Homework

- Reading
  - PAL pp 119-125, 161-172
- Continue mp1
  - Questions?
- Continue lab sessions with your section

Memory Architectures

- Von Neumann Architecture
  - Code and data are accessed in one address space
  - The I/O ports may be in:
    - The same address space (Memory Mapped I/O)
    - A separate I/O address space (Intel)
- Harvard Architecture
  - Code is accessed in one address space
  - Data and I/O ports are in another address space

Memory Architectures

- Comparison of Memory Architectures

	Code	Data	I/O Ports	Architecture
Atmega				Harvard
Intel 386				Von Neumann
Motorola 68xxx				

Intel Memory Architecture

- We'll discuss various aspects of memory use:
  - Storage of bytes
  - Storage of words and long words
  - Storage of strings
  - The stack and the stack pointer
  - RAM and ROM
  - Addressing - real mode and protected mode

Storage of Bytes

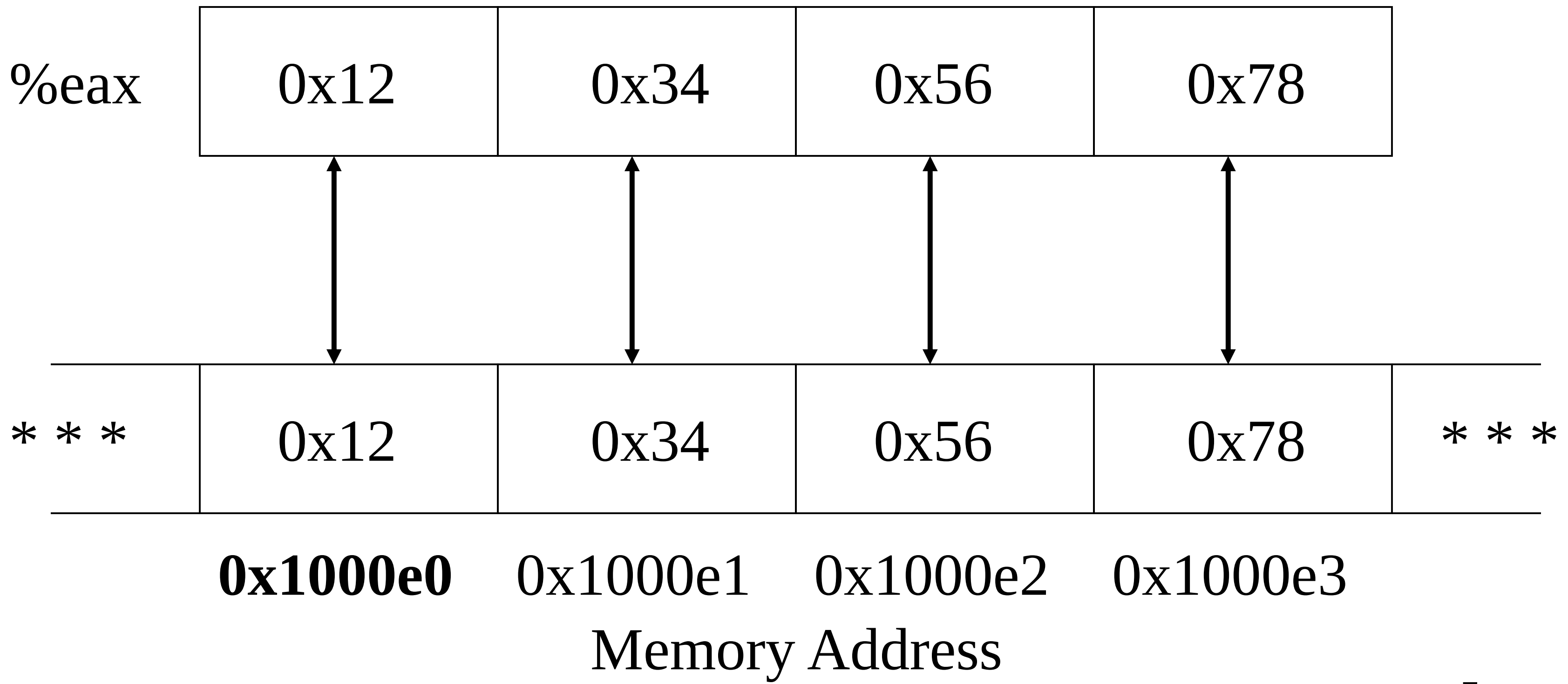
- Memory is byte addressable (8 bits/byte)
- A memory address “points” to the location of a specific byte in memory
- As we know from C data types, a “pointer” can be “cast” to point to a type or structure in memory that is larger than one byte
- In that case, the memory address “points” to the lowest address of the storage area used

