/logic units
- 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	10-100X	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

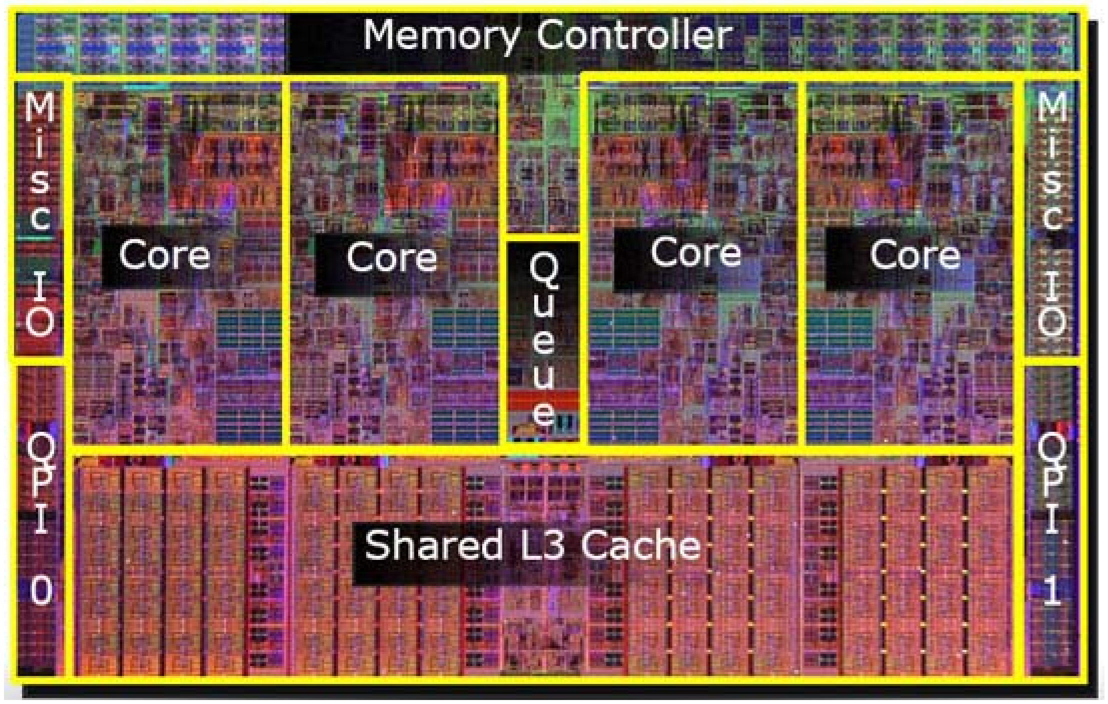
- SRAM: bulkier and expensive → can't have too much
  - Plagued by Space & Cost constraints
- Compromise:
  - Have some SRAM on-chip, making up what is called the "cache"
  - Have a lot of inexpensive DRAM off-chip, making up the "main memory"
    - Also called "system memory"
- Hopefully your program 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

