**@bhansa**

**INTRODUCTION**

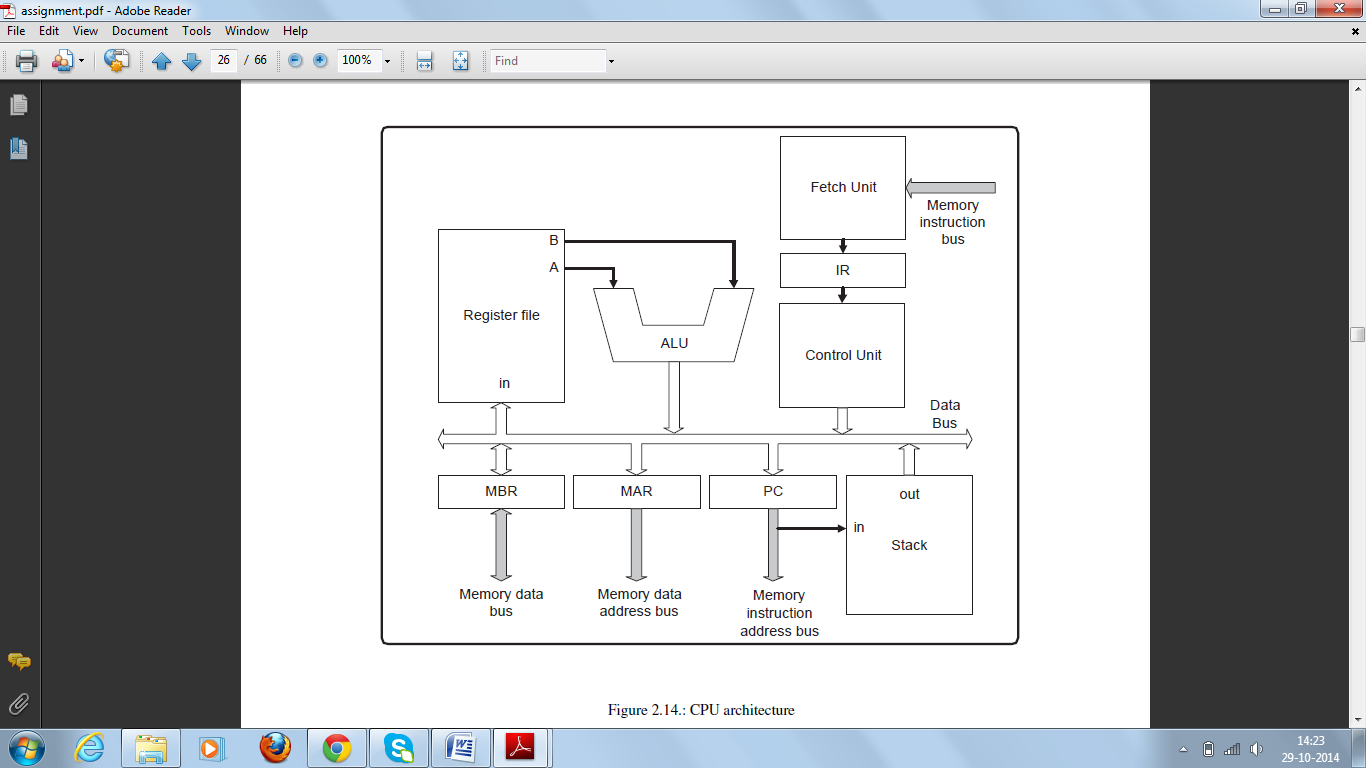
32 bit microprecessor is a simple von-Neumann computer system. It is build of the 3 main units that characterize von-Neumann machines.

1. a CPU: Central Processing Unit
2. a Main memory
3. an I/O unit

It provides two separate sets of buses for program instructions and data and utilizes a single block memory. It supports its own instruction set. Hence, an assembler has to be developed as a basic software development environment.

In this project, we will present 32 bit microprocessor specifications. The first section demonstrates the CPU. The next two sections, we follow a top-down approach to illustrate the main memory and I/O unit highlighting their sub-components and their functionalities.

**CPU ARCHITECTURE:**



The CPU as shown in figure is built of different components as listed below:

1. Register File: Holds intermediate computation data.

2. ALU: Arithmetic and Logic Unit. It

.

3. Fetch Unit: Fetches instructions from the memory.

4. IR: The Instruction Register: It holds the current instruction. 32-bit register.

5. Control Unit: It consists of the Decode Unit and the Execute Unit.

6. PC: Program Counter3: 12-bit up-counter: It holds the address of the next instruction to be fetched.

7. Stack: Stores the contents of the Program Counter PC during subroutine calls and restores it when

returning from subroutines.

