Unit 2: HTML5, JQuery and Ajax
XML Vs JSON

chatgpt explanation :https://chatgpt.com/c/66ed070b-5d74-8011-a0ab-3b47d3490581

XML Vs JSON

JSON, XML, are Text-file formats that can be used to store structured data that can be handy for embedded and Web applications.

XML (Extensible Markup Language) has been around for more than 3 decades now and it is an integral part of every web application. Be it a configuration file, mapping document or a schema definition, XML made life easier for data interchange by giving a clear structure to data and helping in dynamic configuration and loading of variables!

JSON stores all of its data in a map format (key/value pairs) that was neat and easier to comprehend. JSON is said to be slowly replacing XML because of several benefits like ease of data modeling or mapping directly to domain objects, more predictability and easy to understand the structure. JSON is just a data format whereas XML is a markup language.

Structure of XML vs JSON :

XML (Extensible Markup Language)	JSON (JavaScript Object Notation)
01 Adam Cloud computing </technology > Development	{ "employees": [{ "id": "01", "name": "Adam", "technology": "Cloud computing", "title": "Engineer", "team": "Development" } }

Why JSON is Better Than XML:

XML is much more difficult to parse than JSON.
JSON is parsed into a ready-to-use JavaScript object.

Difference between XML and JSON:

XML (Extensible Markup Language)	JSON (JavaScript Object Notation)
XML is a markup language, not a programming language, that has tags to define elements.	JSON is just a format written in JavaScript.
XML data is stored as a tree structure. Example – 2001 Varsha 2002 Akash	Data is stored like a map with key value pairs. Example – {“employees”: [{"id":"2001", “name":"Varsha"}, {"id":"2002", "name":"Akash"}]}

The biggest difference is:

XML has to be parsed with an XML parser. JSON can be parsed by a standard JavaScript function.

	JavaScript has a built in function for converting JSON strings into JavaScript objects: JSON.parse() JavaScript also has a built in function for converting an object into a JSON string: JSON.stringify()
Can perform processing and formatting documents and objects.	It does not do any processing or computation
Bulky and slow in parsing, leading to slower data transmission	Very fast as the size of file is considerably small, faster parsing by the JavaScript engine and hence faster transfer of data
Supports namespaces, comments and metadata	There is no provision for namespace, adding comments or writing metadata
Document size is bulky and with big files, the tag structure makes it huge and complex to read.	Compact and easy to read, no redundant or empty tags or data, making the file look simple.
Doesn't support array directly. To be able to use array, one has to add tags for each item. science maths computers	Supports array which can be accessed as – x = student.subjects[i]; **where** "subjects" **is** an array **as** – "subjects": ["science", "math", "computers"]
Supports many complex data types including charts, images and other non-primitive data types.	JSON supports only strings, numbers, arrays Boolean and object. Even object can only contain primitive types.
XML supports UTF-8 and UTF-16 encodings.	JSON supports UTF as well as ASCII encodings.

XML structures are prone to some attacks as external entity expansion and DTD validation are enabled by default. When these are disabled, XML parsers are safer.	JSON parsing is safe almost all the time except if JSONP is used, which can lead to Cross-Site Request Forgery (CSRF) attack.
Though the X is AJAX stands for XML, because of the tags in XML, a lot of bandwidth is unnecessarily consumed, making AJAX requests slow.	As data is serially processed in JSON, using it with AJAX ensures faster processing and hence preferable. Data can be easily manipulated using eval() method.

XML Parser

The XML DOM (Document Object Model) defines the properties and methods for accessing and editing XML.

XML parser is a software library or a package that provides interface for client applications to work with XML documents. It checks for proper format of the XML document and may also validate the XML documents.

XMLSerializer

The XMLSerializer interface provides the serializeToString() method to construct an XML string representing a DOM tree.

Eg –

```
var s = new XMLSerializer();
 var d = document;
 var str = s.serializeToString(d);
 saveXML(str);
```

Example

```
var text =
"<studentdetails><student>"+"<name>John</name>"+"<subject>Web</subj
ect>"+"</studentdetails></student>";

//converts a DOM string to XML DOM structure

var parser = new DOMParser();

var xmldoc = parser.parseFromString(text,"text/xml");

document.getElementById("demo").innerHTML =

xmldoc.getElementsByTagName("name")[0].childNodes[0].nodeValue;
```

JSON

- JSON: JavaScript Object Notation.
- JSON is a syntax for storing and exchanging data.
- JSON is text, written with JavaScript object notation.
- JSON is a lightweight data-interchange format
- JSON is "self-describing" and easy to understand
- JSON is language independent

JSON uses JavaScript syntax, but the JSON format is text only. Text can be read and used as a data format by any programming language.

