**Course Objectives: Where are we….**

- Bits&bytes: Logic devices
  - HW building blocks
- Processor: ISA, datapath
  - Using building blocks to assemble a processor (LC3)
- Programming the processor: Assembly
- Translating high level programs to Proc
  - Implementing C on LC3
- Working with C

**Computer Architecture: Memory Organization & Performance**

**Bits&Bytes to High level Programs**

- User application written in high level language
- Program runs on a processor

- How are high level programs implemented on processor?
  - Run-time stack, allocation of variables, translation of high level code to machine code
  - Map high level data structures to low level data structures
    - Struct to linear mapping in memory
- What else does software developer want after program is implemented correctly?

- PERFORMANCE!

**Next…the end is near !**

- Performance of programs
  - What to measure
  - Model?
  - Technology trends
    - "real" processors…how to improve performance
      - Pipelining, ILP, Multi-core
- Memory organization basics…TODAY
  - Memory hierarchy: cache, main memory, etc.
  - How to rewrite your program to make it run faster…code optimization…Next Class
    - Project 6 – rewriting code to improve performance
Performance of Programs
• “Complexity” of algorithms
  • How good/efficient is your algorithm
    - Measure using Big-Oh notation: O(N log N)
  • Next question: How well is the code executing on the machine?
    - Actual time to run the program
    - Effect of H/W design issues on SW performance

How to Model Performance
• The asymptotic complexity – “big O”
  - Time = O (f(n)) : function of the size of the input
  - Sorting: O(n log n)
  - This measures efficiency of your algorithm
    - i.e., how ‘good’ is solution technique
    - Is this enough when we talk of actual time measured on the processor?
• There’s more to performance than asymptotic complexity
  - Must optimize at multiple levels: algorithm, data representations, procedures, and loops
  - Must understand system to optimize performance
    - How programs are compiled and executed, data storage, data structures, I/O management

Performance Trends: Summary
• Workstation performance (measured in Spec Marks) improves roughly 50% per year (2X every 18 months)
  - Performance will include not just processor, but memory and disk I/O
• Improvement in cost performance estimated at 70% per year

Performance – what to measure?
Which of these airplanes has the better performance?

<table>
<thead>
<tr>
<th>Plane</th>
<th>DC to Paris</th>
<th>Speed</th>
<th>Passengers</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boeing 747</td>
<td>6.5 hours</td>
<td>610 mph</td>
<td>470</td>
<td></td>
</tr>
<tr>
<td>BAD/Sud Concorde</td>
<td>3 hours</td>
<td>1350 mph</td>
<td>132</td>
<td></td>
</tr>
</tbody>
</table>
The Bottom Line: Performance metric depends on application

- Time to run the task (Execution Time/Response Time/Latency)
  - Time to travel from DC to Paris
- Tasks per unit time (Throughput/Bandwidth)
  - Passenger miles per hour; how many passengers transported per unit time

<table>
<thead>
<tr>
<th>Plane</th>
<th>DC to Paris</th>
<th>Speed</th>
<th>Passengers</th>
<th>Throughput (pmph)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boeing 747</td>
<td>6.5 hours</td>
<td>610 mph</td>
<td>470</td>
<td>286,700</td>
</tr>
<tr>
<td>BAD/Sud Concorde</td>
<td>3 hours</td>
<td>1350 mph</td>
<td>132</td>
<td>178,200</td>
</tr>
</tbody>
</table>

Computer Performance: TIME, TIME, TIME
- Response Time (latency)
  - How long does it take for my job to run?
  - How long does it take to execute a job?
  - How long must I wait for the database query?
- Throughput
  - How many jobs can the machine run at once?
  - What is the average execution rate?
  - How much work is getting done?

Metric chosen usually depends on user community: sys admin vs single user
- Ques.: If we upgrade a machine with a new processor what do we increase?
- Ques: If we add a new machine to the server farm what do we increase?

