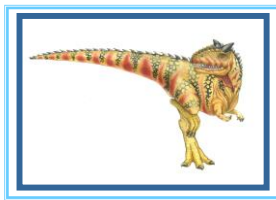


# BASIC COMPUTER ORGANIZATION

---



## Topics

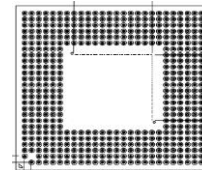
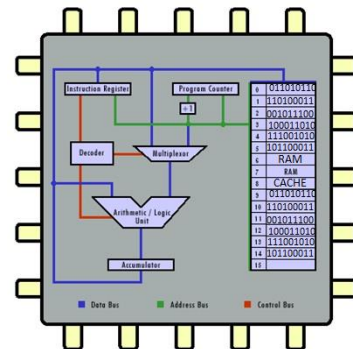
---

- CPU Structure
- Registers
- Memory Hierarchy (L1/L2/L3/RAM)
- Machine Language
- Assembly Language
- Running Process



## CPU – Central Processing Unit

- Roughly on 1 cm<sup>2</sup>
- 500 to 1000 pins (input/output/control and power)
- Recent CPU's have 4.5 billion transistors! (on 1 cm<sup>2</sup> !)
- L1/L2/L3 Cache size ~ 256K/1MB/6MB (L1/L2 inside CPU)
- RAM ~ 4GB to 128GB (out of CPU)
- Pins connected to buses that travel across the system board to other devices (disk controllers, graphic cards, ...)
- Registers: special memory units inside the chip with fastest access time.  
Types: 16bit, 32bit, 64bit registers  
Number: 16 to 128 registers on chip
- Important Registers:
  - **Program counter (PC)**: holds a pointer to current command in the running program (in RAM)
  - **Instruction Register (IR)**: holds the currently running instruction
  - **Accumulator**

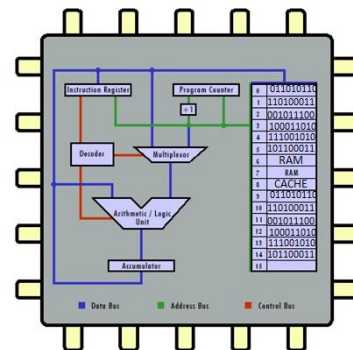


<http://courses.cs.vt.edu/csonline/MachineArchitecture/Lessons/CPU/Lesson.html>



## CPU – Central Processing Unit

- **Machine Code:**
  - First loaded to RAM and then fetched to CPU CACHE (by pages, not all program!)
  - Machine instruction length is usually 16bit, 32bit, and even up to 128bit
  - At each step an instruction is loaded to the Instruction Register (IR), decoded and executed
- **ALU – Arithmetic Logic Unit:**
  - Performs all the mathematical calculations of the CPU
  - The ALU can add, subtract, multiply, divide, and perform a host of other calculations on binary numbers
- **Control Unit:**
  - this component is responsible for directing the flow of instructions and data within the CPU
  - The Control Unit is actually built of many selection circuits such as decoders and multiplexors
  - In the diagram above, the Decoder and the Multiplexor compose the Control Unit