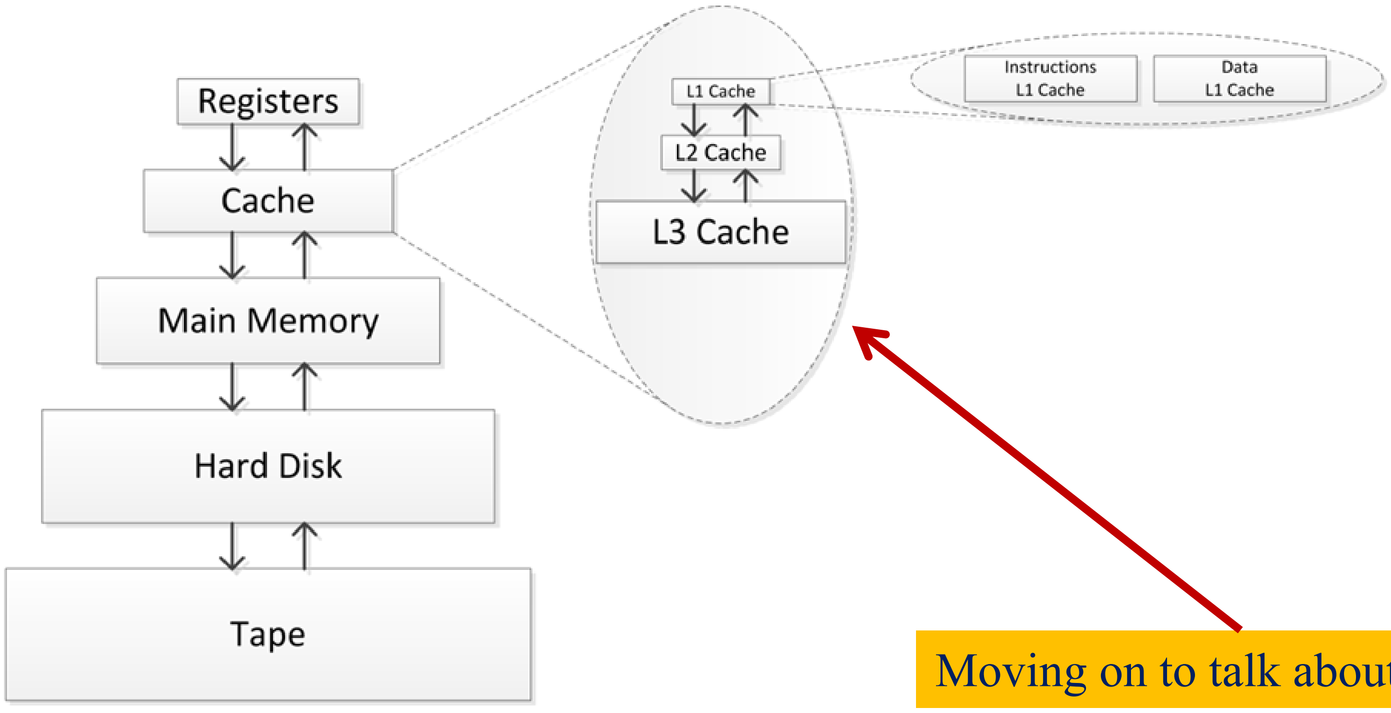
Example: Intel Chip Architecture

- 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:



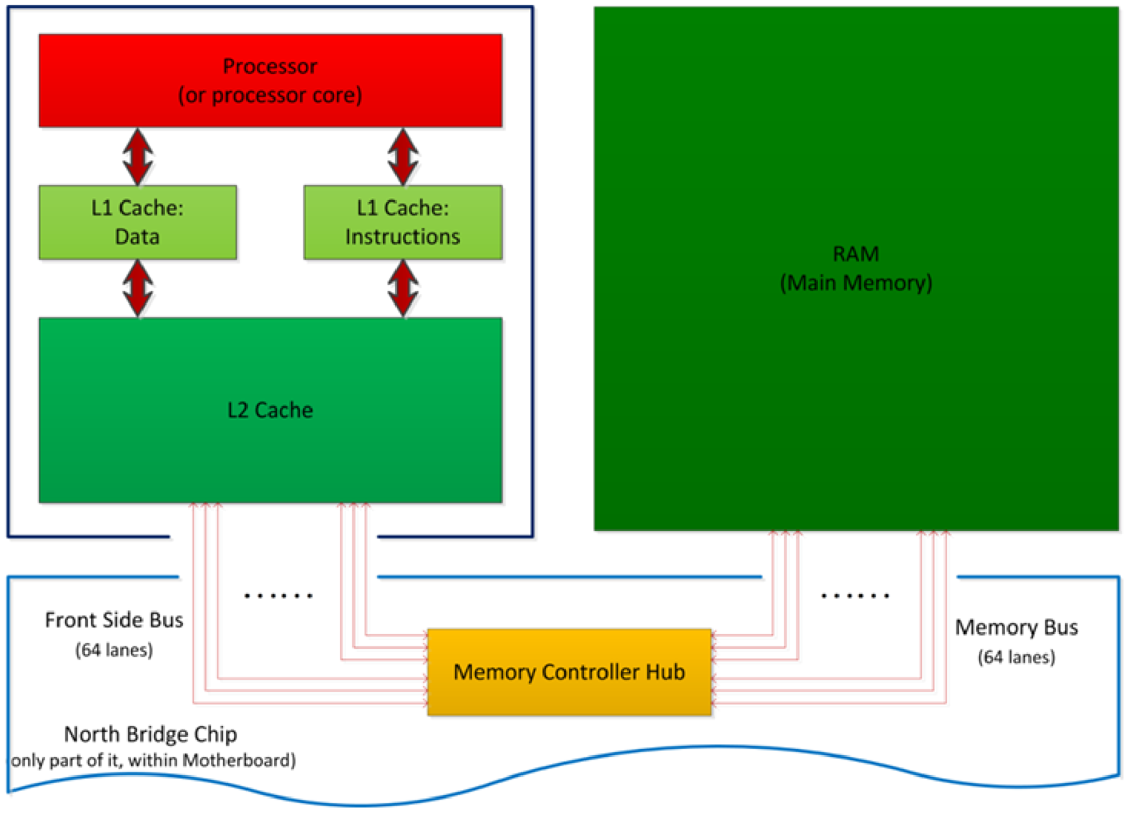
Moving on to talk about caches

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

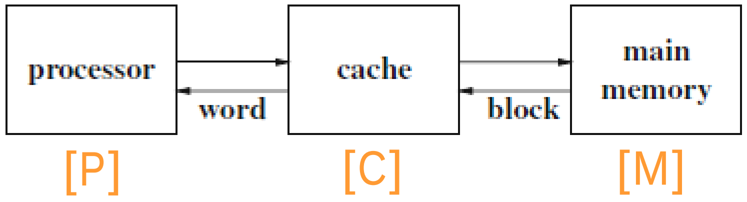
Split vs. Unified Caches

- 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 as far as data/instructions are concerned