Storage of Words and Long Words

- Word Sizes
  - A Word usually means 16 bits or 2 bytes
  - A Long Word usually means 32 bits or 4 bytes
- Many processors (Motorola 68000/PPC and Sun) store words and long words in memory in “Big Endian” fashion
- Intel and Atmega Processors store words and long words in memory in “Little Endian” fashion

“Big Endian”

- Mapping a long word from register to memory:



“Big Endian” Appearance

- Hex value as seen in register:

  %eax: 12 34 56 78

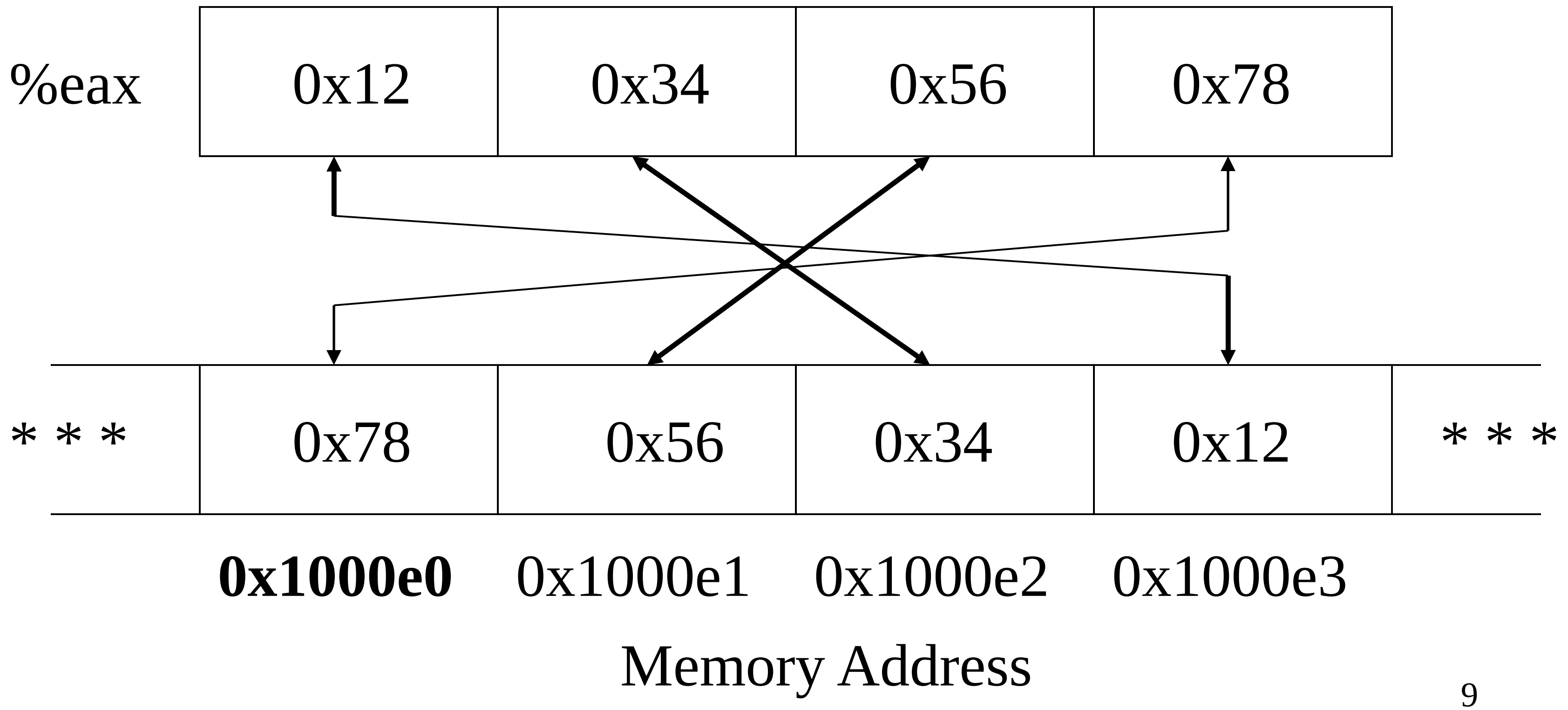
- Hex value(s) as seen in memory bytes (md):