8. MAR: Memory Address Register: 12-bit registers that holds the address of the memory location to

be accessed by the Data Path.

9. MBR: Memory Buffer Register: 32-bit registers that holds the data to be written or read to and from

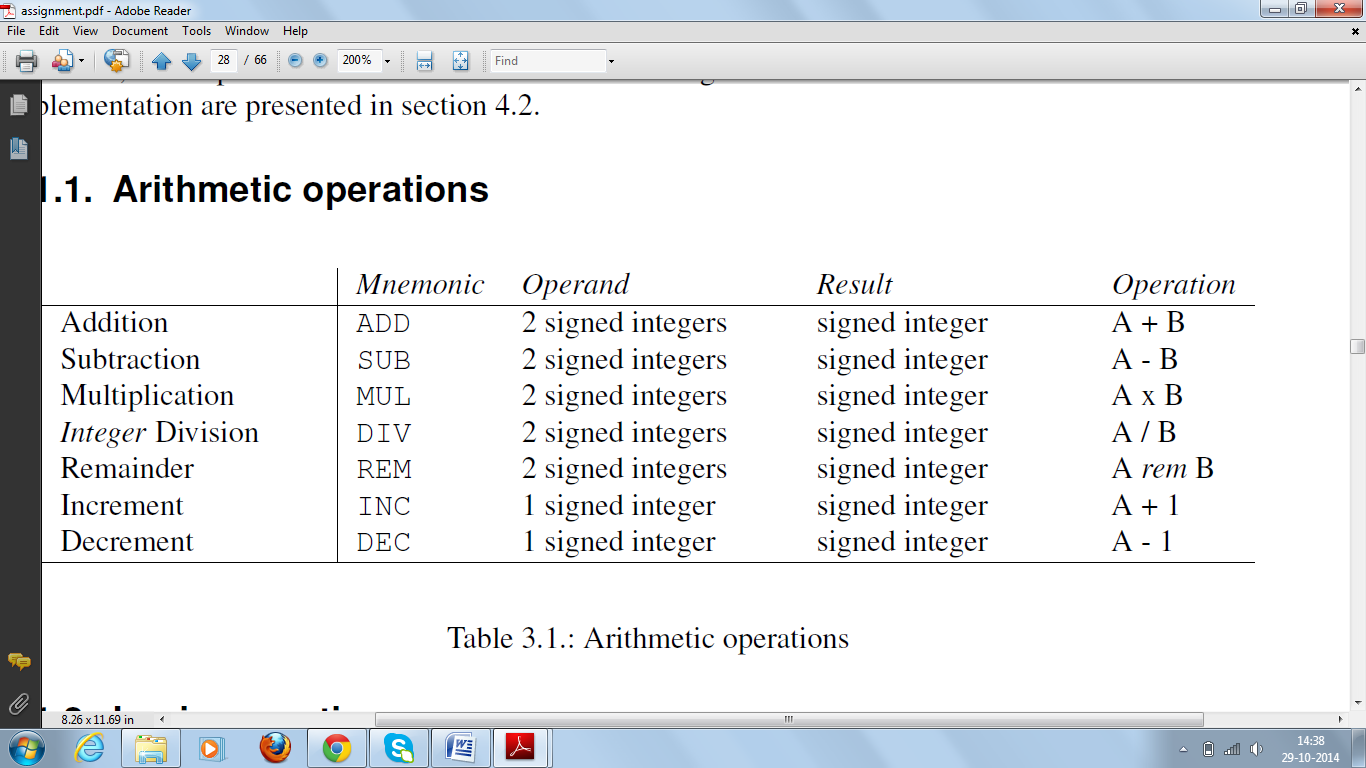
the main memory. It also holds the data to be transmitted or received to and from the I/O Unit.