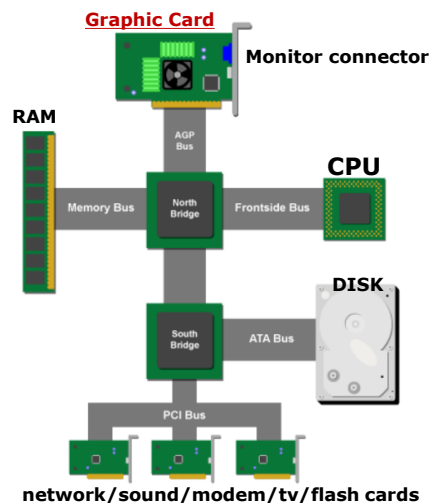


<http://courses.cs.vt.edu/csonline/MachineArchitecture/Lessons/CPU/Lesson.html>



# Computer System BUSES

- The information highway for the CPU
- Buses are bundles of bits that carry data between components
- The three most important buses are:
  - **Address Bus (32-64 bit)**  
Used to specify a physical address. The processor or DMA-enabled device needs to read or write to a memory location, it specifies that memory location on the address bus (the value to be read or written is sent on the data bus). The width of the address bus determines the amount of memory a system can address. For example, system with a 32-bit address bus can address  $2^{32}$  (4 GB) bytes.
  - **Data Bus (8-64 bits)**  
allows data flow in both directions
  - **Control Bus**  
carries commands from the CPU and returns status signals from devices. Example: if the data is being read or written to the device the appropriate line (read or write) will be active (0/1 bit).

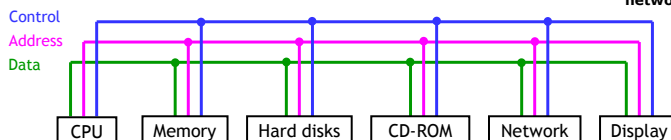
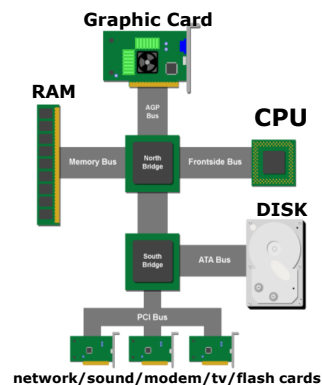


[http://testbench.in/introduction\\_to\\_pci\\_express.html](http://testbench.in/introduction_to_pci_express.html)



# Read/Write BUS Control

- To send or receive data, a device must acquire control of the bus from the CPU
- It may wait on a queue, and when the cpu grants control proceeds as follows:
  - To send data x to a device y it needs to
    - Put x on data bus
    - Put an address on address bus
    - Send a write signal thru control bus
- All devices are listening but only the targeted device will copy the data and send a received signal after read (thru the control bus)

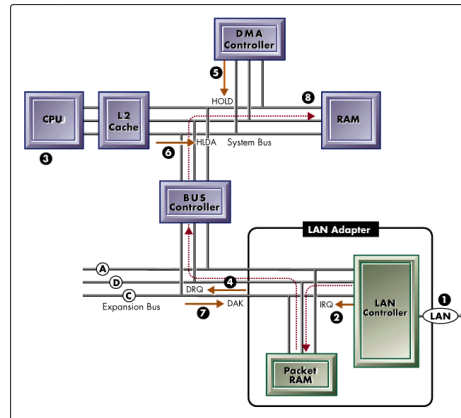


[www.cs.wustl.edu/~fredk/Courses/cse422/sp04/Lectures/io.ppt](http://www.cs.wustl.edu/~fredk/Courses/cse422/sp04/Lectures/io.ppt)



## DMA – Direct Memory Access

- The CPU is too expensive to be engaged with slow I/O transfers:
  - A typical CPU operates at several GHz (i.e., several  $10^9$  instructions per second)
  - A typical hard disk has a rotational speed of 7200 revolutions per minute for a half-track rotation time of 4 ms
  - This is 4 million times slower than the processor!
- Instead the CPU initiates a transfer with the DMA, does other operations while the transfer is in progress, and receives an **interrupt** from the DMA controller when the operation is done
- So in effect, the DMA is a mini-controller that works for the CPU and does I/O transfer jobs for it
- Some systems contain several DMA's
- DMA is also used for “memory to memory” copying in multi-core processors
- Hardware systems such as disk drives, graphic network and sound cards use the DMA for passing data around