JSON NOTATION

JSON data is written as name/value pairs.

A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value. The values can be a string, number, an object(JSON object),an array, a Boolean, null.

Eg-

{ "name": "John" } in JSON and

{ name: "John" } in JavaScript

If you have data stored in a JavaScript object, you can convert the object into JSON, and send it to a server:

```
var obj = { name: "John", age: 30, city: "New York" };

var myJSON = JSON.stringify(obj);

document.getElementById("demo").innerHTML = myJSON;
```

If you receive data in JSON format, you can convert it into a JavaScript object:

```
var myJSON = '{ "name":"John", "age":31, "city":"New York" }';

var myObj = JSON.parse(myJSON);

document.getElementById("demo").innerHTML = myObj.name;
```

JSON vs XML formats

Eg- employee details

```
{"employees":[
  { "firstName":"John", "lastName":"Doe" },
  { "firstName":"Anna", "lastName":"Smith" },
  { "firstName":"Peter", "lastName":"Jones" }
]}
```


```
<employees>
  <employee>
    <firstName>John</firstName> <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName> <lastName>Smith</lastName>
  </employee>
  <employee>
    <firstName>Peter</firstName> <lastName>Jones</lastName>
  </employee>
</employees>
```

Unit 2: HTML5, JQuery and Ajax

HTML5 <audio> tag

It is used to play audio in html pages. It takes the below basic format in its simplest form:

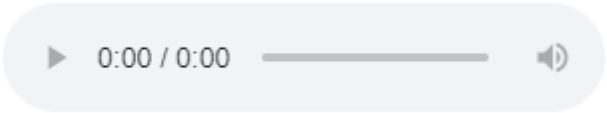
<audio src="my_music.mp3" controls></audio>

With the above structure, when the html page loads the page requests for the audio file listed in the "src" attribute and the “controls” attribute displays the browser default audio player for controlling playback.

CODE:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Intro to Audio</title>
</head>
<body>
    <audio src="sample.mp3" controls></audio>
</body>
</html>
```

Results:



The <audio> tag comes with many inline global attributes that help in modifying its behaviour. Some of the attributes are:

1. autoplay
2. buffered

3. controls
4. loop
5. muted
6. played
7. preload
8. src
9. Volume

Some of the attributes, values and description are given below:

Attribute	Value	Description
autoplay	autoplay	Specifies that the audio will start playing as soon as it is ready.
controls	controls	Specifies that controls will be displayed, such as a play button.
loop	loop	Specifies that the audio will start playing again (looping) when it reaches the end
preload	preload	Specifies that the audio will be loaded at page load, and ready to run. Ignored if autoplay is present.
src	*url*	Specifies the URL of the audio to play

HTML 5 <video> Tag

The HTML 5 <video> tag is used to specify video on an HTML document. For example, it can be embed in a music video on web page for visitors to listen to and watch. The <video> tag was introduced in HTML 5.

The HTML 5 <video> tag accepts attributes that specify how the video should be played. Attributes include preload, autoplay, loop and more.

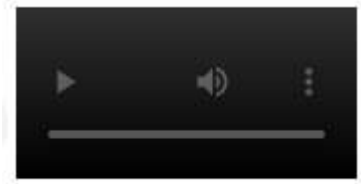
Any content between the opening and closing <video> tags is fallback content. This content is displayed only by browsers that don't support the <video> tag.

Attribute	Value	Description
audio	muted	Defining the default state of the the audio. Currently, only "muted" is allowed
autoplay	autoplay	If present, then the video will start playing as soon as it is ready
controls	controls	If present, controls will be displayed, such as a play button
height	*pixels*	Sets the height of the video player
loop	loop	If present, the video will start over again, every time it is finished
poster	*url*	Specifies the URL of an image representing the video
preload	preload	If present, the video will be loaded at page load, and ready to run. Ignored if "autoplay" is present
src	*url*	The URL of the video to play
width	*pixels*	Sets the width of the video player

CODE:

