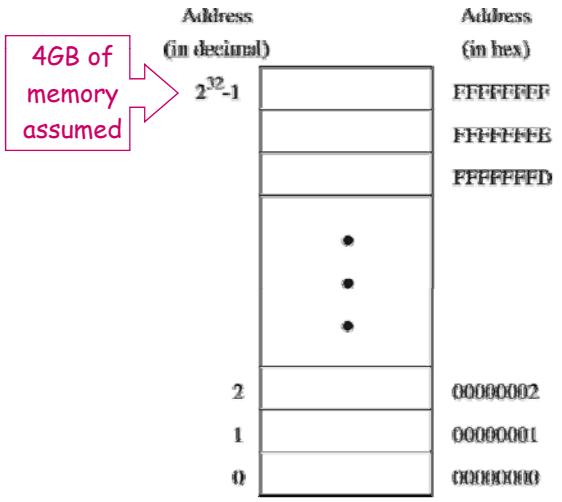
# Computer Organization/Design Overview
## (programmers' view of computer's memory)

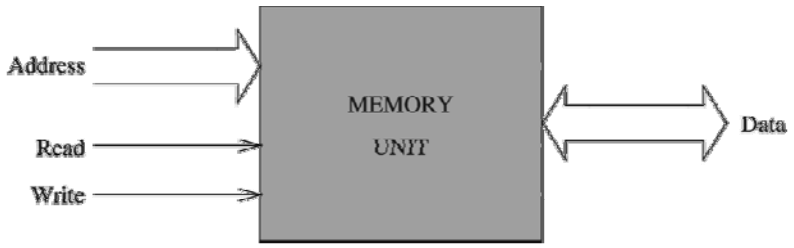





# Computer Organization/Design Overview
## (how small a chunk versus smallest chunk)

- Most modern processors support *byte addressable memory*
  - What does *byte addressable* mean?
- *Byte addressable* means:
  - CPU can address memory in chunks *as small as* one byte
    - This does *not* imply CPU can access 8 bits on any *arbitrary* boundary
  - One byte is the *smallest* unit of memory that can be accessed at once by the processor
    - If processor wants to access a 4-bit value, it must still read 8 bits and then ignore the extra 4 bits

# Computer Organization/Design Overview
## (memory unit)






# Computer Organization/Design Overview
## (memory operations are relatively slow)

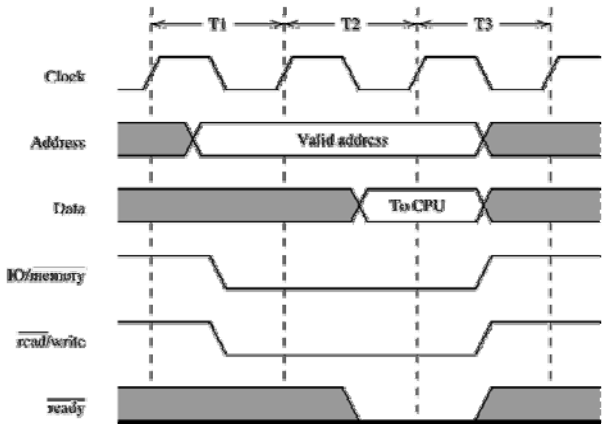
- Memory access is important factor affecting program speed
  - Multiple machine cycles are required when reading from memory
    - Because memory responds much more slowly than CPU can
- Simplified view of what is involved during a memory read
  - Place address on address bus
  - Assert memory read control signal
  - Wait for memory to retrieve data
    - Introduce *wait states* if necessary
  - Read data from data bus
  - Drop memory read signal