Cache Transactions: Words and Lines

- 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 for P ↔ C transactions and for C ↔ M transactions
  - P ↔ C: moved one word at a time
  - C ↔ 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 $\leftrightarrow$ 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:
  - <u>Cache read miss from an instruction cache</u>: 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

  - <u>A cache read miss from a data cache</u>: 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.

  - <u>A cache write miss to a data cache</u>: 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.

[It makes sense to ask this]

Question:

- Can you control what’s in the cache and anticipate future memory requests?
  - Not directly…
    - Most systems have a hardware implemented cache controller with mind of its own
  - Indirectly though, it’s possible
    - Design your software implementation for 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

The Concept of Virtual Memory

Why Talk About Virtual Memory?

- It will help us understand a little bit better why accessing memory is expensive

[recall that memory access pattern is what dictates mostly speed of execution]

A Bunch of Curious Things

- Question 1: On a 32 bit machine, how come you can have 512MB of main memory yet allocate an array of 1 GB?
- Question 2: How can you compile a program on a Windows workstation with 2 GB of memory and run it later on a different laptop with 512 MB of memory?
- Question 3: How can several processes run seemingly at the same time on a processor that only has one core?
    - Don’t these processes step on each other’s memory?

Preamble, Virtual Memory

- The three questions raised on previous slide answered by the interplay between the compiler, the operating system (OS), and the execution model embraced by the processor

- When you compile a program there is no way to know where in the physical memory the code will get its data allocated
  - There are other “tenants” that inhabit the physical memory, and they are there before you get there
  - In other words, one can’t assume that absolute addresses will be ok

- The solution is for the code to be compiled and assumed to lead to a process that executes in a virtual world in which it has access to 4 GB of memory (on 32 bit systems).
  - The “virtual world” is called the virtual memory space

Virtual vs. Physical Memory

- Virtual memory: a nice and immaculate space of 2^{32} addresses (on 32 bit architectures) in which a process sees its data being placed, the instructions stored, etc.

- Physical memory: a busy place that hosts at the same moment in time data and instructions associated with [most likely] multiple applications running on the system