```
<video src="/video/pass-countdown.ogg" width="170" height="85" controls>
<p>If you are reading this, it is because your browser does not support the HTML5 video element.</p>
</video>
```

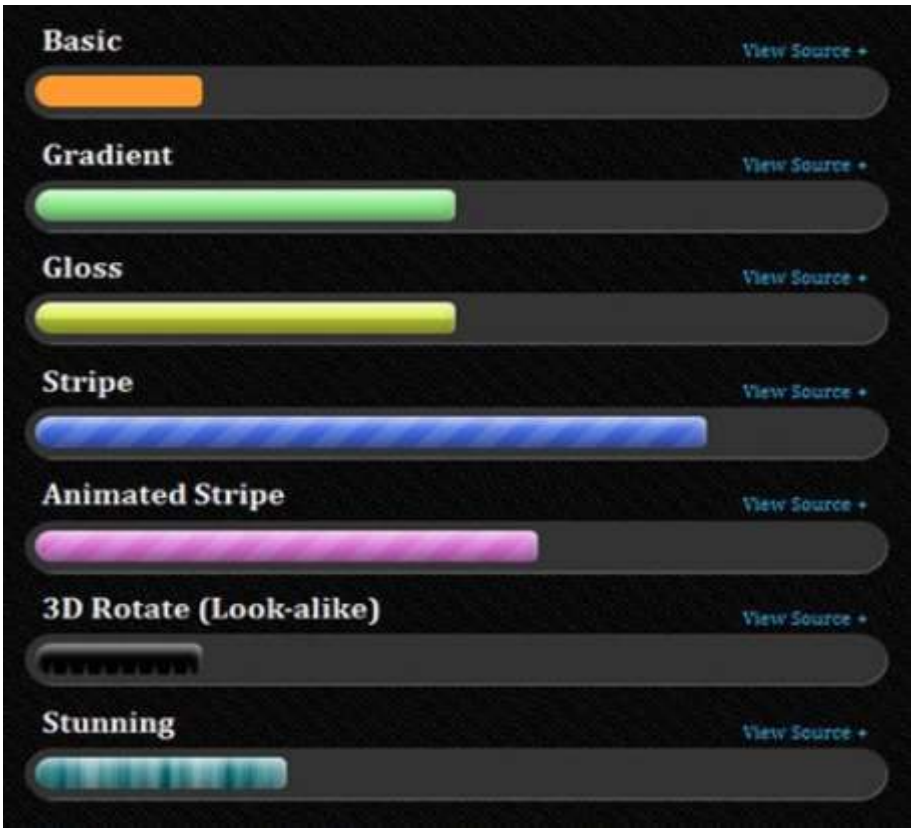
Results:



HTML 5 <progress> Tag

The <progress> element is used to create a progress bar to serve as a visual demonstration of progress towards the completion of task or goal. The max and value attributes are used to define how much progress (value) has been made towards task completion (max).

It can be indeterminate progress bar, which can be either in the form of spinning wheel or a horizontal bar. In this mode, the bar only shows cyclic movements and do not provide the exact progress indication. This mode is usually used at the time when the length of the time is not known.



CODE:

```
<progress value="33" max="100"></progress>
```

Results:

Reference Link for styling progress elements:

https://css-tricks.com/html5-progress-element/

Unit 2: HTML5, JQuery and Ajax

HTML 5 <canvas> Tag

The Canvas API provides a means for drawing graphics via JavaScript and the HTML <canvas> element. It can be used for animation, game graphics, data visualization, photo manipulation, and real-time video processing. Canvas allows you to render graphics powered by JavaScript. Some of the Canvas context methods are following:

Method	Description
fillRect(x, y, width, height)	Draws a filled rectangle
strokeRect(x, y, width, height)	Draws a rectangular outline
clearRect(x, y, width, height)	Clears the specified rectangular area, making it fully transparent
moveTo(x, y)	Moves the pen to the coordinates specified by x and y
lineTo(x, y)	Draws a line from the current drawing position to the position specified by x and y
arc(x, y, r, sAngle, eAngle, anticlockwise)	Draws an arc centered at (x, y) with radius r starting at sAngle and ending at eAngle going anticlockwise (defaulting to clockwise).
arcTo(x1, y1, x2, y2, radius)	Draws an arc with the given control points and radius, connected to the previous point by a straight line

CODE:

```
<p>Before canvas.</p>
<canvas width="120" height="60"></canvas>
<p>After canvas.</p>
<script>
  let canvas = document.querySelector("canvas");
  let context = canvas.getContext("2d");
  context.fillStyle = "red";
  context.fillRect(10, 10, 100, 50);
</script>
```

Results:

Before canvas.

After canvas.

HTML <svg>Tag

The svg element is a container that defines a new coordinate system and viewport. It is used as the outermost element of SVG documents, but it can also be used to embed a SVG fragment inside an SVG or HTML document.

The xmlns attribute changes an element (and its children) to a different XML namespace. This namespace, identified by a URL, specifies the dialect that we are currently speaking. The <circle> and <rect> tags, which do not exist in HTML, do have a meaning in SVG—they draw shapes using the style and position specified by their attributes.

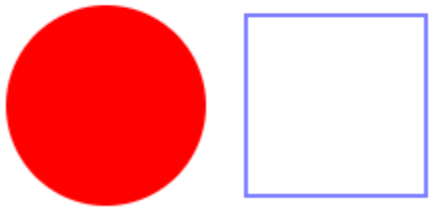
These tags create DOM elements, just like HTML tags, that scripts can interact with.

CODE:

```
<p>Normal HTML here.</p>
<svg xmlns="http://www.w3.org/2000/svg">
  <circle r="50" cx="50" cy="50" fill="red"/>
  <rect x="120" y="5" width="90" height="90"
        stroke="blue" fill="none"/>
</svg>
```

Results:

Normal HTML here.



Unit 2: HTML5, JQuery and Ajax

HTML5 Geolocation API:

The Geolocation API of HTML5 helps in identifying the user's location, which can be used to provide location specific information or route navigation details to the use. There are many techniques used to identify the location of the user. The Geolocation API protects the user's privacy by mandating that the user permission should be sought and obtained before sending the location information of the user to any website. So the user will be prompted with a popover or dialog requesting for the user's permission to share the location information. The user can accept or deny the request.

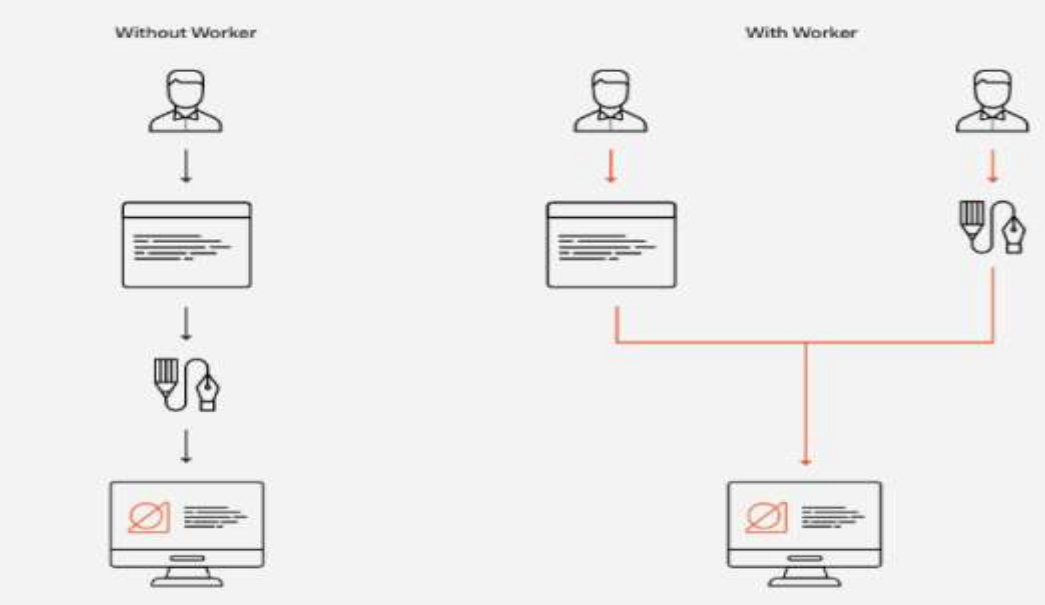
The current location of the user can be obtained using the getCurrentPosition function of the navigator.geolocation object. This function accepts three parameters – Success callback function, Error callback function and position options. If the location data is fetched successfully, the success callback function will be invoked with the obtained position object as its input parameter. Otherwise, the error callback function will be invoked with the error object as its input parameter.

Web Workers

Web Workers are a simple means for web content to run scripts in background threads.The worker thread can perform tasks without interfering with the user interface. In addition, they can perform I/O using XMLHttpRequest(although the responseXML and channel attributes are always null).Once created, a worker can send messages to the JavaScript code that created it by posting messages to an event handler specified by that code (and vice versa).

Using web workers in HTML5 allows you to prevent the execution of bigger tasks from freezing up your web page. A web worker performs the job in the

background, independent of other scripts and thus not affecting their performance. The process is also called threading, i.e., separating the tasks into multiple parallel threads. During the time, the user can browse normally, as the page stays fully responsive.



TYPES OF WEB WORKERS

Dedicated Workers

- Dedicated Web Workers are instantiated by the main process and can only communicate with it.
- A dedicated worker is only accessible by the script that called it.

Shared Workers

- Shared workers can be reached by all processes running on the same origin (different browser tabs, iframes or other shared workers).
- A shared worker is accessible by multiple scripts, similar to the basic

dedicated worker, except that it has two functions available handled by different script files

Workers are created and executed in one of the main program files with their code housed in a separate file. A worker is built using the Worker constructor method, which takes a parameter of worker.js, the JavaScript file storing the worker code.

Using .postMessage(), the parent thread can communicate messages to its workers. .postMessage() is a cross-origin API the can transmit primitive data and JSON structures, but not functions.

The parent code may also have a callback function that listens for a response from the worker confirming its work is complete in order for it to enact another action. As in the example below, the callback function will contain a target, which identifies the worker ('message') , and data, or the message posted by the worker.

This is main.js.