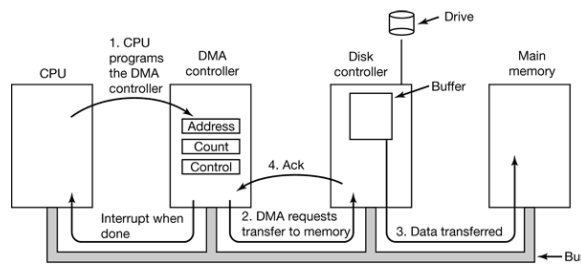


<http://support.novell.com/techcenter/articles/ana19950501.html>



## DMA – Direct Memory Access (1)

- The CPU programs the DMA controller by writing to its registers the addresses and operation codes (what to transfer and to where?)
- It also issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum
- When valid data are in the disk controller's buffer, DMA can begin
- The DMA controller initiates the transfer by issuing a read request over the bus to the disk controller
- The disk controller does not know or care whether it came from the CPU or from a DMA controller

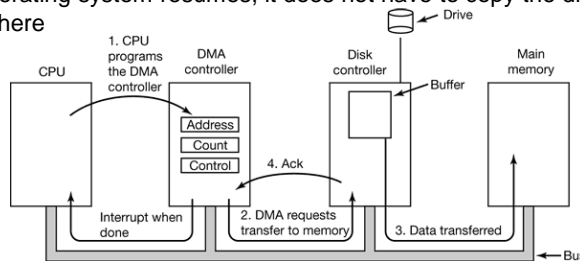


<http://lovingod.host.sk/tanenbaum/INPUT-OUTPUT.html>



## DMA – Direct Memory Access (2)

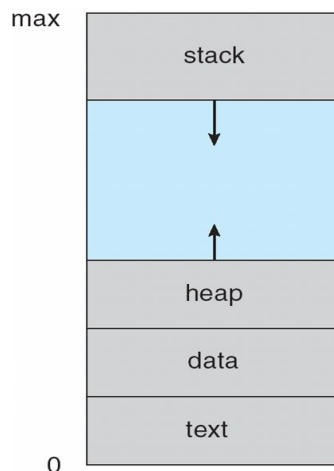
- The memory address to write to is on the address bus, so when the disk controller fetches the next word from its internal buffer, it knows where to write it
- The write to memory is another data bus cycle
- When the write is complete, the disk controller sends an acknowledgement signal to the DMA over the control bus
- The DMA controller then increments the memory address to use and decrements the byte count - this goes on until the byte count reaches 0
- The DMA controller interrupts the CPU to let it know that the transfer is now complete
- When the operating system resumes, it does not have to copy the disk block to memory; it is already there



<http://lovingod.host.sk/tanenbaum/INPUT-OUTPUT.html>



## Process in Memory





## How Program looks in Disk or Memory

```
0110011111000101010100110010001001111101000011010000000001001100110100010011011111001100101010
11100011010011010101010001101000011100010011101000111000101100111101110001101101100010000101110
010010010100001000010111010001100011110101111100010101101001001011010000010100001010100011001
101001111000001101101000110110111011101010111000011001110010001100010001011110111011001000111
111110010100011101011000000100000001110001001010101101111011110111101001010101011110011000
000010011110100111001010111111110011101011100011101101111110011101101000110101010101010010
11111101101101001000000010000100110001100111100000100110011111111111110001001100011010101011
10111011110001010111110111100101111101100000100011111001101000111101001110001101100001110
01011100011010111010001110000001101100111100010001011010000110101000000110101001010011111001011
11011101110011000010:
011101011010100101110:
101110001100110111011:
010011000110011100010:
10110111101010100100:
110100001010100011010:
111011100001011000110:
00101111010110001001:
100100010111011001001:
101100010010100100100:
11000000000101010110:
110001000011101011100:
000000011001001100110:
001011101100111111000:
10000001100100010000:
11011001101010011011:
11111001111101110011000110000111001101010101100010000110101110101010111000011
110011000010100000110011011000011100101001100001001110111000010110011011100000101001110111110
01101011100111100100100100000101111011001111110010110000111010000101101001100001100111011010
101010101101100010000011001101110010110011110100000100111001110010001010001001110101100110010011
10101100111110010101100000001101110001010010011110111100110110011101111100011101010011010011011
10110100100011011111101001100001001011100010000111101110001101100111100111110000001011011111
```