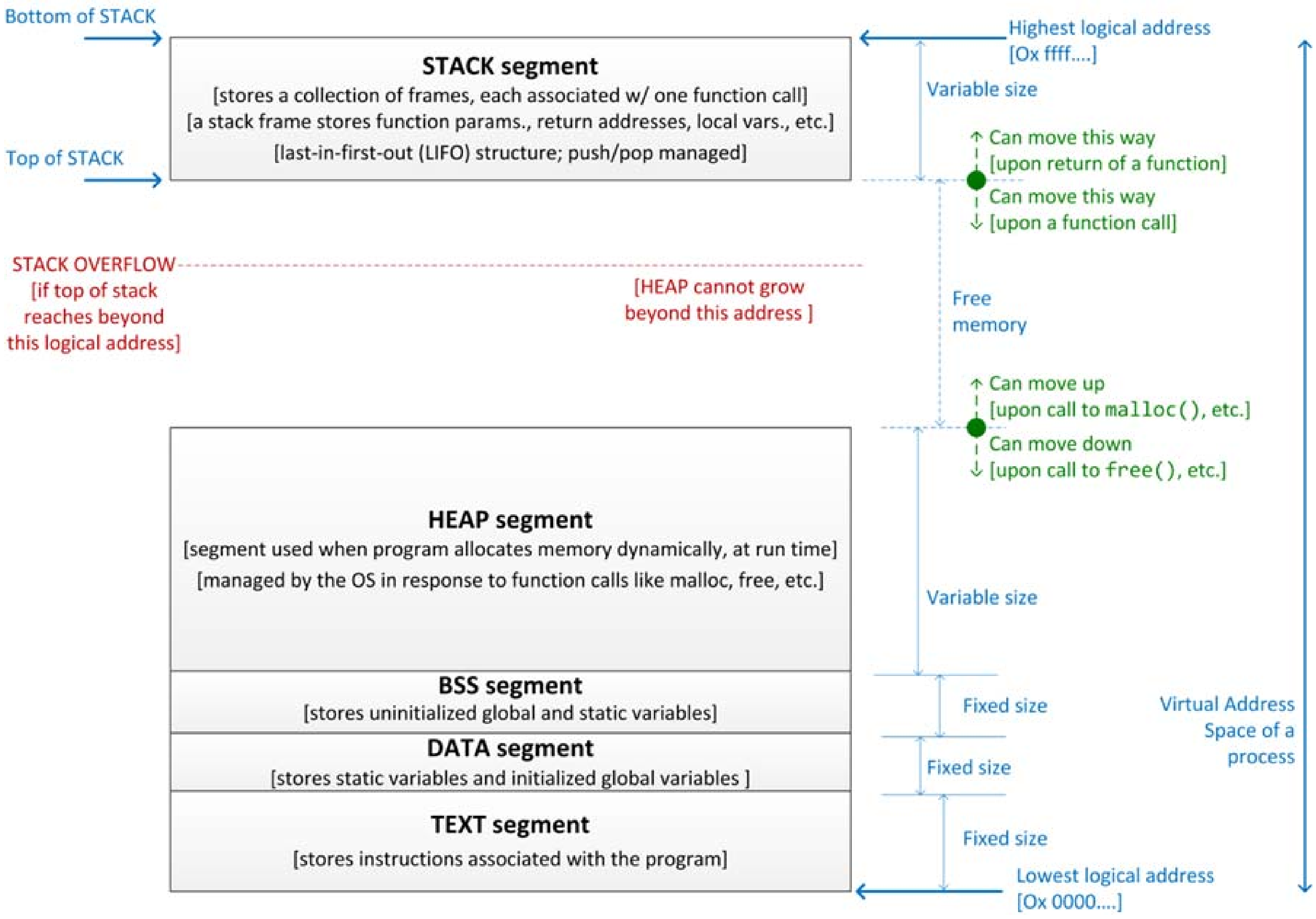
- The virtual memory is connected back to, or mapped back into, the physical memory through a Page Table which enables address translation
  - The Page Table's purpose in life: facilitates the aliasing

One Word:

Aliasing

Anatomy of the 32 bit Virtual Memory

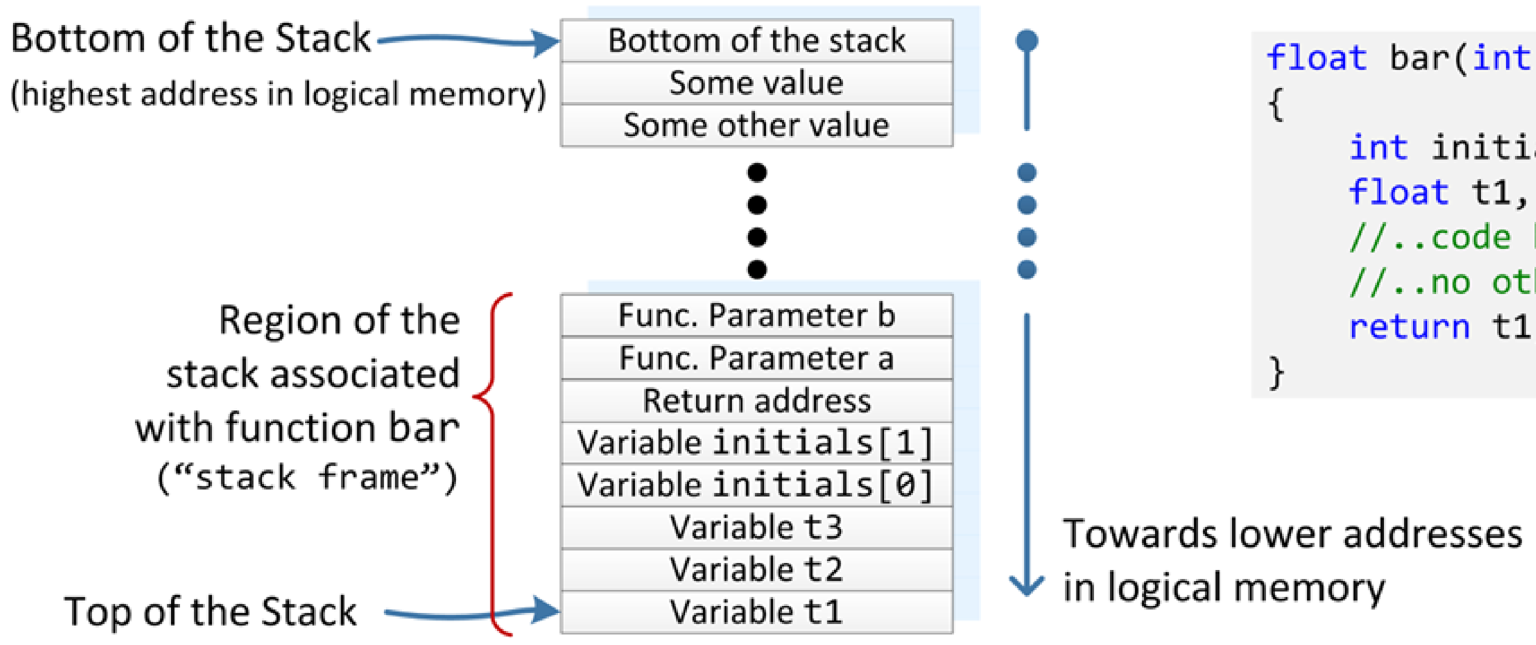
[first order approximation]