Execution Time
- Elapsed Time
  - counts everything (disk and memory accesses, I/O, etc.)
  - a useful number, but often not good for comparison purposes
- CPU time
  - doesn’t count I/O or time spent running other programs
  - can be broken up into system time, and user time

Our focus in this course: user CPU time
- time spent executing the lines of code that are “in” our program

Processor time: how to measure?
- Number of clock cycles it takes to complete the execution of your program
  - Time = Number of clock cycles * CPU clock cycle
  - Example: 1000 cycles and 1 GHz clock = 1000 * 1 nanosecond
- What is your program
  - A number of instructions
    - Different types: load, store, ALU, branch
  - Stored in memory
  - Executed on the CPU
Model for CPU Performance

CPU time = Seconds = Instructions x Cycles x Seconds

Program Program Instruction Cycle

\[ CPU = IC \times CPI \times Clk \]

simple but very effective and standard way to measure

CPI

- Cycles per instruction
  - Different instructions may take different time
  - Example in LC 3?
  - We observed that not every instruction needs to go through all the instruction execution steps
    - Eg: no need to calculate effective address, fetch from memory or registers
    - In "real" processors: different times associated with different operations
      - Especially true of memory operations

Average CPI

- Application has an "instruction mix"
  - Profile of application instruction types
  - ALU, Load/Store (memory), Branch, Jumps, etc.
    - \( x_1, x_2, x_3 \ldots \) as percentage (\( x_1=0.4 \))
  - Processor has CPI for each type of instruction
    - Part of ISA of a processor...specifications doc
    - Example: ALU=1.0, Load/Store=2.0, etc.
    - \( t_1, t_2, t_3 \ldots \)
- What is effective CPI?
- Weighted average
  - CPI = \( x_1 \times t_1 + x_2 \times t_2 + \ldots \)

CPI: Cycles per instruction

- Depends on the instruction
  \[ CPI_i = \text{Execution time of instruction } i / \text{Cycle time} \]
- Average cycles per instruction
  \[ CPI = \sum CPI_i \times F_i \quad \text{where } F_i = \frac{IC_i}{IC_{tot}} \]
- Example:

<table>
<thead>
<tr>
<th>Op</th>
<th>Cycles</th>
<th>CPI</th>
<th>%time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>1</td>
<td>9.5</td>
<td>33%</td>
</tr>
<tr>
<td>Load</td>
<td>2</td>
<td>9.4</td>
<td>27%</td>
</tr>
<tr>
<td>Store</td>
<td>2</td>
<td>9.2</td>
<td>13%</td>
</tr>
<tr>
<td>Branch</td>
<td>2</td>
<td>9.4</td>
<td>27%</td>
</tr>
</tbody>
</table>

\[ CPI_{tot} = 1.5 \]
Next : Memory

• Key component in von Neumann computer?
• Memory

• How are “real” memory systems organized?
  • How do they affect performance?

Technology Trends: The CPU-Memory Gap

• The increasing gap between DRAM, disk, and CPU speeds.

Memory Technology

• Random access memory
  • Can read from any location by supplying address of data

• Memory Comes in Many Flavors
  • Main RAM memory Key features
    - RAM is packaged as a chip.
    - Basic storage unit is a cell (one bit per cell).
    - Multiple RAM chips form a memory.
  • SRAM (Static Random Access Memory) or DRAM (Dynamic Random Access Memory)

• ROM, EPROM, EEPROM, Flash, etc. – Non-Volatile
  • Read only memories – store OS

• “Secondary memory” Disks, Tapes, Flash etc.

• Difference in speed, price and “size”
  • Fast is small and/or expensive
  • Large is slow and/or cheap
**CPU & Memory Interaction...speed mismatch**

- Many types of memory with different speeds
- Processor speed and memory speed mismatched
  - Data transferred between memory and processor
    - Instructions or data
- What does processor do while waiting for data to be transferred?
  - Idle – processor is stalled leading to slowdown in speed and lower performance
- Why can’t we have memory as fast as processor
  - Technology, cost, size
- What is the solution then?

**Memory Hierarchies**

- Organize memory as a memory hierarchy.
- Key Principles
  - Locality – most programs do not access code or data uniformly
  - Smaller hardware is faster
- Goal
  - Design a memory hierarchy “with cost almost as low as the cheapest level of the hierarchy and speed almost as fast as the fastest level”
    - This implies that we be clever about keeping more likely used data as “close” to the CPU as possible
- Levels provide subsets
  - Anything (data) found in a particular level is also found in the next level below.
  - Each level maps from a slower, larger memory to a smaller but faster memory

**“Memory” organization**

Items on a desktop (register) or in a drawer (cache) are more readily accessible than those in a file cabinet (main memory) or in a closet in another room.