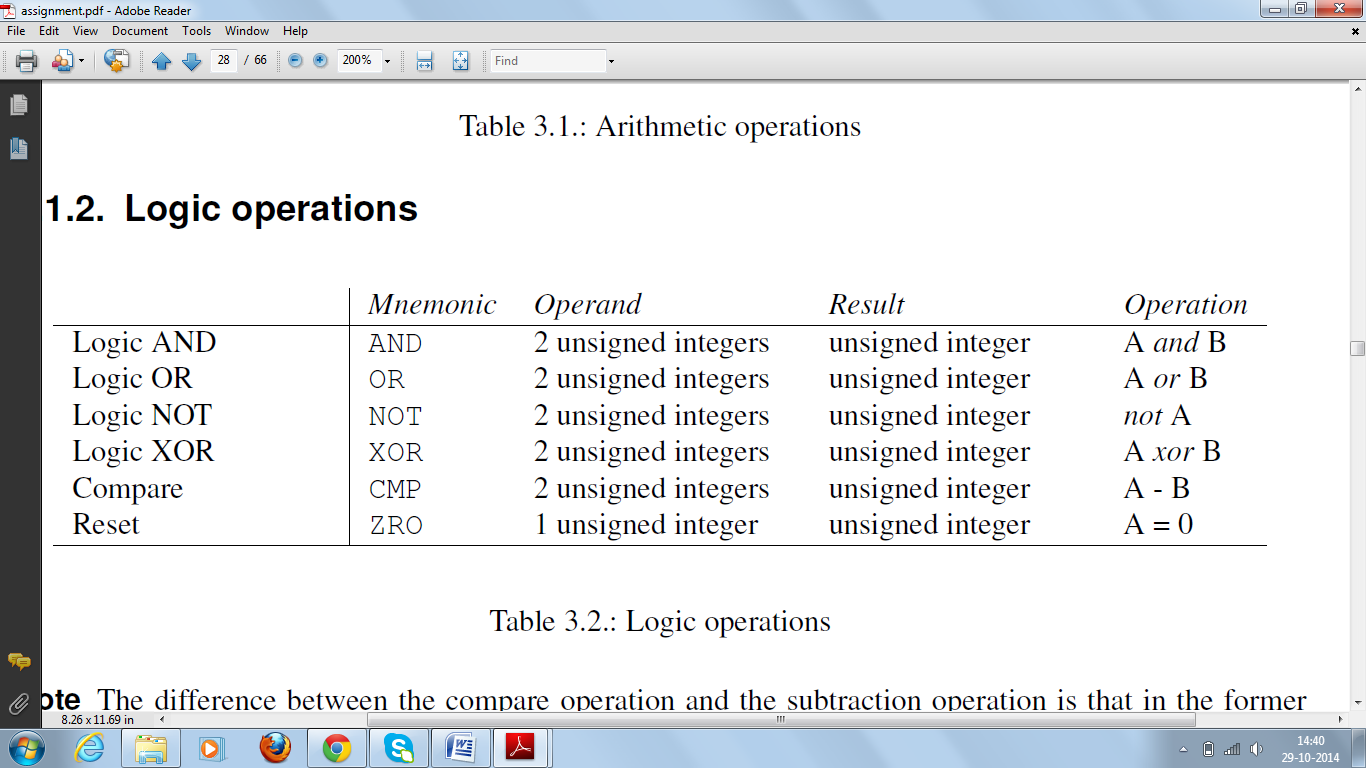
10. Data Bus: 32-bit bus. It is implemented as a multiplexer since internal tri-state signals are not

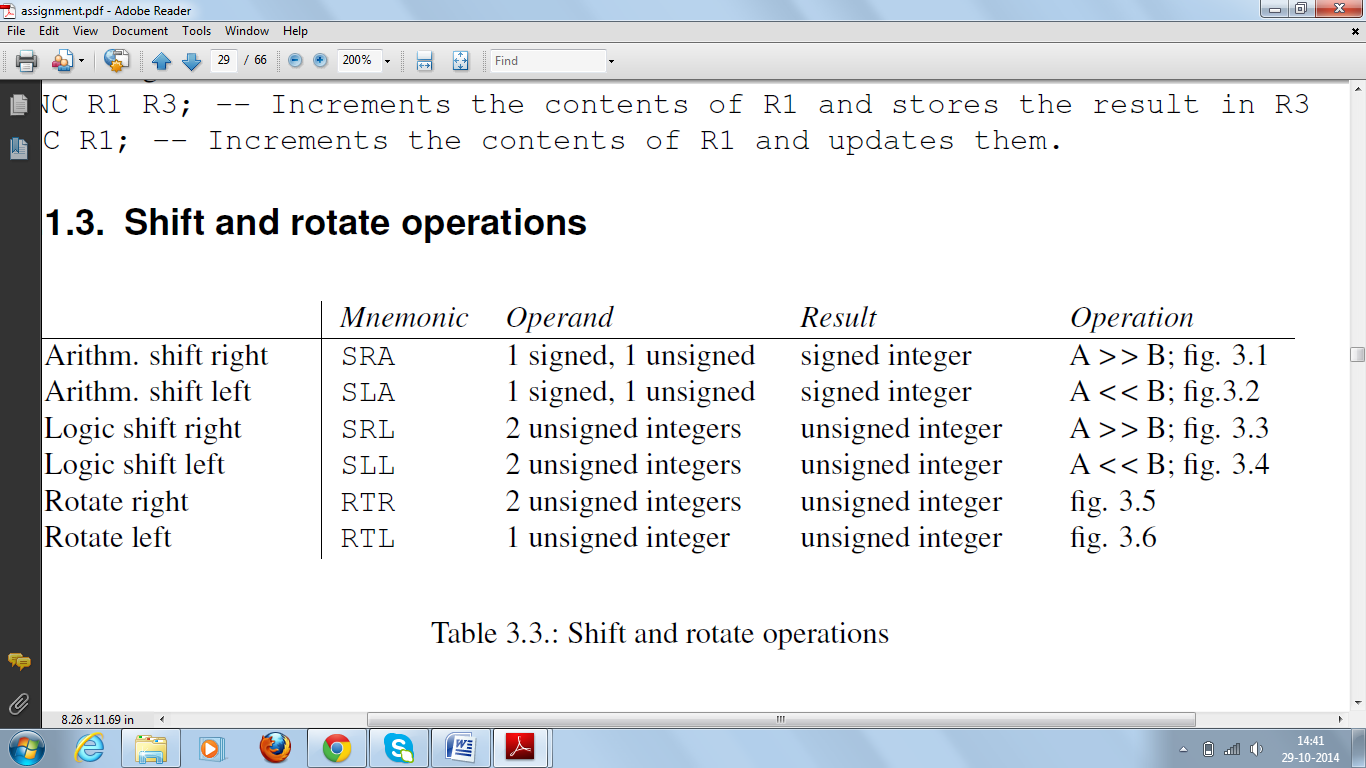
allowed in the majority of FPGA architectures. The Data Bus has 4 inputs: the ALU, the Stack, the