# Computer Organization/Design Overview
## (memory operations are relatively slow)

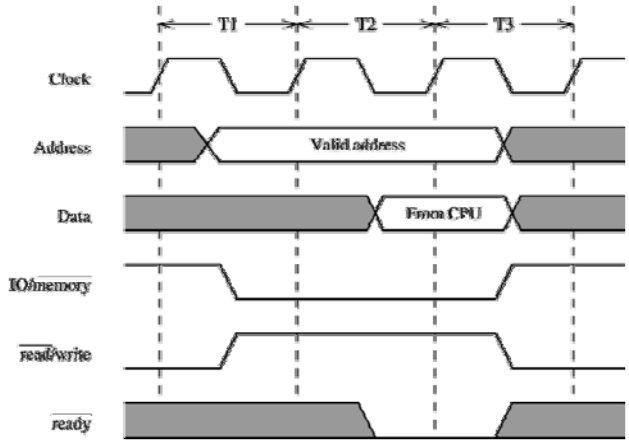
- Memory access is important factor affecting program speed
  - Multiple machine cycles are required when reading from memory
    - Because memory responds much more slowly than CPU can
- Simplified view of what is involved during a memory write
  - Place address on address bus
  - Place data on data bus
  - Assert memory write control signal
  - Wait for memory to retrieve data
    - Introduce *wait states* if necessary
  - Drop memory write signal



# Computer Organization/Design Overview
## (timing waveforms: memory read, no wait state)



# Computer Organization/Design Overview
## (timing waveforms: memory write, no wait state)



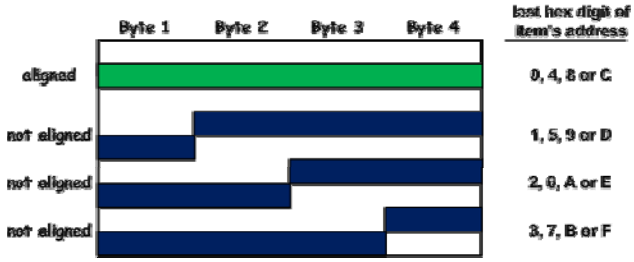



# Computer Organization/Design Overview
## (memory alignment can be important)

- Each architecture specifies what a *word* is (in bits)
  - 32 bits (4 bytes) for MIPS32
- Byte, halfword, word and doubleword items may begin at any *valid* address in memory
- However, starting anything larger than byte at an arbitrary address is not a good idea in general
  - Un-*aligned* items in memory may affect program speed
    - System crash/lock-up and silent program failure are also possible
  - *What (is alignment)? How (does it affect program speed)?*

# Computer Organization/Design Overview
## (what memory alignment means)

- An item is aligned if its address is multiple of its size
  - *E.g.:* 4-byte (32-bit) item – aligned if "address % 4" is 0
    - How should address look like (in *binary*) for item to be aligned?

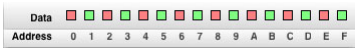



  - What about an 8-byte (64-bit) item?



# Computer Organization/Design Overview
## (how memory alignment may affect program speed)

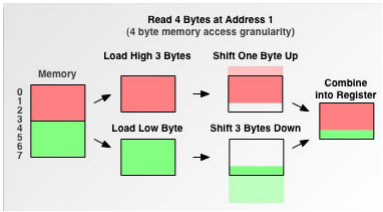
- How programmers see memory vs how processors see memory
  - (for processors w/ 32-bit *memory access granularity*, like 1 based on MIPS32)