```
let worker = new Worker('worker.js');

worker.postMessage("Hello World");

worker.addEventListener('message', function(e) {
  console.log('Worker said: ', e.data);
}, false);
```

This is worker.js

```
self.addEventListener('message', function(e) {
  self.postMessage(e.data);
}, false);
```

Meanwhile, the worker, living in its own file, stands by with an eventListener waiting to be called. When it receives the message event from the parent code, it likewise communicates its response via the postMessage method.

HTML5 File API (Other Features in HTML5 – for reference only)

The HTML5 file API enables JavaScript inside HTML5 pages to load and process files from the local file system. Via the HTML5 file API it is possible for JavaScript to process a file locally, e.g. compress, encode or encrypt it, or upload the file in smaller chunks. Of course the HTML5 file API raises some security concerns.

Core Objects of HTML5 File API

The HTML5 file API contains the following core objects:

- FileList
- File
- Blob
- FileReader

The File object represents a file in the local file system.

The FileList object represents a list of files in the local file system. For instance, a list of files inside a directory.

The Blob object represents a Binary Large OBject (BLOB) which is used to hold the contents of a single file from the local file system.

Unit 2: HTML5, JQuery and Ajax

jQuery

jQuery is a JavaScript Library. jQuery simplifies JavaScript programming. jQuery is a lightweight, "write less, do more", and JavaScript library. The purpose of jQuery is to make it much easier to use JavaScript on website. jQuery was originally released in January 2006 at BarCampNYC byJohn Resig.

jQuery is a JavaScript library that allows web developers to add extra functionality to their websites. It is open source and provided for free under the MIT license. In recent years, jQuery has become the most popular JavaScript library used in web development.

With jQuery you select (query) HTML elements and perform "actions" on them. Basic syntax is:

$(selector).action ()

- A $ sign to define/ access jQuery.
- “$()” access an element in current html document.

A (selector) to "query (or find)" HTML elements.

A jQuery action () to be performed on the element(s).

Examples:

- $("p").hide() - hides all elements.
- $("#test").hide() - hides the element with id="test".
- $(".test").hide() - hides all elements with class="test".

JavaScript vs. jQuery

Let us try to understand the difference between JavaScript and jquery.

Example 1 - Hide an element with id "textbox“