  0x1000e0: 12 34 56 78

- Irrelevant to the end user – self consistent!
- Has a logical appearance to a programmer using a debugger!

“Little Endian”

- Mapping a long word from register to memory:



“Little Endian” Appearance

- Hex value as seen in register:
  - %eax: 12 34 56 78
- Hex value(s) as seen in memory bytes (md):
  - 0x1000e0: 78 56 34 12
- Irrelevant to the end user – self consistent!
- A bit confusing to programmers using a debugger!

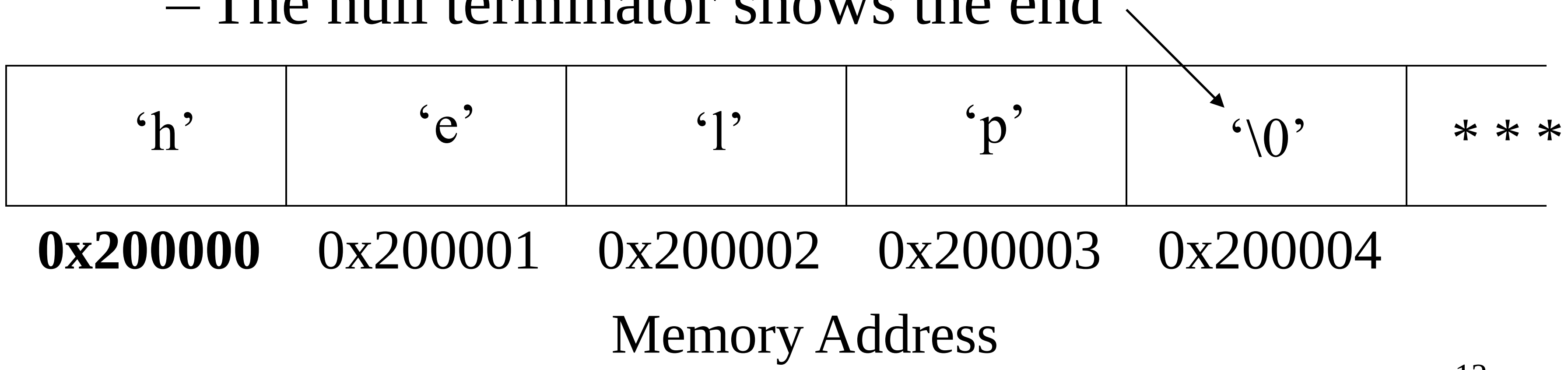
Debugging with “Little Endian”

- Mostly learn to live with it!
- To help with displays, the “real” Tutor monitor has an "mdd" command (memory-display-doubleword)
- This command reorders the bytes and displays four bytes at a time as a double word value:

```
Tutor> mdd 1000e0
1000e0 12345678 ...
```

Storage of Strings

- How are strings stored?
  - Look at string “help” stored at location 0x200000
  - Since memory is “byte addressable” each character (i.e. each byte) has its own address
  - The address of the entire string is the address of the first byte of the string (the lowest address)
  - The null terminator shows the end



The Stack

- One register is called the “extended stack pointer” and is used to implement a stack
- There are various times when data is pushed on the stack and/or popped off the stack that we will study later
  - Function calls and returns
  - Interrupts and returns
  - System calls and returns