Control Unit and the MBR. However, the term “Data Bus” does not explain its function completely.

**INSTRUCTION SETS**

****





**INSTRUCTION TYPES**

1. Register-direct

Data is stored in the accumulator or a register of the register file

2. Register-indirect

The location of the data to be manipulated is stored in a register of the register file.

3. Register-indexed

Similar to register-indirect but the memory address is expressed by the sum of the contents of a

general-purpose register and the contents of an index register.

4. Stack-register

Data is stored in the stack segment of the main memory. A stack pointer is used to access data using

this addressing mode.

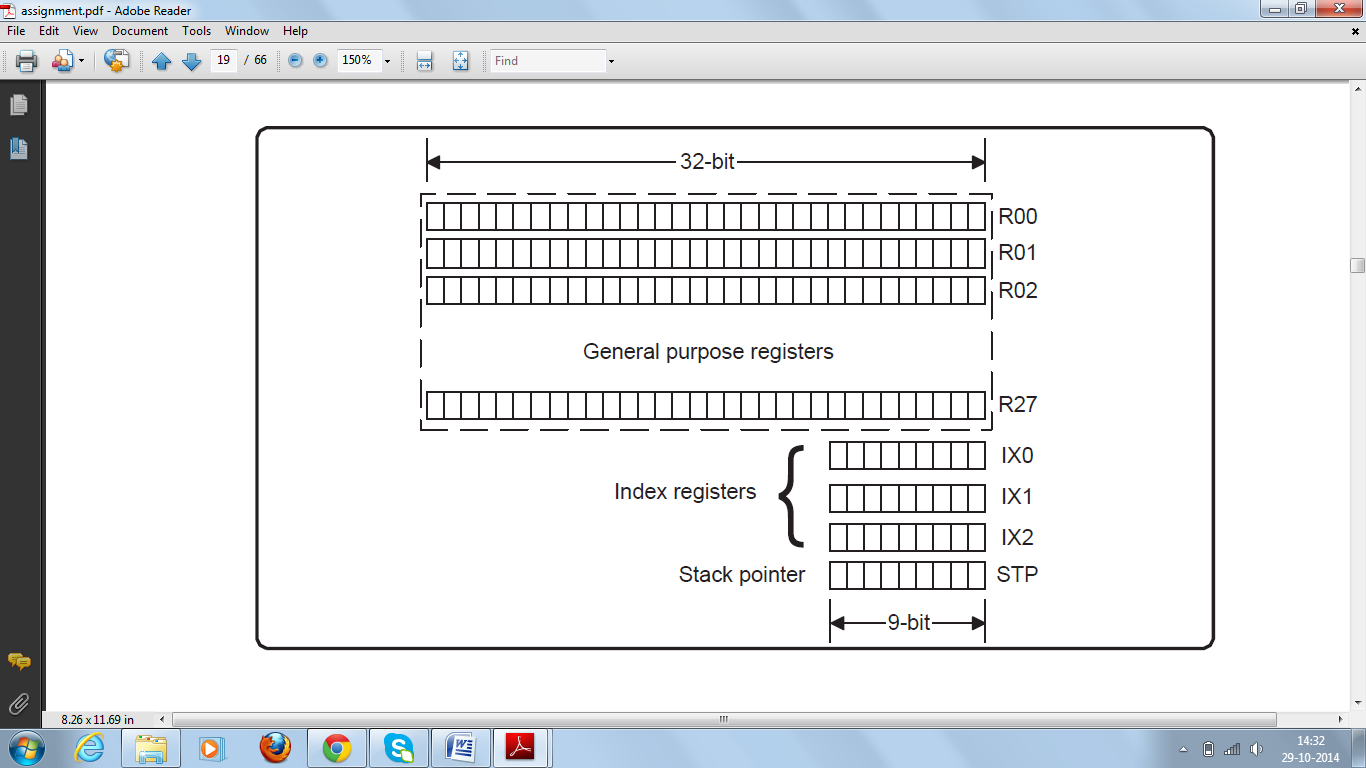
5. Immediate