Keep more frequently accessed items on desktop, next frequent in drawer, etc. and things you need a lot less often in the closet in another room!

**Why does this work? Key Concept: Locality**

- Principle of Locality:
  - Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
  - Temporal locality: Recently referenced items are likely to be referenced in the near future.
    - Nearby in time
  - Spatial locality: Items with nearby addresses tend to be referenced close together in time.
    - Nearby in space
Locality: Example

```c
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

• Data
  - Reference array elements in succession: **Spatial locality**
  - Reference `sum` each iteration: **Temporal locality**

• Instructions
  - Reference instructions in sequence: **Spatial locality**
  - Cycle through loop repeatedly: **Temporal locality**

Design of Memory Hierarchy

• Brief Overview of Memory Design
  - Focus on Cache design
    ➢ How does Cache memory work?
    ➢ How are addresses mapped to Cache
    ➢ How to rewrite code to get better cache performance? – code optimization
  - How do disks work?
  - Later in Operating Systems: Virtual memory – what is it?

Today's Memory Hierarchy

- Smaller, faster, and costlier (per byte) storage devices
- Larger, slower, and cheaper (per byte) storage devices

- L0: registers
- L1: on-chip L1 cache (SRAM)
- L2: off-chip L2 cache (SRAM)
- L3: main memory (DRAM)
- L4: local secondary storage (local disks, SSD)
- L5: remote secondary storage (distributed file systems, Web servers)

CPU registers hold words retrieved from L1 cache.
L1 cache holds cache lines retrieved from the L2 cache memory.
L2 cache holds cache lines retrieved from main memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote network servers.

Link to Performance?

• Simplified model:
  ➢ Processor is (1) in execution or (2) waits for memory
    ➢ "effective" (Real) CPI increases
  ➢ execution time = (execution cycles + memory stall cycles) * cycle time
Memory Access time and Performance?

\[
\text{CPU time} = \text{Seconds} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Program}} \times \frac{\text{Seconds}}{\text{Instruction}} \times \text{Cycle}
\]

\[\text{CPU} = IC \times CPI \times Clk\]

• How does memory access time fit into CPU time equation?

• More stall cycles = increase in CPI

Performance: Simplified Goal?

• Improve performance = decrease stall cycles
  • Decrease time to access memory
  • How?
    • Organize memory as a hierarchy!
    • Most of the time you should access data in the fastest memory

Recall – Programmers view of Memory Unit

• An ordered sequence of storage cells, each capable of holding a piece of data.
• Address space
  • Size of memory: N bit address space = \(2^N\) memory locations
• Addressability
  • Size of each memory location – k bits
• Total memory size = \(k \times 2^N\) bits
• Assumption thus far: Processor/CPU gets data or instruction from some memory address (Inst fetch or Load/Store instruction)
  • Time depends on how is memory actually organized?
  • Can everything we need fit into a memory that is close to the CPU?

Typical Bus Structure Connecting CPU and Memory

• A bus is a collection of parallel wires that carry address, data, and control signals.
• Buses are typically shared by multiple devices.
Memory Read Transaction (1)
- CPU places address A on the memory bus.

Memory Read Transaction (2)
- Main memory reads A from the memory bus, retrieves word x, and places it on the bus.

Memory Read Transaction (3)
- CPU reads word x from the bus and copies it into register %eax.

Reality…Memory access
I/O Bus

Main memory

System bus

Memory bus

Bus interface

I/O bridge

ALU

Register file

CPU chip

USB controller

Graphics adapter

Disk controller

Mouse/keyboard

Monitor

Disk

Expansion slots for other devices such as network adapters.

What happens if not found in main memory

Read from Disk

CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.

Data not found in memory

Reading a Disk Sector (2)

Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.

Reading a Disk Sector (3)

When the DMA transfer completes, the disk controller notifies the CPU with an interrupt (i.e., asserts a special "interrupt" pin on the CPU).
Why Cache?
• Gap between main memory speed and processor speed
  • Reading data/instr from memory will take more than 1 processor cycle