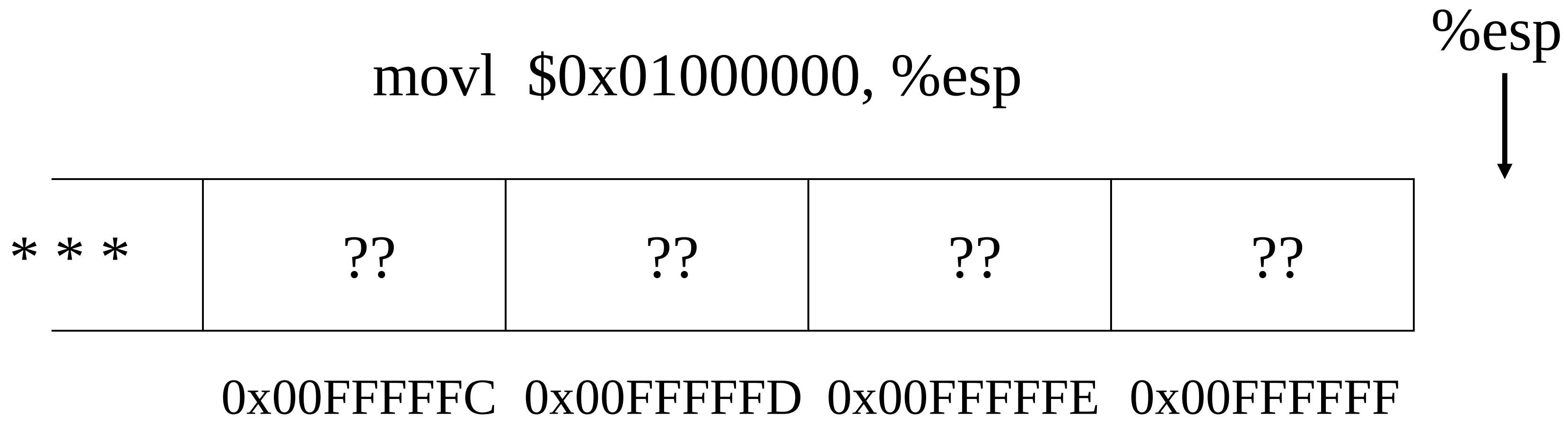
The Stack

- For now, we will just understand how the esp register is used to push and pop data on and off the stack
- Why is a specific register used?   Can't the software use any register for a stack pointer?
- Not for the system stack! Both hardware and software must push and pop data on and off the same stack
- Hardware is designed to use a specific register (%esp) so software must be consistent and use same register

The Stack - Initialization

- Software initializes system stack pointer to point just after the end of available area in memory (i.e., not used for another purpose than stack)
- Contents of the stack area are not initialized

```
movl  $0x01000000, %esp
```