32 bit microprocessor does not actually support this addressing mode in a direct manner. Since instructions utilizing the immediate addressing mode are made up of at least 2 memory words: 1 for the instruction and 1 for the immediate data. In Micro6, all instruction occupy one memory word. To achieve this objective, immediate addressing mode is effectively replaced by page-0 addressing scheme. That is, all immediate data reside in the topmost page of the memory.

**REGISTER FILES**

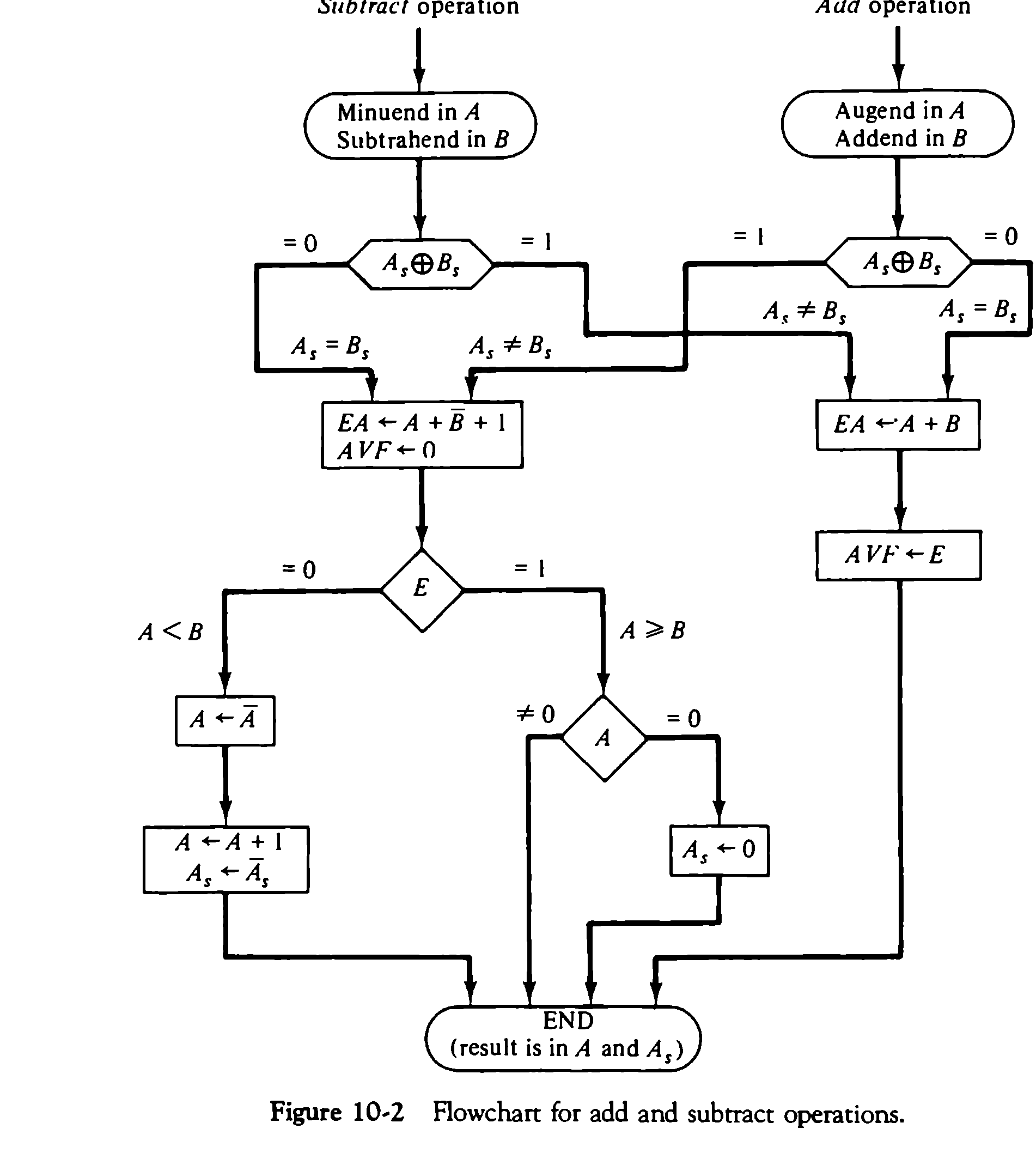
The register file is a small and fast intermediate storage medium, usually implemented inside the CPU. Data in the register file is necessary for the operation of the ALU and the control unit. The size and function(s) of the register file are critical design parameters.