```
//JavaScript
 document.getElementById(textbox).style.display = "none";
//jQuery
$(#textbox).hide();
```

Example 2 - Create a <h1> tag with "my text“

```
//JavaScript
 var h1 = document.CreateElement("h1");
 h1.innerHTML = "my text";
document.getElementsByTagName(body)[0].appendChild(h1);
//jQuery
$(body).append( $("<h1/>").html("my text") ;
```

The Document Ready Event:

```
$(document).ready(function(){
// jQuery methods go here...
}
```

This is to prevent any jQuery code from running before the document is finished loading (is ready).

- Trying to hide an element that is not created yet.
- Trying to get the size of an image that is not loaded yet

Alternate Syntax :

```
$(function(){

// jQuery methods go here...

}
```

jquery Selectors:

jQuery selectors allow you to select and manipulate HTML With jQuery selectors you can find elements based on their id, classes, types, attributes, values of attributes and much more. It's based on the existing CSS Selectors and in addition, it has some own custom selectors. All type of selectors in jQuery, start with the dollar sign and parentheses: $().

Types of jquery selectors .

- Element selector Id
- (#) selector Class
- (.) selector

Element Selector:

The jQuery element selector selects elements based on their tag names. Example:

```
$(document).ready(function()
{
    $("#button").click(function()
   {
       $("p").hide();
    });
});
```

Id (#) Selector :

The jQuery #id selector uses the id attribute of an HTML tag to find the specific element.

Example :

```
$(document).ready(function()
{
        $("#button").click(function()
        {
            $("#test").hide();
        });
});
```

Class (.) Selector:

The jQuery class selector finds elements with a specific class.

Example :

```
$(document).ready(function()
{
        $("#button").click(function()
        {
            $(".test").hide();
        });
});
```

More jquery Selectors :

Syntax	Description
$("*")	Selects all elements
$(this)	Selects the current HTML element
$("p.intro")	Selects all <p> elements with class="intro"
$("p:first")	Selects the first <p> element
$("ul li:first")	Selects the first element of the first
$("ul li:first-child")	Selects the first element of every
$("[href]")	Selects all elements with an href attribute
$("a[target='_blank']")	Selects all <a> elements with a target attribute value equal to "_blank"
$("tr:even")	Selects all even <tr> elements
$("tr:odd")	Selects all odd <tr> elements

jquery Effects :

There are 3 types of jQuery Effects and they are:

- jQuery hide()
- jQuery show()
- jQuery toggle()

Syntax:

$(selector).hide(speed, callback);

$(selector)show(speed, callback);

$(selector).toggle(speed, callback);

Speed and callback are optional parameters

jQueryEvent Methods:

Mouse Events	Keyboard Events	Form Events	Document/Window Events
click	keypress	submit	load
dblclick	keydown	change	resize
mouseenter	keyup	focus	scroll
mouseleave		blur	unload

jQuery Terminology:

- The **jQuery function** refers to the global jQuery object or the $ function depending on the context .
- A **jQuery object** the object returned by the jQuery function that often represents a group of elements.
- **Selected elements** refers to the DOM elements that you have selected for, most likely by some CSS selector passed to the jQuery function and possibly later filtered further.

jQuery Methods

1. DOM Manipulation

- before(), after(), append(), appendTo()

Example: Move all paragraphs in div with id "contents"

```
$("p").appendTo("#contents");
        $("h1").append(" Dom Manipulation");
       <body>
        <h1>jQuery Dom Manipulation</h1>
       <div id="contents">
              <p>jQuery is good</p>
               <p>jQuery is better</p>
              <p>jQuery is the best</p>
        </div> </body>
```

2. Attributes

- css(), addClass(), attr(), html(), val()

Example : Setting

```
$("img.logo").attr("align", "left");
$("p.copyright").html("&copy; 2009 ajaxray");
$("input#name").val("Spiderman");
```

Example : Getting

```
var allignment = $("img.logo").attr("align");
var copyright = $("p.copyright").html();
var username = $("input#name").val();
```

3. Events

- click(), bind(), unbind(), live()

Example: Binding all interactions on events.

```
$(document).ready(function(){
$("#message").click(function(){
        $(this).hide(); })
});
<span id="message" onclick="…"> blah blah </span>
```

4. Effects

- hide(), fadeOut(), toggle(), animate()

Example :When "show-cart" link clicked, slide up/down "cart" div.