Bottom of STACK
Top of STACK
STACK segment
[stores a collection of frames, each associated w/ one function call]
[a stack frame stores function params., return addresses, local vars., etc.]
[last-in-first-out (LIFO) structure; push/pop managed]
Highest logical address
[Ox ffff....]
Variable size
Can move this way
[upon return of a function]
Can move this way
[upon a function call]
STACK OVERFLOW
[if top of stack reaches beyond this logical address]
[HEAP cannot grow beyond this address]
Free memory
Can move up
[upon call to malloc(), etc.]
Can move down
[upon call to free(), etc.]
HEAP segment
[segment used when program allocates memory dynamically, at run time]
[managed by the OS in response to function calls like malloc, free, etc.]
Variable size
BSS segment
[stores uninitialized global and static variables]
Fixed size
DATA segment
[stores static variables and initialized global variables]
Fixed size
Virtual Address Space of a process
TEXT segment
[stores instructions associated with the program]
Fixed size
Lowest logical address
[Ox 0000....]

The Anatomy of the Stack
[going into the weeds a bit]

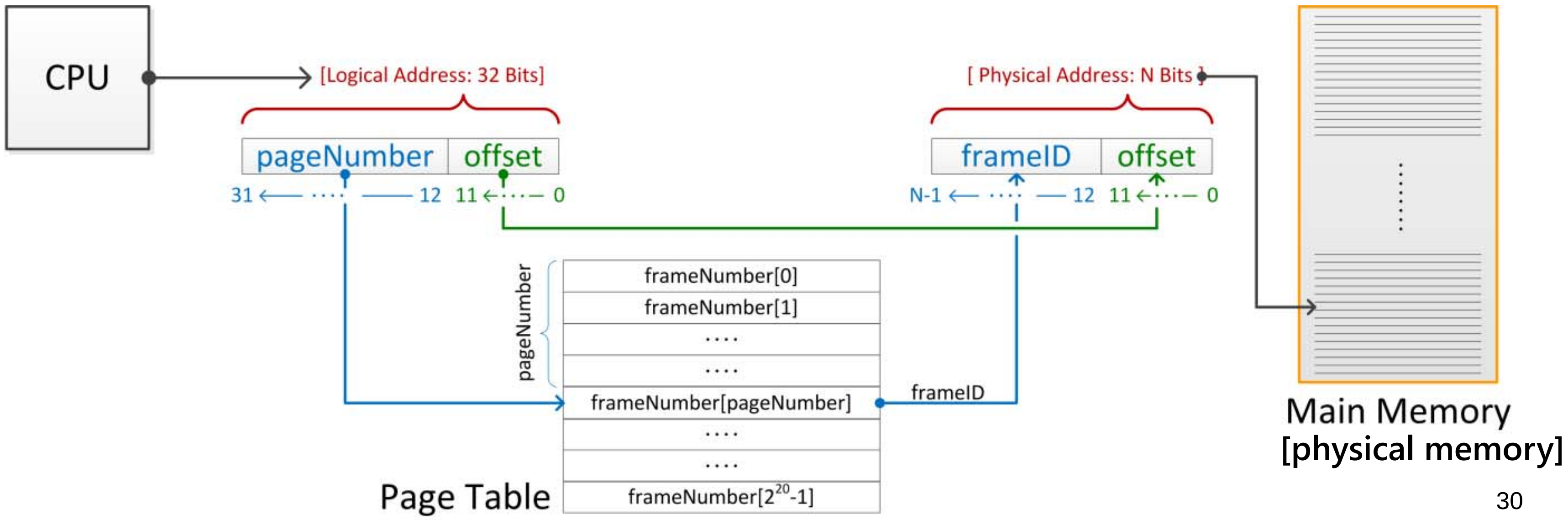
- Function `bar` and associated stack frame



```
float bar(int a, float b)
{
    int initials[2];
    float t1, t2, t3;
    //..code here..
    //..no other variables..
    return t1;
}
```