Register file is comprised of 28 general-purpose registers, 3 index-registers and a memory-stack pointer.



**ADDITION AND SUBTRACTION**

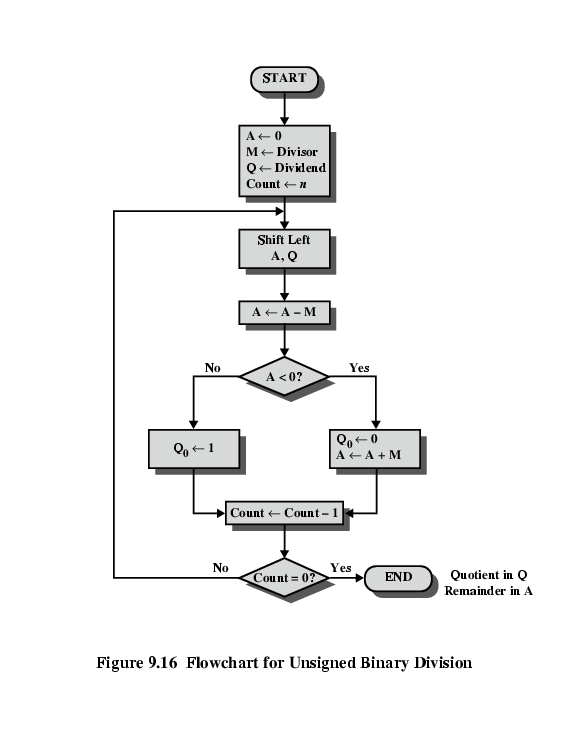
**ALGORITHM**



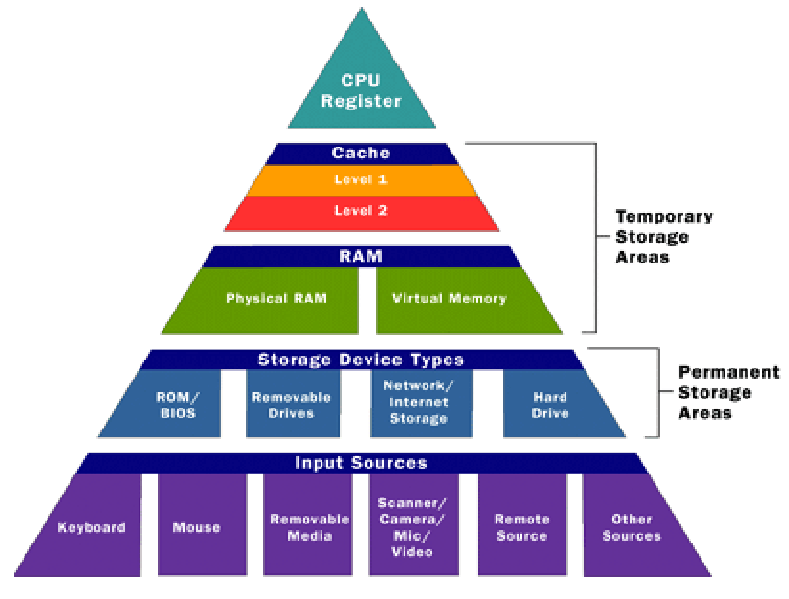
**MULTIPLICATION ALGORITHM**



**DIVISION ALGORITHM**

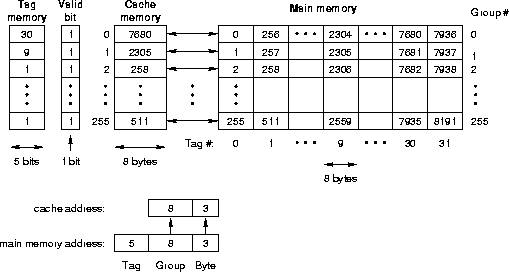


**MEMORY HEIRARCHY**

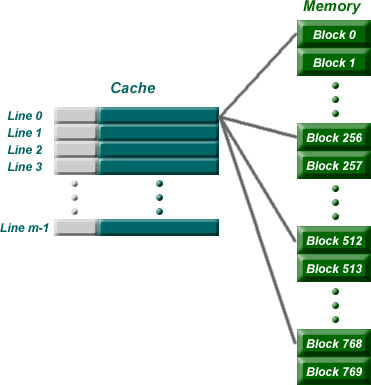


**CACHE MEMORY:**

**DIRECT MEMORY MAPPED:**



***Direct Mapping***

Remember that direct mapping assigned each memory block to a specific line in the cache. If a line is all ready taken up by a memory block when a new block needs to be loaded, the old block is trashed. The figure below shows how multiple blocks are mapped to the same line in the cache. This line is the only line that each of these blocks can be sent to. In the case of this figure, there are 8 bits in the block identification portion of the memory address.