**RAM is really a 1-dimensional array!**  
**So you should think of RAM as a very large C array:**

**byte\* memory[4294967296]**

**where 4GB = 4294967296 (but could be larger!)**

**This is a two-dimensional view, but in memory or disk it is really 1-dimensional!**



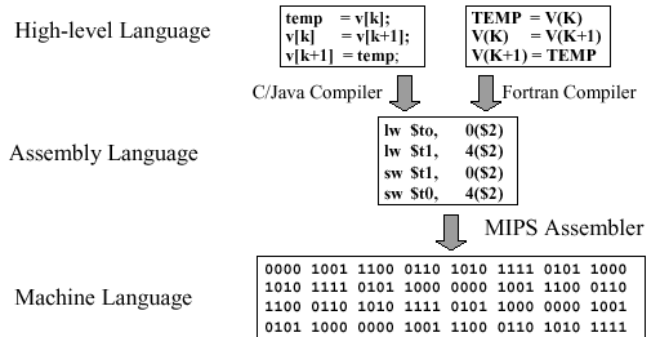
## Example

```
double PI = 3.14159 // data section
unsigned int u = 27 // data section
char * str = "No changes allowed"; // data section

int foo()
{
    char *pBuffer; // nothing allocated yet (excluding the pointer itself,
                  // which is allocated here on the stack).
    bool b = true; // Allocated on the stack
    if(b)
    {
        long int x, y, z ; // Create 3 longs on the stack! (local vars)
        char buffer[500]; // Create 500 bytes on the stack! (local var)
        pBuffer = new char[500]; // Create 500 bytes on the heap! (array of char objects)
    } // buffer is deallocated here, pBuffer is not!
} // oops there's a memory leak, should have called:
  // delete[] pBuffer;
  // new operator always uses the heap ! Why? Think of WinWord.exe reading a doc ...
```



## High code → Assembly → Machine Code



<http://www.cise.ufl.edu/~mssz/CompOrg/CDA-lang.html>



## C code → Assembly → Machine Code

C code:	c = a+b	a, b, c variables in C language
Assembly (MIPS):	add \$8, \$17, \$18	Add the contents of CPU registers \$17 and \$18 and put the result in register \$8
Machine Code:	00000010001100100100000000100000	
Decoder:		
	000000	10001 10010 01000 00000 100000
	add	reg17 reg18 reg8 shamt funct

1. The C language has no limit on number of variables but the MIPS Architecture has only 32 registers available
2. So we may have 700 C variables but the assembler will have to manage all of them with only 32 registers!
3. Every MIPS machine instruction has 32 bits length



## Machine Code Structure (MIPS R-Type)

000000	10001	10010	01000	00000	100000	32 bits instruction
OPCODE	RD	RS	RT	SHAMT	FUNCT	meaning

OPCODE	Basic Operation Code	6 bits
RS	First Source Register (operand)	5 bits
RT	Second Source Register (operand)	5 bits
RD	Destination Register (operand)	5 bits
SHAMT	Shift Amount	5 bits
FUNCT	Specific Functionality	6 bits

- MIPS has 32 registers, so 5 bits are enough to address them all
- FUNCT is used to select a specific variant of the operation code
- OPCODE has exactly 6 bits, which means that we can have at most 63 operations in our language ... could be risky ...?
- MIPS has more instruction types different from the above!



## Machine Code Structure (MIPS R-Type)

### MIPS32 Add Immediate Instruction

001000	00001	00010	0000000101011110
OP Code	Addr 1	Addr 2	Immediate value

Equivalent mnemonic: **addi \$r1, \$r2, 350**

[http://en.wikipedia.org/wiki/File:Mips32\\_addi.svg](http://en.wikipedia.org/wiki/File:Mips32_addi.svg)