Anatomy of the 32 bit Virtual Memory
[first order approximation]

- A virtual address has two parts: the page number, and the offset
- The rules:
  - To each virtual memory page corresponds a [physical memory] frame ID
    - The former (stored as the most significant 20 bits) is an alias for the latter
  - To each offset in a page corresponds an offset in the frame.
    - These offsets are identical (least significant 12 bits)



Anatomy of the 32 bit Virtual Memory

[first order approximation]

- A page of virtual memory corresponds to a frame of physical memory

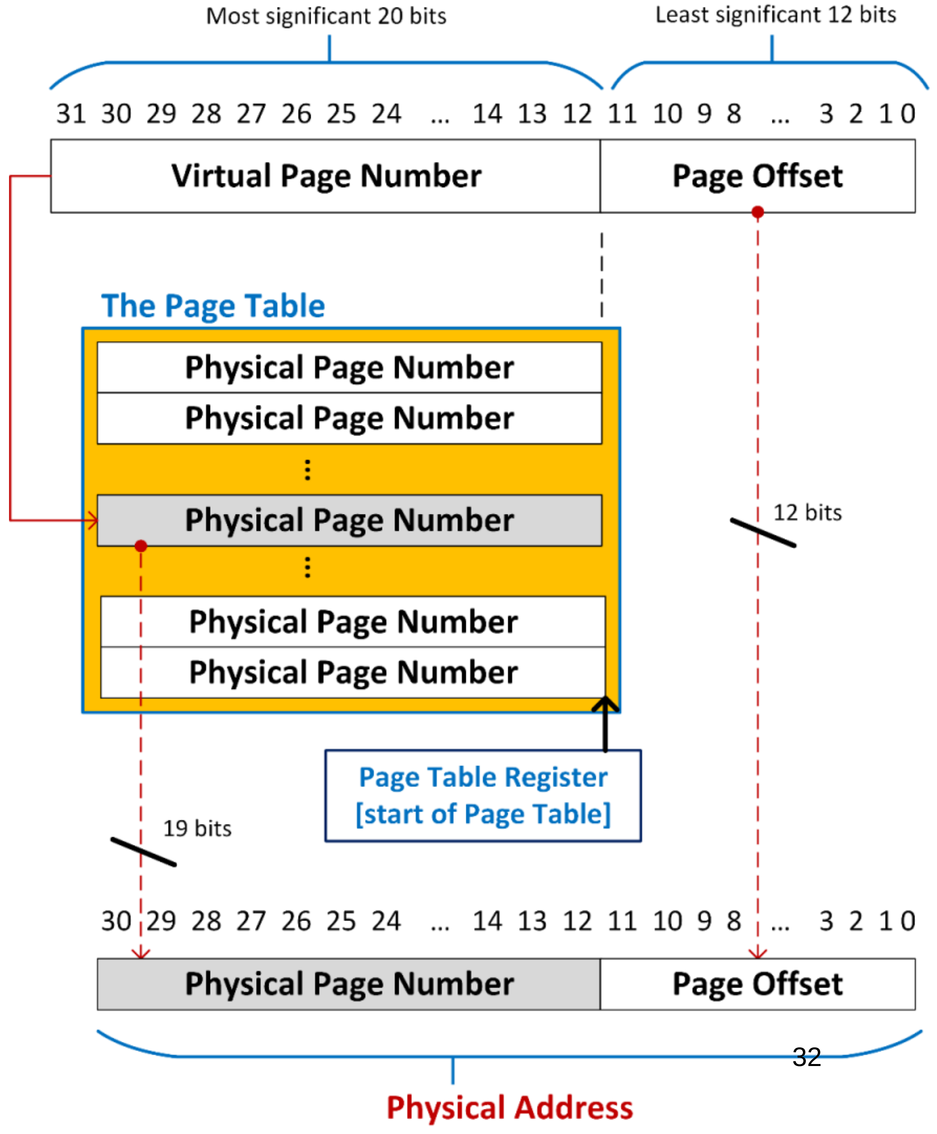
- The size of a page (or frame, for that matter) is typically 4096 bytes

- 2^{12}=4096: 12 address bits are sufficient to relatively position each byte in a page

Example

[The Translation Process]

- Imagine that your physical memory is 2 GB
- The physical address has 31 bits: 2^{31}=2GB
- Then the page table converts bits 12 through 31 of the virtual address into bits 12 through 30 of the physical address