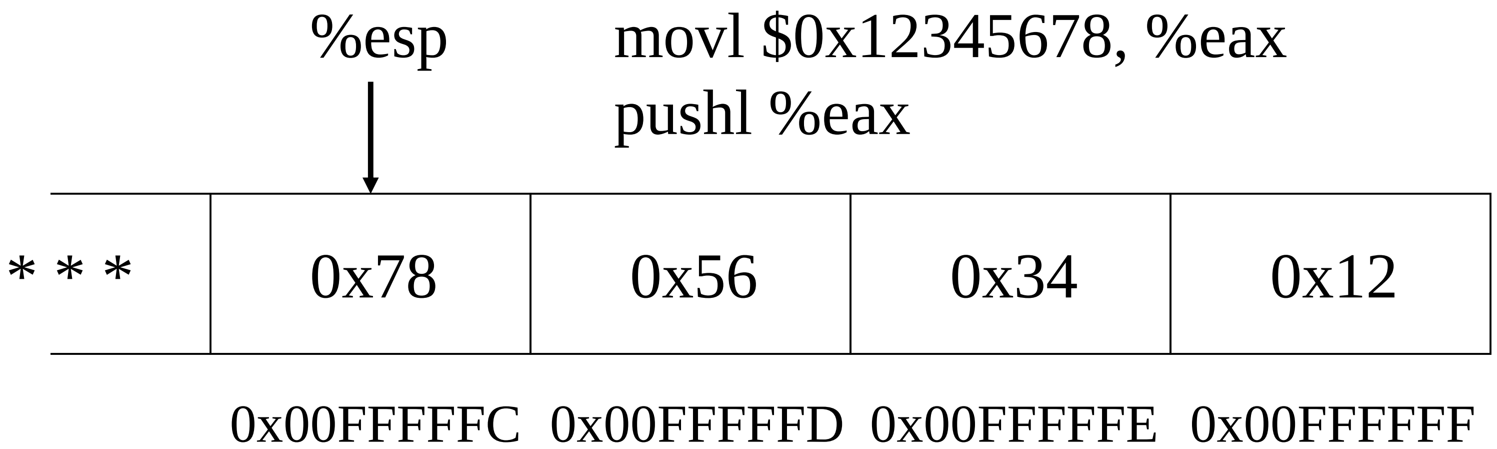


Memory Address

The Stack - Push

- When data is pushed onto the stack, the esp register is decremented and the value being pushed is stored to memory at the address value ("pointer") stored in the esp register

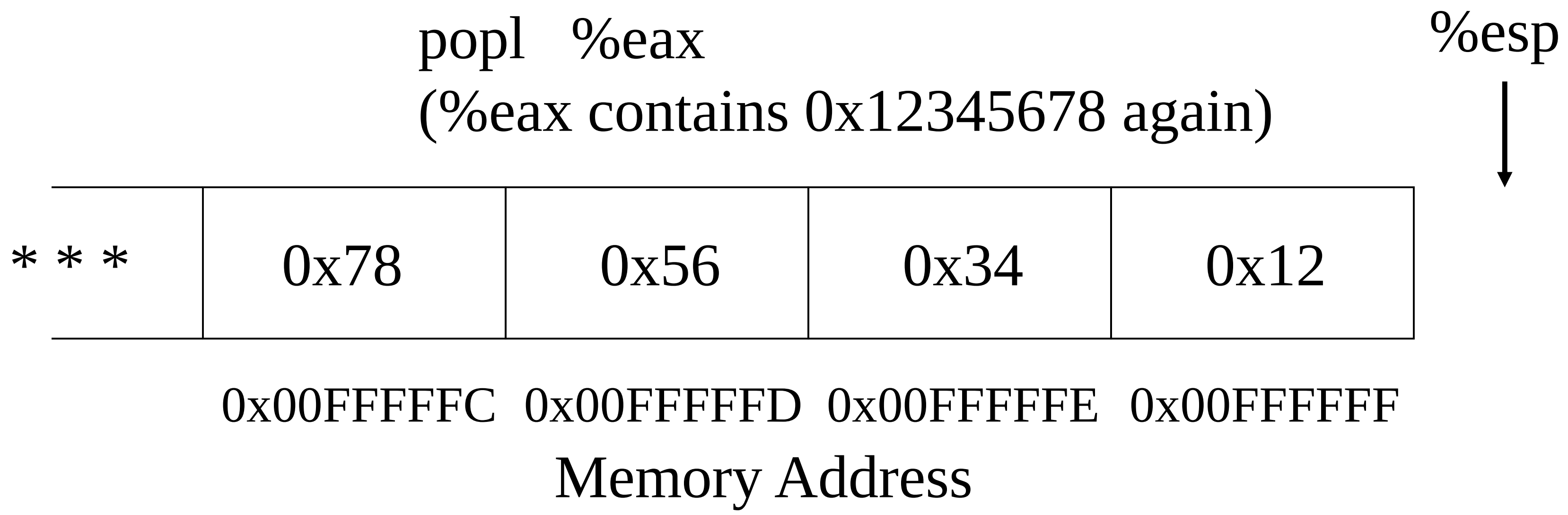
```
movl $0x12345678, %eax
pushl %eax
```