The address for this example is broken down something like the following:

|  |  |  |
| --- | --- | --- |
| Tag | 8 bits identifying line in cache | word id bits |

Once the block is stored in the line of the cache, the tag is copied to the tag location of the line.

***Direct Mapping Summary***

The address is broken into three parts: (s-r) MSB bits represent the tag to be stored in a line of the cache corresponding to the block stored in the line; r bits in the middle identifying which line the block is always stored in; and the w LSB bits identifying each word within the block. This means that:

* The number of addressable units = 2s+w words or bytes
* The block size (cache line width not including tag) = 2w words or bytes
* The number of blocks in main memory = 2s (i.e., all the bits that are not in w)
* The number of lines in cache = m = 2r
* The size of the tag stored in each line of the cache = (s - r) bits

Direct mapping is simple and inexpensive to implement, but if a program accesses 2 blocks that map to the same line repeatedly, the cache begins to thrash back and forth reloading the line over and over again meaning misses are very high.

***Full Associative Mapping***

In full associative, any block can go into any line of the cache. This means that the word id bits are used to identify which word in the block is needed, but the tag becomes all of the remaining bits.

|  |  |
| --- | --- |
| Tag | word id bits |

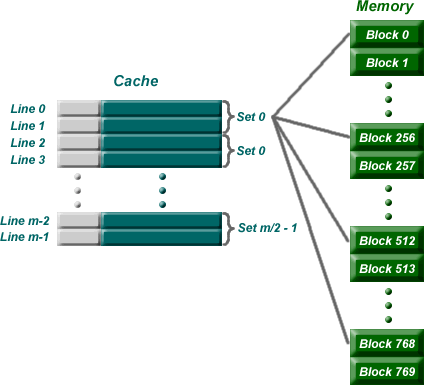
***Full Associative Mapping Summary***

The address is broken into two parts: a tag used to identify which block is stored in which line of the cache (s bits) and a fixed number of LSB bits identifying the word within the block (w bits). This means that:

* The number of addressable units = 2s+w words or bytes
* The block size (cache line width not including tag) = 2w words or bytes
* The number of blocks in main memory = 2s (i.e., all the bits that are not in w)
* The number of lines in cache is not dependent on any part of the memory address
* The size of the tag stored in each line of the cache = s bits

***Set Associative Mapping***

This is the one that you really need to pay attention to because this is the one for the homework. Set associative addresses the problem of possible thrashing in the direct mapping method. It does this by saying that instead of having exactly one line that a block can map to in the cache, we will group a few lines together creating a ***set***. Then a block in memory can map to any one of the lines of a specific set. There is still only one set that the block can map to.



Note that blocks 0, 256, 512, 768, etc. can only be mapped to one set. Within the set, however, they can be mapped associatively to one of two lines.

The memory address is broken down in a similar way to direct mapping except that there is a slightly different number of bits for the tag (s-r) and the set identification (r). It should look something like the following:

|  |  |  |
| --- | --- | --- |
| Tag (s-r bits) | set identifier (r bits) | word id (w bits) |

Now if you have a 24 bit address in ***direct mapping*** with a block size of 4 words (2 bit id) and 1K lines in a cache (10 bit id), the partitioning of the address for the cache would look like this.

Direct Mapping Address Partitions

|  |  |  |
| --- | --- | --- |
| Tag (12 bits) | line identifier (10 bits) | word id (2 bits) |

If we took the exact same system, but converted it to 2-way ***set associative mapping*** (2-way meaning we have 2 lines per set), we'd get the following:

|  |  |  |
| --- | --- | --- |
| Tag (13 bits) | line identifier (9 bits) | word id (2 bits) |

Notice that by making the number of sets equal to half the number of lines (i.e., 2 lines per set), one less bit is needed to identify the set within the cache. This bit is moved to the tag so that the tag can be used to identify the block within the set.

RAM and ROM

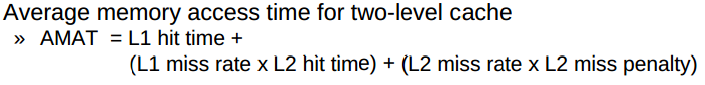
**Read-only memory** or **ROM** is a form of data storage in computers and other electronic devices that can not be easily altered or reprogrammed. RAM is referred to as volatile memory and is lost when the power is turned off whereas ROM in non-volatile and the contents are retained even after the power is switched off.