**R1 = R2 + Immediate\_Value**





## More Example of Assembly Instructions

- sub \$8, \$5, \$1** Subtract the contents of CPU registers \$5 and \$1 and store the result in register \$8  
Exactly as in:  $\$8 = \$5 - \$1$   
In C language,  $a = b - c$  can be compiled to such instruction
- lw \$17, 101 (\$8)** Load word from memory address  $\$8 + 101$  to register \$17  
The number '101' is called the '**offset**' and the memory address stored in register \$8 is called the '**base register**' and the address it stores is called '**base address**'. This type of instructions is useful for scanning C arrays (the base address is the array pointer and the offset runs from 0 to n-1)
- sw \$4, 1010 (\$9)** Store word from memory address \$4 to memory address  $\$9 + 1010$ . The number '1010' is called the '**offset**' and the memory address stored in register \$9 is called the '**base address**' (register \$9 is called the '**base register**')



## C to Assembly Example

### C program

A is an array of size 100

```
for (i=0; i<100; i++)
    A[i] = A[i] + 127
```

### Assembly program

#### MIPS R2000

Before foo: register \$1 points to address A[0], \$2=127, \$4=100

```
foo: lw $3, 0($1)      # load A[i] from $1 to $3
     add $3, $2, $3     # A[i] = A[i] + 127
     sw $3, 0($1)      # store A[i]
     addi $1, $1, 1     # i = i + 1
     bne $1, $4, foo    # if i != 100, continue at "foo",
                       # otherwise at next instruction
```

**Addi** = Add immediate value (direct in instruction, not thru register)  
**bne** = branch not equal: if registers not equal jump to label (foo)  
else continue to next instruction



## Machine Code: I-type instruction format

100011	01001	01000	0000011111010000	32 bits instruction
lw	\$9	\$8	2000	Semantics

lw	Load Word OPCODE	6 bits
\$9	Register 9	5 bits
\$8	Register 8	5 bits
2000	Offset from address \$8	16 bits

- Instruction type is determined by opcode (first 6 bits)
- Note that offset has room for 16 bits only, so maximal offset value is  $2^{16}-1 = 65535$



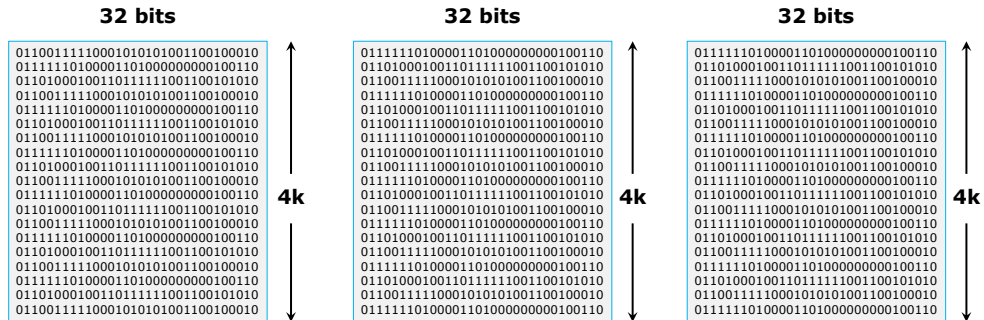
## More Info on MIPS and x86/amd64

- To get more information on MIPS here is a very short summary of all the MIPS language:  
**A Minimalistic Introduction to MIPS Instruction**  
[http://people.cs.pitt.edu/~xujie/cs447/MIPS\\_Instruction.htm](http://people.cs.pitt.edu/~xujie/cs447/MIPS_Instruction.htm)
- This MIPS reference is barely 5 pages !  
(*Simplicity is the ultimate sophistication* ... -Leonardo Da Vinci)
- If you want, here is a full reference to Intel® 64 and IA-32 Architectures Software Developer's Manual (3289 pages!)  
<http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>  
Highly complex ...



## CPU/Memory View of the Program

- The CPU has a very specific view of the program
- First, the binary data is viewed as a sequence of 32 bits instructions
- Second, these instructions are grouped into pages of (usually 4K) instructions



A program like Microsoft Word which can reach 35 MB size (with DLL's) may get to 9000 pages. The modern CPU has large caches for storing many pages at the CPU itself and thus save a lot of time (the long journey to main memory is very expensive!)  
L1 ~ 64 pages, L2 ~ 256 pages, L3 ~ 2048 pages