- What reading an aligned and unaligned ***4-byte*** item would entail


- How unaligned memory access must be handled

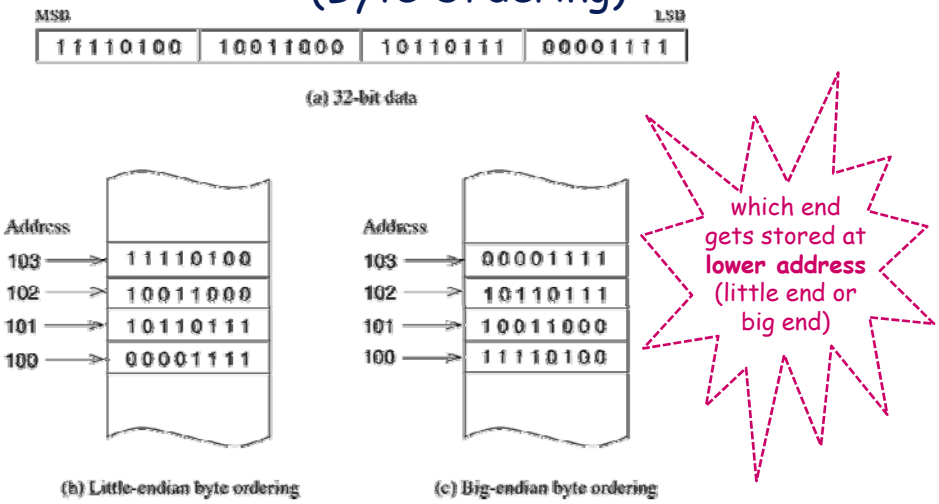


# Computer Organization/Design Overview
## (what if unaligned accesses must be done)

- Hardware support
  - With performance penalty
- Trap to software routine
  - Possible (but typically slower than hardware support)
- ISA (Instruction Set Architecture) support → MIPS32 belongs to this category, as amplified below:
  - Most *load* and *store* instructions operate only on aligned items
    - Recall: MIPS is a *load-store* architecture
    - *E.g.:* load word (**`lw`**)
    - With these, unaligned items will cause bus error
  - Some instructions for manipulating unaligned items
    - *E.g.:* load word left (**`lwl`**) and load word right (**`lwr`**)



# Computer Organization/Design Overview
## (Byte Ordering)



## Computer Organization/Design Overview
(byte ordering)

- Pentium
  - Uses little-endian
- MIPS and PowerPC
  - Use big-endian *by default*
- Modern processors
  - Configurable



## Computer Organization/Design Overview
(von Neumann architecture)

- Used in design of almost all of today's modern computers
  - Often also referred to as *Princeton* architecture
- Architects' views in earlier slides of ***006 Comp…view01***
  - Assume (KiKi's toy is based on) this architecture
- Some designs deviate somewhat from von Neumann
  - Arguably most commonly cited example: *Harvard* architecture
  - How is it different?
  - Need to know key features of von Neumann architecture first

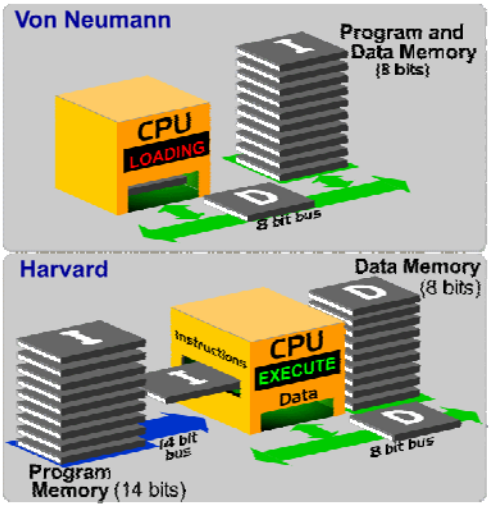
# Computer Organization/Design Overview
## (key features of von Neumann architecture)

- Computer's basic components:
  - *CPU*, *memory* and *I/O devices* (interconnected through *bus*)
- Data and instructions *both* stored in memory
  - Stored program concept
  - Nothing to distinguish *instructions* from *data*
  - Nothing to distinguish different *types* of data
    - Each instruction must know how to interpret data on which it operates
- *One* single, *sequentially addressed* (1-D) memory that is accessed by *address* regardless of what is stored
  - What's stored may be instructions, data, addresses, *etc.*
- Instructions executed *sequentially* unless explicitly altered
  - Instructions laid out in consecutive words in memory
    - Special register (program counter) used to hold address of next instruction
    - Recall: "fetch-decode-execute" cycle