Memory Address

The Stack - Pop

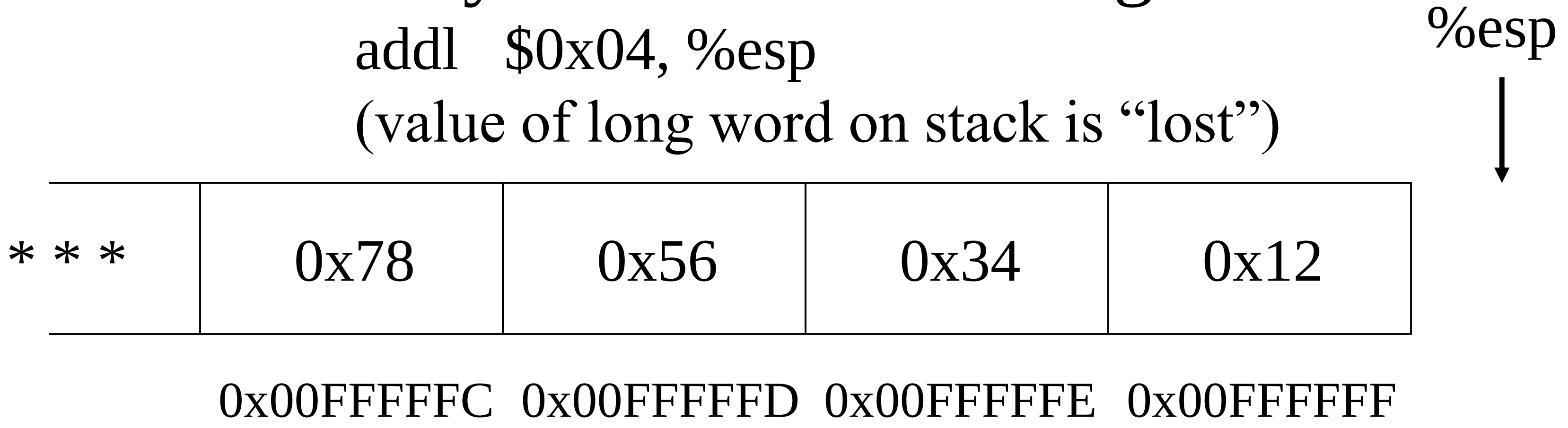
- When data is popped off the stack, the value of the data is read from memory at the address value ("pointer") stored in the esp register and the esp register is incremented



The Stack - Remove

- When data is removed from the stack, the value of the data is NOT read from memory
- The esp register is merely incremented by the size in bytes of the data being removed

addl $0x04, %esp
(value of long word on stack is "lost")