```
$("a#show-cart").click(function(){
$("#cart").slideToggle("slow"); })
```

5. Ajax

- load(), get(), ajax(), getJSON()

Examples: Load a page in a container

$("#comments").load ("/get_comments.php"); $("#comments").load ("/get_comments.php", {max: 5});

jQuery Method Chaining

Chaining Methods, also known as Cascading, refers to repeatedly calling one method after another on an object, in one continuous line of code. This technique abounds in jQuery and other JavaScript libraries and it is even common in some JavaScript native methods.

Example:

```
$("#wrapper").fadeOut().html("Welcome, Sir").fadeIn();
or this:
str.replace("k", "R").toUpperCase().substr(0,4);
```

is not just pleasurable and convenient but also succinct and intelligible. It allows us to read code like a sentence, flowing gracefully across the page. It also frees us from the monotonous, blocky structures we usually construct.

jQuery chaining allows you to execute multiple methods in a single statement. By doing that, it removes the need for repeatedly finding the same element to execute code. It also makes the code more compact and readable.

To perform jQuery method chaining, you should append actions to one another. Usually each statement is run as a separate operation. Chaining in jQuery is used to link multiple statements together. A chained jQuery statement is executed as one operation. Therefore, it runs faster.

Unit 2: HTML5, JQuery and Ajax

jQuery

jQuery is a JavaScript Library. jQuery simplifies JavaScript programming. jQuery is a lightweight, "write less, do more", and JavaScript library. The purpose of jQuery is to make it much easier to use JavaScript on website. jQuery was originally released in January 2006 at BarCampNYC byJohn Resig.

jQuery is a JavaScript library that allows web developers to add extra functionality to their websites. It is open source and provided for free under the MIT license. In recent years, jQuery has become the most popular JavaScript library used in web development.

With jQuery you select (query) HTML elements and perform "actions" on them. Basic syntax is:

$(selector).action ()

- A $ sign to define/ access jQuery.
- “$()” access an element in current html document.

A (selector) to "query (or find)" HTML elements.

A jQuery action () to be performed on the element(s).

Examples:

- $("p").hide() - hides all elements.
- $("#test").hide() - hides the element with id="test".
- $(".test").hide() - hides all elements with class="test".

JavaScript vs. jQuery

Let us try to understand the difference between JavaScript and jquery.

Example 1 - Hide an element with id "textbox“

```
//JavaScript
 document.getElementById(textbox).style.display = "none";
//jQuery
$(#textbox).hide();
```

Example 2 - Create a <h1> tag with "my text“

```
//JavaScript
 var h1 = document.CreateElement("h1");
 h1.innerHTML = "my text";
document.getElementsByTagName(body)[0].appendChild(h1);
//jQuery
$(body).append( $("<h1/>").html("my text") ;
```

The Document Ready Event:

```
$(document).ready(function(){
// jQuery methods go here...
}
```

This is to prevent any jQuery code from running before the document is finished loading (is ready).

- Trying to hide an element that is not created yet.
- Trying to get the size of an image that is not loaded yet

Alternate Syntax :

```
$(function(){

// jQuery methods go here...

}
```

jquery Selectors:

jQuery selectors allow you to select and manipulate HTML With jQuery selectors you can find elements based on their id, classes, types, attributes, values of attributes and much more. It's based on the existing CSS Selectors and in addition, it has some own custom selectors. All type of selectors in jQuery, start with the dollar sign and parentheses: $().

Types of jquery selectors .

- Element selector Id
- (#) selector Class
- (.) selector

Element Selector:

The jQuery element selector selects elements based on their tag names. Example:

```
$(document).ready(function()
{
    $("#button").click(function()
   {
       $("p").hide();
    });
});
```

Id (#) Selector :

The jQuery #id selector uses the id attribute of an HTML tag to find the specific element.

Example :

```
$(document).ready(function()
{
        $("#button").click(function()
        {
            $("#test").hide();
        });
});
```

Class (.) Selector:

The jQuery class selector finds elements with a specific class.

Example :