# Computer Organization/Design Overview
## (von Neumann vs Harvard)

**(Different in Memory Aspect)**




**- fetches instructions & data from one memory**
**- limits operating bandwidth**

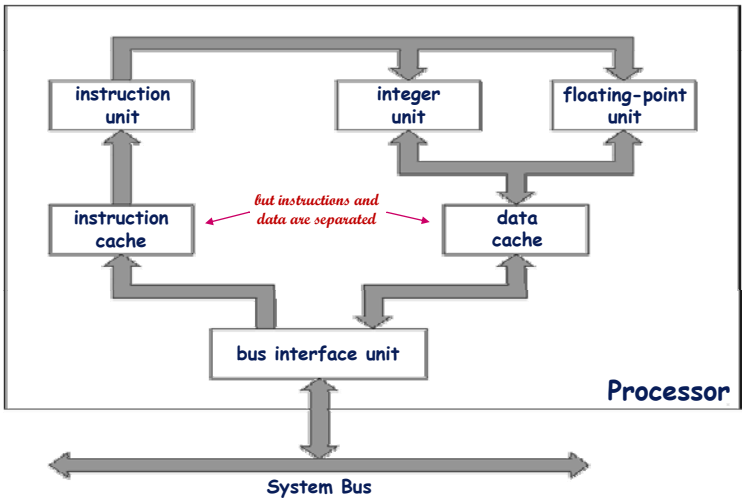
**- 2 separate memory spaces for instructions & data**
**- increases throughput**
**- different program & data bus widths are possible**

# Computer Organization/Design Overview
## ("von Neumann bottleneck")

- Refers to limited throughput between CPU and memory:
  - Amount of work CPU can do in the time it takes to retrieve data from memory is rising
  - With that, amount of time CPU spends idling (*i.e.*, doing nothing while waiting for data to be fetched from memory) outpaces amount of time CPU spends doing actual work
  - Thus, faster CPU no longer translates to faster computer
  - Throughput (bandwidth/latency) between CPU and memory becomes limiting part ("bottleneck") of computer
- So-called because it's a potential bottleneck on computers that use von Neumann architecture:
  - Fundamental view of memory as "one word at a time" device
  - Stored items (data, instructions, *etc.*) must travel between memory and CPU "*one word at a time*"



# Computer Organization/Design Overview
## (recall: Slide 6, *006 Computer…Overview01*)



## Computer Organization/Design Overview
("slightly" non-von Neumann?)

- Modern high performance CPU chip designs incorporate both von Neumann and Harvard aspects
  - Main memory:
    - Not divided into separate instruction and data sections
    - Cache miss → CPU access memory (in von Neumann fashion)
  - On-chip cache memory:
    - Divided into *instruction cache* and *data cache*
    - Cache hit → CPU accesses cache (in Harvard fashion)
- As it appears to programmer:
  - von Neumann
- In hardware implementation:
  - Takes advantage of Harvard efficiencies



## Computer Organization/Design Overview
(von Neumann quiz)

- Which of following may be represented by `39383700h` appearing somewhere in memory?

  *(Assume little-endian and byte addressable.)*
  - Integer `959985408`
  - String `"987"`
  - Float `0.000175680965185165405273437 5`
  - Instruction `xor $24, $9, 0x3700`

## Computer Organization/Design Overview
### (von Neumann quiz continued)

- Which of following may be represented by `39383700h` appearing somewhere in memory?

  *(Assume little-endian and byte addressable.)*

  - Integer `959985408`
  - String `"987"`
  - Float `0.000175680965185165405273437 5`
  - Instruction `xor $24, $9, 0x3700`
- Answer: ***all of them***
  - They are different interpretations of the given bit pattern
- Follow-up question:
  - How would machine know which interpretation to choose?



## Computer Organization/Design Overview
### (von Neumann quiz continued)

- Answer: ***all of them***
  - They are different interpretations of the given bit pattern
- Follow-up question:
  - How would machine know which interpretation to choose?
- Machine must be explicitly told the desired interpretation
  - For instance:
    - As **int**: when it's used with **integer load**
    - As **float**: when it's used with **floating-point load**
    - As **instruction**: where it's to be treated as instruction (**branch** or **jump**)
    - As **null-terminated string**: when it's used with **print string system call**