Short Digression 1: The Unit of Address Resolution

- How many bits are available for data storage at each address?
- Example:
  - We have 2^{32} addresses that we can access
  - If each address points to a location that stores 8 bits (one byte) then we have 4 GB of addressable memory
  - However, if each address refers to a location that stores 2 bytes, we have 8 GB of addressable memory
- Intel and AMD CPUs: the unit of address resolution is 1 byte (8 bits)

- Consequence: the Intel 32 bit processors “see” a virtual memory space that can be 4 GB big and can reference each byte therein

Short Digression 2:
32 to 64 Bit Migration – The Need

- If the architecture and OS use 32 bits to represent addresses, it means that 2^{32} addresses can be referenced
- If unit of address resolution is 1 byte, that means that the size of the virtual memory space can be 4 GB
- This is hardly enough today when programs are very large and the amounts of data they manipulate can be staggering
- This motivated the push towards having addresses represented using 64 bits: the memory space balloons to 2^{64} bytes, that is 16 times 1,152,921,504,606,846,976 bytes

Short Digression 3:
32 to 64 Bit Migration - Implications

- Note that a 64 bit architecture typically calls for two things:
- From a hardware perspective, the size of the registers, integer size, and word size is 64 bits
  - The microarchitecture changes
- From a software perspective, the program "operates" in a huge virtual memory space
  - The operating system (OS) is the party managing the execution of a program in the 64 bit universe
  - Can allocate very large arrays
  - In reality, the virtual memory addresses are represented using only about 40 to 45 bits

Comments on the Page Table Preamble to TLB.

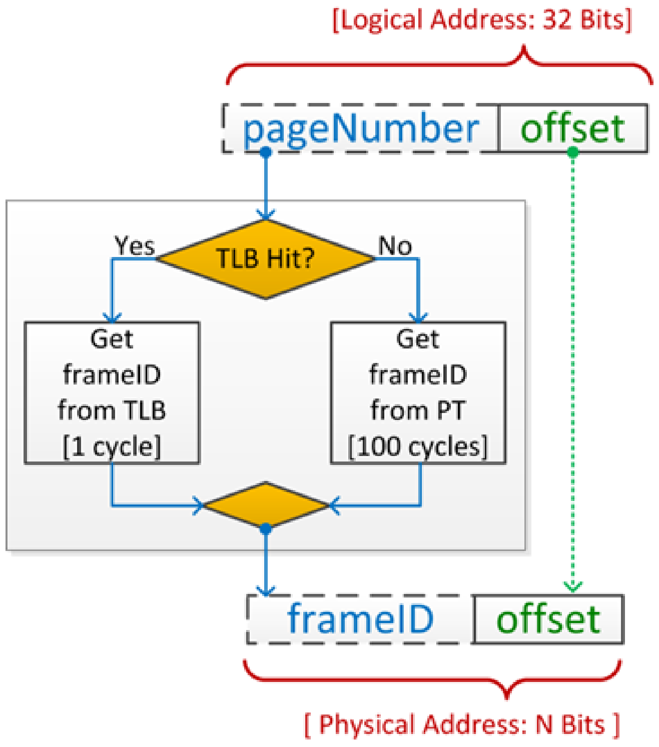
- The page table is the key ingredient that allows the translation of virtual addresses into physical addresses
- Every single process executing on a processor and managed by the OS has its own page table
- Page table is stored in main memory
  - For a 32 bit operating system size of a page table can be up to 4 MB in size
    - 4 MB: 2^20 addresses, each requiring 4 bytes to encode

The TLB
[Translation Lookaside Buffer]

- If Page Table stored in main memory it means that each address translation would require a trip to main memory
  - This would be very costly
- There is a “cache” for this translation process: TLB
  - Translation lookaside buffer: holds the translation of a small collection of virtual page numbers into frame IDs
- Best case scenario: the TLB leads to a hit and allows for quick translation
- Bad scenario: the TLB doesn’t have the required information cached and a trip to main memory is in order

Illustration: The Role of the TLB

- A TLB is just like a cache with the caveat that it is exclusively engaged in address translation
- A TLB miss leads to substantial overhead in the translation of a virtual memory address
- Yet there is something worse: the requested frame is not in the main memory and a trip to secondary memory is in order
    - Called "page fault"