    ➢ Increases time to execute program
• What?: Place a small but fast memory close to the processor
• How?: Why does this work
  • Principle of Locality

Simple Model of Memory Hierarchy...
• Sequence of addresses
  • How many?
• CPU generates request for memory location – i.e., an address
  • How long does it take to get this data?
    ➢ Depends where it is in the Memory hierarchy
• Simplified Model for memory hierarchy:
  • Small amount of Fast On-chip Cache memory
  • Larger amount of off-chip Main memory
  • Huge Disk

How does Cache memory work?
• Address space = $2^N$ words each of some size K bits
  • N bit address
• Memory addresses go from 0 to $2^N-1$
  • These are the addresses that the processor requests in the Load or Store instructions, or when fetching a new instruction (value in PC)
• Some of these memory locations are placed in the cache
  • If you see it in the cache then don’t need to go all the way to memory to read them
    ➢ Faster time to read/write inst/data!
Memory Access times

• memory access time
  • On-chip Cache takes 1 processor cycle
  • Main memory takes a number (10-50) processor cycles
  • Disk takes a huge amount

• Simple model we will use:
  • Memory = Cache + Main memory
  • Small size Cache = not everything fits in it

• Cache organization:
  • Cache consists of a set of blocks each of some number of bytes
  • Only a block can be fetched into and out of cache
  • Eg: if block is 16 bytes, then load 16 bytes into cache
    ➢ Cannot load a single byte

Terminology Summary

• Hit: data appears in block in upper level (i.e. block X in cache)
  • Hit Rate: fraction of memory access found in upper level
  • Hit Time: time to access upper level which consists of
    ➢ RAM access time + Time to determine hit/miss

• Miss: data needs to be retrieved from a block in the lower level (i.e. block Y in memory)
  • Miss Rate = 1 - (Hit Rate)
  • Miss Penalty: Extra time to replace a block in the upper level +
    ➢ Time to deliver the block the processor

• Hit Time << Miss Penalty (500 instructions on Alpha 21264)

Memory Access times using Simplified Model

• If data is found in Cache then time = 1
  • Called a cache hit

• Else time is Main memory access time
  • Cache miss, means read from next level

• Note: need a ‘control unit’ to determine if location is in cache or not
  • Cache controller

• Why does concept of caching work?
  • Principle of Locality
    ➢ Programs access data nearby, or data/instructions that were used recently

Memory Hierarchy—Performance

• Placing the fastest memory near the CPU can result in increases in performance

• Consider the number of cycles the CPU is stalled waiting for a memory access: memory stall cycles

• CPU execution time =
  (CPU clk cycles + Memory stall cycles) * clk cycle time.

• Memory stall cycles = number of misses * miss penalty

• Fewer misses in cache = better performance!
**Average Memory Access Time**

\[
AMAT = \text{HitTime} + (1 - h) \times \text{MissPenalty}
\]

- **Hit time**: basic time of every access.
  - Always look to check if data is in cache
- **Hit rate (h)**: fraction of access that hit
  - usually substituted by \( \text{miss rate } m = (1-h) \)
- **Miss penalty**: extra time to fetch a block from lower level, including time to replace in CPU
  - Access time to read/write to memory module

Example 1

- System = Processor, Cache, Main Memory
  - Cache time = 1 processor cycle
  - Memory access time = 50 processor cycles
- Suppose out of 1000 memory accesses (due to Load/Store and Inst fetch)
  - 40 misses in the cache
    - 960 hit in the Cache
  - Miss ratio = 40/1000 = 4%
- Average memory access time with and without cache?

Example.

- Average memory access time with and without cache?
- \( \text{AMAT-cache} = 1 + \text{miss ratio} \times \text{miss penalty} \)
  - \( 1+ (0.04)\times 50 = 3 \)
  - \( \text{AMAT-without-cache} = 50 \)
- What happens if miss ratio increases?

**Cache Memory Hardware Design**

- Main memory has \( 2^n \) locations
- Cache has \( 2^k \) locations
  - Smaller than main memory
  - How to "organize" these cache locations?
- Processor generates N bit address
- The **Cache Controller** hardware must look at this N bit address and decide (a) if it is in cache and (b) where to place it in cache?
**Cache Memory Design: Definitions**

- Cache has a total size – number of bytes in cache