**Random Access Memory** or **RAM** is a form of data storage that can be accessed randomly at any time, in any order and from any physical location in contrast to other storage devices, such as hard drives, where the physical location of the data determines the time taken to retrieve it. RAM is measured in megabytes and the speed is measured in nanoseconds and RAM chips can read data faster than ROM.

## Different Types of RAM vs ROM

The main types of RAM include static RAM (SRAM), dynamic RAM (DRAM). Static RAM is more expensive and has more capacity for storage than dynamic RAM that has to be refreshed more often, and is thus slower.

ROMs are integrated circuits that contain data and most often cannot be altered. There are some types that can be somewhat modified that include programmable ROM (PROM), erasable programmable ROM (EPROM), electrically erasable programmable ROM (EEPROM) and Flash, which is a type of EEPROM. PROM is a type of ROM that can be programmed only once by a special device and uses high voltages. EPROM can be rewritten using UV radiation, whereas EEPROM can be rewritten electrically and such devices do not require to be removed from the computer. Flash drives are modern version of EEPROM and fastest to erase and rewrite. Some other common types of ROM are CD-ROM, [CD](http://www.diffen.com/difference/CD_vs_Vinyl_Record)-R and CD-RW which is used to store media and music files.



AMAT = hit time + local miss rate X miss penalty

**Programmed I/O**

Programmed I/O (PIO) refers to data transfers initiated by a CPU under driver software control to access registers or memory on a device.

The CPU issues a command then waits for I/O operations to be complete. As the CPU is faster than the I/O module, the problem with programmed I/O is that the CPU has to wait a long time for the I/O module of concern to be ready for either reception or transmission of data. The CPU, while waiting, must repeatedly check the status of the I/O module, and this process is known as Polling. As a result, the level of the performance of the entire system is severely degraded.

Programmed I/O basically works in these ways:

**CPU requests I/O operation**

**I/O module performs operation**

**I/O module sets status bits**

**CPU checks status bits periodically**

**I/O module does not inform CPU directly**

**I/O module does not interrupt CPU**

**CPU may wait or come back later**

**Interrupt Driven I/O:**

The CPU issues commands to the I/O module then proceeds with its normal work until interrupted by I/O device on completion of its work.

For input, the device interrupts the CPU when new data has arrived and is ready to be retrieved by the system processor. The actual actions to perform depend on whether the device uses I/O ports, memory mapping.

For output, the device delivers an interrupt either when it is ready to accept new data or to acknowledge a successful data transfer. Memory-mapped and DMA-capable devices usually generate interrupts to tell the system they are done with the buffer.

Although Interrupt relieves the CPU of having to wait for the devices, but it is still inefficient in data transfer of large amount because the CPU has to transfer the data word by word between I/O module and memory.

Below are the basic operations of Interrupt:

**CPU issues read command**

**I/O module gets data from peripheral whilst CPU does other work**

**I/O module interrupts CPU**

**CPU requests data**

**I/O module transfers data**

**Direct Memory Access (DMA)**

Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module controls exchange of data between main memory and the I/O device. Because of DMA device can transfer data directly to and from memory, rather than using the CPU as an intermediary, and can thus relieve congestion on the bus. CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred.

Direct Memory Access needs a special hardware called DMA controller (DMAC) that manages the data transfers and arbitrates access to the system bus. The controllers are programmed with source and destination pointers (where to read/write the data), counters to track the number of transferred bytes, and settings, which includes I/O and memory types, interrupts and states for the CPU cycles.

DMA increases system concurrency by allowing the CPU to perform tasks while the DMA system transfers data via the system and memory busses. Hardware design is complicated because the DMA controller must be integrated into the system, and the system must allow the DMA controller to be a bus master. Cycle stealing may also be necessary to allow the CPU and DMA controller to share use of the memory bus.

**BG**

**BR**

**CPU**

**RD**

**WR**

**Addr**

**Data**

**Interrupt**

**Random-access**

**memory unit (RAM)**

**RD**

**WR**

**Addr**

**Data**

**BR**

**BG**

**RD**

**WR**

**Addr**

**Data**

**Interrupt**

**DS**

**RS**

**DMA**

**Controller**

**I/O**

**Peripheral**

**device**

**DMA request**

**DMA ack.**

**Read control**

**Write control**

**Data bus**

**Address bus**

**Address**

**select**

DESIGN

OF

CPU

DESIGN OF MEMORY SYSTEM

DESIGN OF I/O SYSTEM

**@Bhansa**