Memory Address

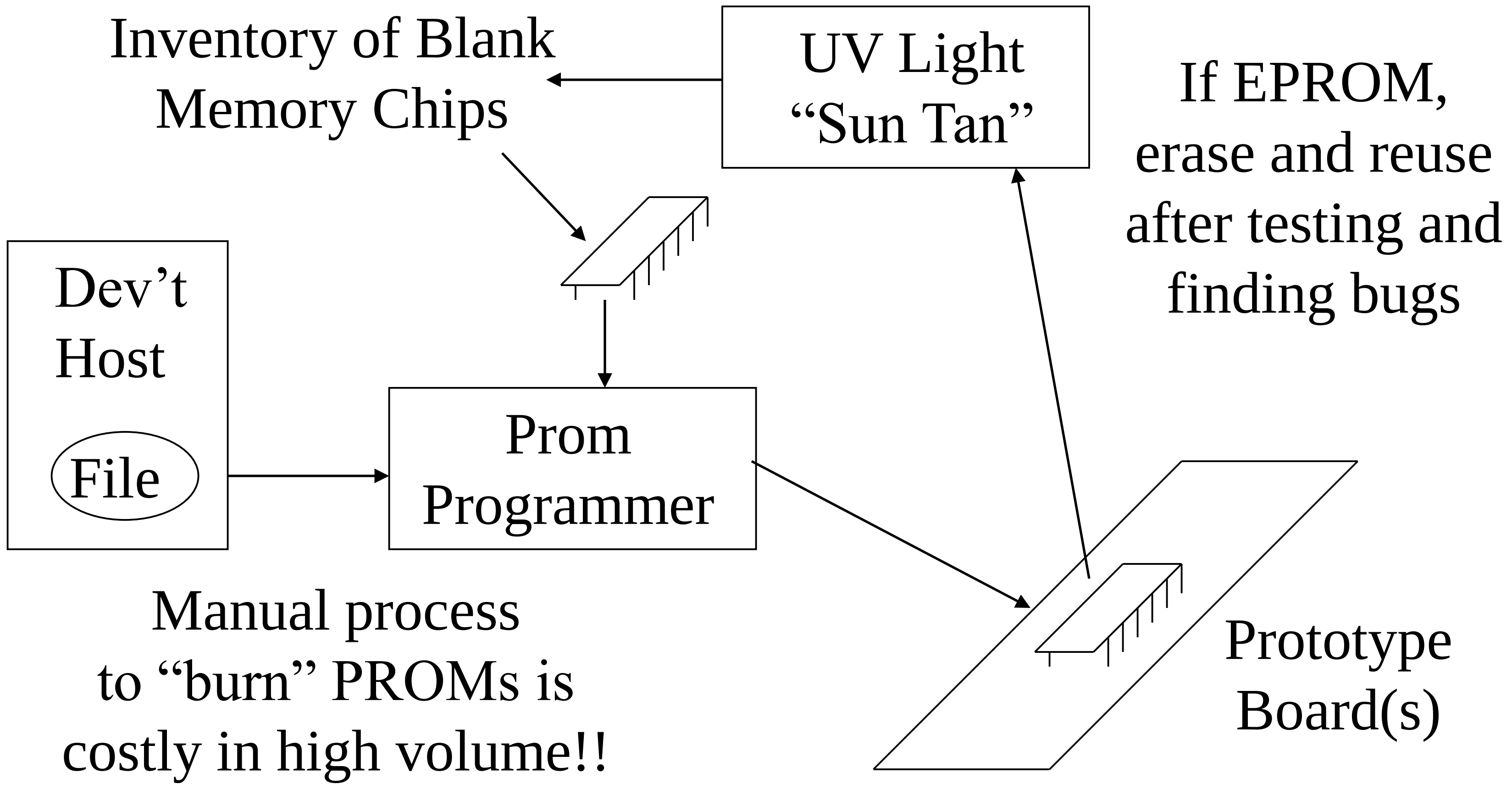
Memory Types

- Random Access Memory (RAM)
  - Contents are “volatile” (lost when power is off)
  - Can be loaded with code from a non-volatile media/source such as a disk or a network server
  - Can be used for reading and writing data via normal reads and writes to the memory address
  - Can be overwritten unintentionally, e.g. using a “bad” pointer when storing data in memory
  - Faster access

Memory Types (Continued)