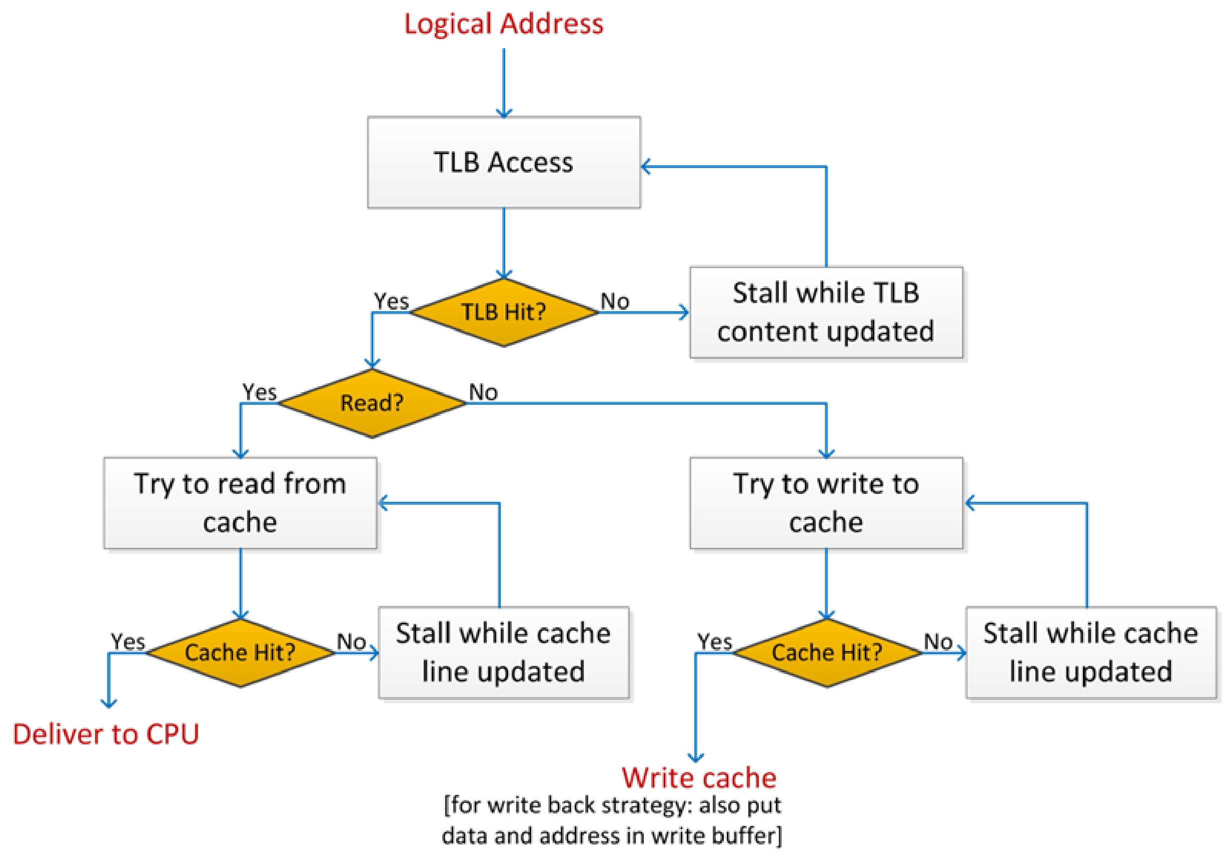

[Logical Address: 32 Bits]
pageNumber
offset
Yes
TLB Hit?
No
Get frameID from TLB [1 cycle]
Get frameID from PT [100 cycles]
frameID
offset
[Physical Address: N Bits]

Page Faults

- Memory frame is not in system memory
- As a result, the OS engages in memory paging
- Memory paging is a lengthy process:
  - Figure out where the frame is in secondary memory
  - Get an empty frame in the system memory
    - If none available, you need to evict
      - A process can get one of its frames evicted, or the OS might steal frame from different process
  - Load data into the available frame
  - Update the page table to connect the virtual page to the new memory frame
  - Service the memory request
- Memory thrashing
  - That situation in which your program keeps running over and over again into page faults

Memory Access: The Big Picture, Includes Cache

- A simplified version of how a memory request is serviced presented below


Logical Address
TLB Access
TLB Hit?
Yes
No
Stall while TLB content updated
Read?
Yes
No
Try to read from cache
Try to write to cache
Cache Hit?
Yes
No
Stall while cache line updated
Cache Hit?
Yes
No
Stall while cache line updated
Deliver to CPU
Write cache
[for write back strategy: also put data and address in write buffer]

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
- Performance metrics for program execution
- SRAM vs DRAM
- Caches and the hierarchical setup of the memory ecosystem
- Virtual Memory
- Virtual Memory address translation and the role of the TLB

Parallel Computing:
Why? & Why Now?

Acknowledgements

- Material presented today includes content due to
  - Hennessy and Patterson (Computer Architecture, 5th edition)
  - John Owens, UC-Davis
  - Darío Suárez, Universidad de Zaragoza
  - John Cavazos, University of Delaware
  - Others, as indicated on various slides
  - I apologize if I included a slide and didn’t give credit where was due