- Transfers take place in blocks
  - A whole block is transferred between memory and cache
- Locating a block requires two attributes:
  - Size of block
  - Organization of blocks within the cache
- **Block size** (also referred to as line size)
  - Smallest usable block size is the natural word size of the processor
    - Else would require splitting an access across blocks and slows down translation

**Memory viewed as blocks**

- If cache block size = K bytes, then memory can be viewed as contiguous set of blocks each of size K

**Cache/Main Memory Structure**

- **Fully Associative**
- **Direct Mapped**
- **Set Associative**

**Where can a block be placed in a cache?**

- **Fully Associative**
  - Block 12 can go anywhere

- **Direct Mapped**
  - Block 12 can go only into Block 4 (12 mod 8)

- **Set Associative**
  - Block 12 can go anywhere in set 0 (12 mod 4)

**What if next block needed is block 4?**
Where can a block be placed in a cache?-Cache Organization
• If each block has only one place it can appear in the cache, it is said to be “direct mapped”
  • mapping is usually (Block address) MOD (Number of blocks in the cache)
• If a block can be placed anywhere in the cache, it is said to be fully associative
• If a block can be placed in a restrictive set of places in the cache, the cache is set associative.
  • A set is a group of blocks in the cache. A block is first mapped onto a set, and then the block can be placed anywhere within that set.
    (Block address) MOD (Number of sets in the cache)
  • If there are $n$ blocks in a set, the cache is called $n$-way set associative

Cache Design: How is data found in Cache?
• Note: Time to find data is the hit time: affects performance
• Processor generates address request – some N bit address
• Two issues:
  • How do we know if a data item is in the cache?
  • If it is, how do we find it? .. In which cache block do we look

Addressing.
• Memory address of N bits
  • This is what is requested by processor
• We need mapping from Addresses to Cache
• How to break up these N bits into fields so that the cache controller can easily determine if the address requested is already stored in cache?

Addressing.
• Focus on Direct mapped:
  • How to implement N Mod.P where P=2$^x$
  • How to determine which block to map the address -- index
  • How to determine which of the words is the one in memory -- tag
  • Since we are working with blocks, how to find the specific word within the block -- offset
  • N bit address is broken into these 3 fields!
Example

<table>
<thead>
<tr>
<th>Memory block 0</th>
<th>Memory block 1</th>
<th>Memory block 2</th>
<th>Memory block 3</th>
<th>Memory block 4</th>
<th>Memory block 5</th>
<th>Memory block 6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cache

<table>
<thead>
<tr>
<th>Cache block 0</th>
<th>Cache block 1</th>
<th>Cache block 2</th>
<th>Cache block 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem Block 4</td>
<td>Mem Block 1</td>
<td>Mem Block 6</td>
<td>Mem Block 3</td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>Memory block 0</th>
<th>Memory block 1</th>
<th>Memory block 2</th>
<th>Memory block 3</th>
<th>Memory block 4</th>
<th>Memory block 5</th>
<th>Memory block 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,1,2,3)</td>
<td>(4,5,6,7)</td>
<td>(8,9,10,11)</td>
<td>(12,13,14,15)</td>
<td>(16,17,18,19)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cache

<table>
<thead>
<tr>
<th>Cache block 0</th>
<th>Cache block 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 0</td>
<td>Block 1</td>
</tr>
</tbody>
</table>

4 bytes in each block
Addresses:
(0,1,2,3)

Addressing -- Example

• 8 bit address
  • Byte addressability
• 4 byte cache blocks
  • Each cache block has 4 bytes
  • Main memory has $2^8 = 256$ bytes….or 64 blocks
• Total size of cache = 16 bytes
  • 4 blocks, each of 4 bytes
• Into which cache block do we place address 01101101
• How do we know which block from memory is in the cache block?
  • Is it 01101101 or 11001101

Addressing -- Example

• 8 bit address
  • Byte addressability & 256 bytes in memory
• 4 byte cache blocks
  • Each cache block has 4 bytes = need 2 bits to specify which byte within the block
  • 4 bytes in each block – memory is 256/4= 64 blocks
• Total size of cache = 16 bytes
  • 4 blocks in cache = apply Modulo 4 mapping
• Into which cache block do we place 01101101