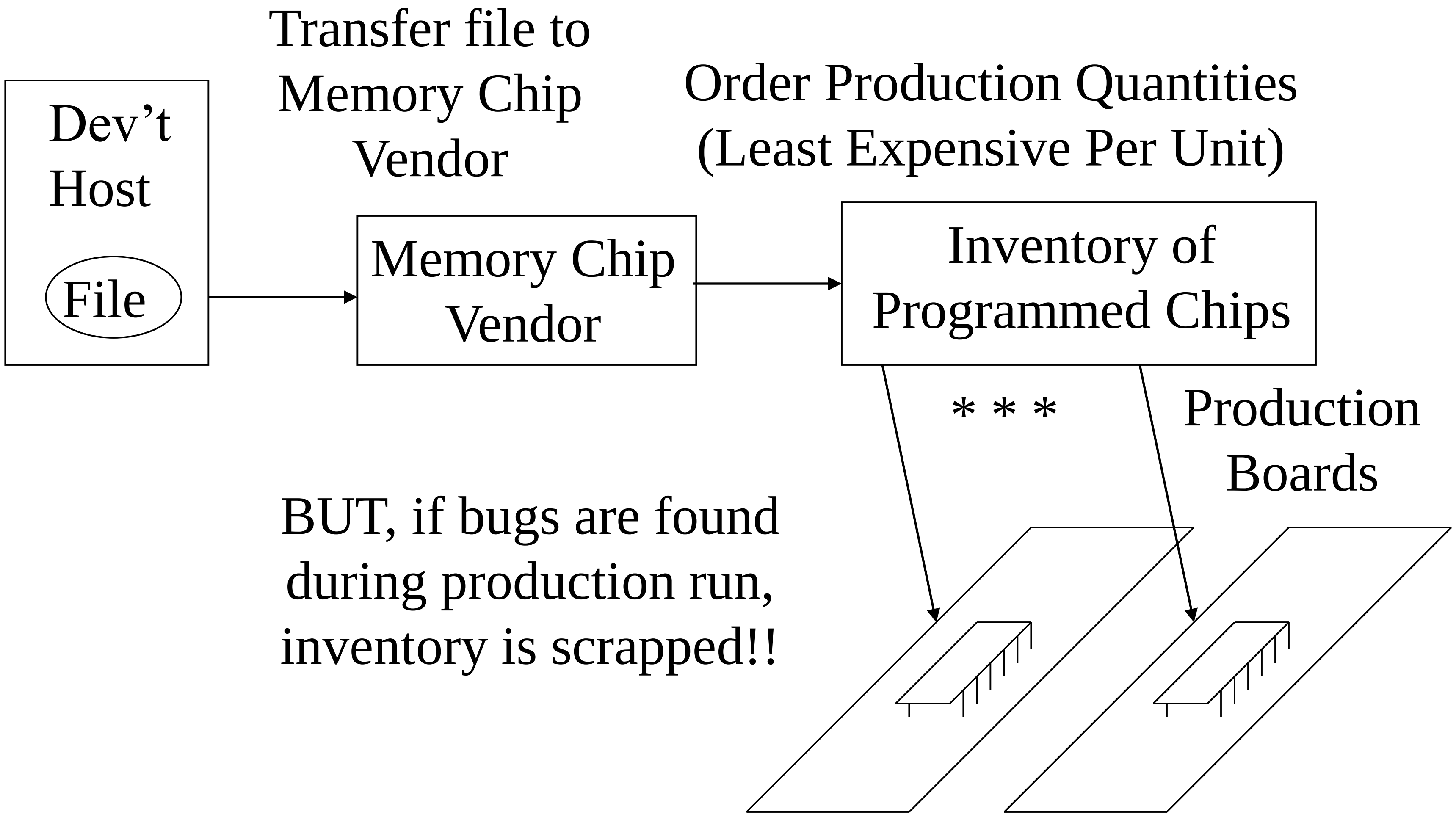
- Read Only Memory (ROM)
  - Contents are “non-volatile” (saved with power off)
  - Contains code needed to “boot” the processor
  - Can be used for reading (but not writing) data
  - Programmable Parts: Low Volume/High Cost (Development and Testing)
  - Preprogrammed Parts: High Volume/Low Cost (Production)

Software in Embedded System Memory (Low volume / High Cost)


Inventory of Blank Memory Chips
UV Light "Sun Tan"
If EPROM, erase and reuse after testing and finding bugs
Dev't Host
File
Prom Programmer
Prototype Board(s)

Manual process to "burn" PROMs is costly in high volume!!

Software in Embedded System Memory (High Volume / Low Cost)


Transfer file to Memory Chip Vendor
Order Production Quantities (Least Expensive Per Unit)
Dev't Host
File
Memory Chip Vendor
Inventory of Programmed Chips
* * *
Production Boards

BUT, if bugs are found during production run, inventory is scrapped!!

EEPROM/Flash Upgrade

- For some computer systems (e.g. PC’s) and embedded devices, the pre-programmed parts are electrically writeable using procedures that are not normal software write operations
- These parts can be reloaded using a special procedure, e.g. BIOS update for a PC
- Risk: If anything goes wrong, the device may become unusable and the part need replacing