  • Last two LSB are offset within a block
  • Next 2 LSB are modulo 4 and specify cache block
• How do we know which block from memory is in the cache block?
  • Is it 01101101 or 11001101
Addressing -- Example

- The three fields are:
  - **Tag**: the 4 most significant bits
    - Tells you which of the memory blocks are in the cache
  - **Offset** (block offset): the 2 least significant bits
    - Gives the address within the cache block
  - **Index**: the remaining 2 bits
    - Tells you into which cache block a memory address is placed

- In general: For \( N \) bit address, \( 2^K \) bytes in cache block, \( 2^j \) blocks of total cache size
  - **Tag** of \( (N-k-j) \) bits: the most significant bits
  - **Offset** of \( K \) bits: the least significant \( K \) bits
  - **Index** of \( j \) bits

- Memory block \( M \) gets placed in cache block \( (M \mod 2^j) \).
  - \( M \) is \( (N-k) \) most significant bits

So how complex is the hardware required?

- Given \( N \) bit address, the cache controller needs to determine if the requested word is in cache or not
  - If not, then send the address to the memory bus
- This process has to be built into hardware
  - Complexity of the circuit determines time taken to determine if word is in cache
  - Hit time is function of complexity of circuit
  - For direct mapped, can you think of a hardware design?

Finding cache block

- Given the \( i \) bits of address, how do you find the cache block?
  - **01101101**
    - Mod 4 (Mod 2\(^j\) where \( 2^j \) is number of cache blocks)
- Given \( j \) bits of index, one of these is selected...
  - **Decoder!**

Finding word within a block

- Following four addresses are all in one block
  - **01101100**
  - **01101101**
  - **01101110**
  - **01101111**
  - How do you select one of these four?
  - **Multiplexer!**
Matching the tag of a memory block
- Number of memory blocks could all be mapped to the same cache block
- Following two will get mapped to same block
  - 01101101
  - 01111101
- You are searching to see if 01101101 is in the cache.....
- How do you check if cache contains this block with tag 0110 ?
- Comparator

Direct Mapped Caches

Cache Design .... Lot more to it!
- Cache performance is related to cache design
  - Size of cache, size of each block, organization
- Two ways of improving performance:
  - decreasing the miss ratio
  - decreasing the miss penalty

Summary: Memory Access time optimization
- If each access to memory leads to a cache hit then time to fetch from memory is one cycle
  - Program performance is good!
- If each access to memory leads to a cache miss then time to fetch from memory is much larger than 1 cycle
  - Program performance is bad!
- Design Goal:
  How to arrange data/instructions so that we have as few cache misses as possible.
How about rewriting code to improve cache rates?
• This is part of what code optimization accomplishes…

Secondary Memory
• CPU fetches from memory, memory is a sequence of $2^N$ locations
• What happens if main memory (chip capacity) is less than $2^N$
• Data and programs may be stored in a non-volatile component
  • MS-Word executable is on disk
  • Your application data
• What happens if more than one process is running on the system
  • Multiprogramming
  • What is the address space available to each user?
• Need to use Disks!

The Complete memory hierarchy
• Processor has a set of registers
  • Processor instructions operate on contents in the registers
• Small, fast cache memory placed near the processor
• Main memory sitting outside the chip
  • If data is not in the cache then fetch from main memory
  • Takes longer to access main memory than cache
• Disk sitting outside the motherboard
  • If data/program not in main memory then fetch/load from disk
  • Takes much longer to access disk than main memory

Why does memory hierarchy work?
• Principle of locality!
  • Exploit at each level.
How do disks work?

Disk Geometry
- Disks consist of platters, each with two surfaces.
- Each surface consists of concentric rings called tracks.
- Each track consists of sectors separated by gaps.

Disk Operation (Single-Platter View)
- The disk surface spins at a fixed rotational rate.
- The read/write head is attached to the end of the arm and flies over the disk surface on a thin cushion of air.
- The arm can position the read/write head over any track.
- By moving radially, the arm can position the read/write head over any track.
**Disk Access Time**
- Average time to access some target sector approximated by:
  - \( T_{access} = T_{avg \ seek} + T_{avg \ rotation} + T_{avg \ transfer} \)
- **Seek time** \( T_{avg \ seek} \)
  - Time to position heads over cylinder containing target sector.
  - Typical \( T_{avg \ seek} = 9 \) ms
- **Rotational latency** \( T_{avg \ rotation} \)
  - Time waiting for first bit of target sector to pass under r/w head.
  - \( T_{avg \ rotation} = \frac{1}{2} \times \frac{1}{RPMs} \times 60 \) sec/1 min
- **Transfer time** \( T_{avg \ transfer} \)
  - Time to read the bits in the target sector.
  - \( T_{avg \ transfer} = \frac{1}{RPM} \times \frac{1}{(avg \ # \ sectors/track)} \times 60 \) secs/1 min.

**Accessing a Disk Page**
- Time to access (read/write) a disk block:
  - **seek time** (moving arms to position disk head on track)
  - **rotational delay** (waiting for block to rotate under head)
  - **transfer time** (actually moving data to/from disk surface)
- Seek time and rotational delay dominate.

**Key to lower I/O cost:** reduce seek/rotation delays!

**Hardware vs. software solutions?**

**Placement of Data on Disk**
- Placement of data on disk can affect performance of program
- Example: If you are reading an entire array, then:
  - place the data in consecutive disk blocks
  - seek time to get first disk block
  - Remaining blocks will incur no seek time
- Time = \( T_{seek} + N \left( T_{transfer} + T_{rot} \right) \)
- Naïve approach: \( N \left( T_{seek} + T_{transfer} + T_{rot} \right) \)
- Speedup: \( N T_{seek} \sim O(N) \)

**Logical Disk Blocks**
- Modern disks present a simpler abstract view of the complex sector geometry:
  - The set of available sectors is modeled as a sequence of \( b \)-sized logical blocks \( (0, 1, 2, ...) \)
  - Mapping between logical blocks and actual (physical) sectors
    - Maintained by hardware/firmware device called **disk controller**.
    - Converts requests for logical blocks into (surface,track,sector) triples.
  - Allows controller to set aside spare cylinders for each zone.
  - Accounts for the difference in “formatted capacity” and “maximum capacity.”
Reading a Disk Sector (1)

CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.

Reading a Disk Sector (2)

Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.

Reading a Disk Sector (3)

When the DMA transfer completes, the disk controller notifies the CPU with an interrupt (i.e., asserts a special "interrupt" pin on the CPU).
**Last step: Virtual Memory**

- N bit address space
- If word is not in main memory then it is on disk — need a controller to manage this (analogous to cache controller)
  - Virtual memory management system

• Note: transfers take place from disk in “blocks” of M bytes (sector) — page of data
• Memory consists of a number of pages
  - Determine if the page is in main memory or not
• N bit address broken into fields which determine page number, and whether page is in the memory or disk
• If page size is 1024 bits...how to organize the N bit address??

**Virtual Memory**

- Main memory can act as a cache for the secondary storage (disk)
- Advantages:
  - Ilusion of having more physical memory
  - Program relocation
  - Protection

![Virtual Memory Diagram](attachment:image.png)

**What is multiprogramming and Why?**

- Processor overlaps execution of two processes/programs
  - When one is waiting for I/O, the other is “swapped” in to the processor
  - Save the “state” of the process being swapped out
- Processes need to share memory
  - Each has its own address space
  - Leads to better throughput and utilization

**Memory Hierarchy: The Big Picture**

![Memory Hierarchy Diagram](attachment:image.png)

Data movement in a memory hierarchy.
Mapping Virtual Memory to Physical Memory

- Divide Memory into equal sized "chunks" (say, 4KB each)
- Any chunk of Virtual Memory assigned to any chunk of Physical Memory ("page")

Pages: virtual memory blocks

Page faults: the data is not in memory, retrieve it from disk
- huge miss penalty, thus pages should be fairly large (e.g., 4KB)
- reducing page faults is important (LRU is worth the price)
- can handle the faults in software instead of hardware
- using write-through is too expensive so we use writeback

More on Virtual memory
- This is part of the Operating system