```
$(document).ready(function()
{
        $("#button").click(function()
        {
            $(".test").hide();
        });
});
```

More jquery Selectors :

Syntax	Description
$("*")	Selects all elements
$(this)	Selects the current HTML element
$("p.intro")	Selects all <p> elements with class="intro"
$("p:first")	Selects the first <p> element
$("ul li:first")	Selects the first element of the first
$("ul li:first-child")	Selects the first element of every
$("[href]")	Selects all elements with an href attribute
$("a[target='_blank']")	Selects all <a> elements with a target attribute value equal to "_blank"
$("tr:even")	Selects all even <tr> elements
$("tr:odd")	Selects all odd <tr> elements

jquery Effects :

There are 3 types of jQuery Effects and they are:

- jQuery hide()
- jQuery show()
- jQuery toggle()

Syntax:

$(selector).hide(speed, callback);

$(selector)show(speed, callback);

$(selector).toggle(speed, callback);

Speed and callback are optional parameters

jQueryEvent Methods:

Mouse Events	Keyboard Events	Form Events	Document/Window Events
click	keypress	submit	load
dblclick	keydown	change	resize
mouseenter	keyup	focus	scroll
mouseleave		blur	unload

jQuery Terminology:

- The **jQuery function** refers to the global jQuery object or the $ function depending on the context .
- A **jQuery object** the object returned by the jQuery function that often represents a group of elements.
- **Selected elements** refers to the DOM elements that you have selected for, most likely by some CSS selector passed to the jQuery function and possibly later filtered further.

jQuery Methods

1. DOM Manipulation

- before(), after(), append(), appendTo()

Example: Move all paragraphs in div with id "contents"

```
$("p").appendTo("#contents");
        $("h1").append(" Dom Manipulation");
       <body>
        <h1>jQuery Dom Manipulation</h1>
       <div id="contents">
              <p>jQuery is good</p>
               <p>jQuery is better</p>
              <p>jQuery is the best</p>
        </div> </body>
```

2. Attributes

- css(), addClass(), attr(), html(), val()

Example : Setting

```
$("img.logo").attr("align", "left");
$("p.copyright").html("&copy; 2009 ajaxray");
$("input#name").val("Spiderman");
```

Example : Getting

```
var allignment = $("img.logo").attr("align");
var copyright = $("p.copyright").html();
var username = $("input#name").val();
```

3. Events

- click(), bind(), unbind(), live()

Example: Binding all interactions on events.

```
$(document).ready(function(){
$("#message").click(function(){
        $(this).hide(); })
});
<span id="message" onclick="…"> blah blah </span>
```

4. Effects

- hide(), fadeOut(), toggle(), animate()

Example :When "show-cart" link clicked, slide up/down "cart" div.

```
$("a#show-cart").click(function(){
$("#cart").slideToggle("slow"); })
```

5. Ajax

- load(), get(), ajax(), getJSON()

Examples: Load a page in a container

$("#comments").load ("/get_comments.php"); $("#comments").load ("/get_comments.php", {max: 5});

jQuery Method Chaining

Chaining Methods, also known as Cascading, refers to repeatedly calling one method after another on an object, in one continuous line of code. This technique abounds in jQuery and other JavaScript libraries and it is even common in some JavaScript native methods.

Example:

```
$("#wrapper").fadeOut().html("Welcome, Sir").fadeIn();
or this:
str.replace("k", "R").toUpperCase().substr(0,4);
```

is not just pleasurable and convenient but also succinct and intelligible. It allows us to read code like a sentence, flowing gracefully across the page. It also frees us from the monotonous, blocky structures we usually construct.

jQuery chaining allows you to execute multiple methods in a single statement. By doing that, it removes the need for repeatedly finding the same element to execute code. It also makes the code more compact and readable.

To perform jQuery method chaining, you should append actions to one another. Usually each statement is run as a separate operation. Chaining in jQuery is used to link multiple statements together. A chained jQuery statement is executed as one operation. Therefore, it runs faster.

jquery: Callback

A callback function is executed after the current effect is 100% finished. JavaScript statements are executed line by line. However, with effects, the next line of code can be run even though the effect is not finished. This can create errors. To prevent this, you can create a callback function.

Syntax :

```
$(selector).hide(speed, callback);
```

Example : callback 1 .html

In JavaScript, statement lines are executed one by one. It might cause problems at times, as a certain effect might start running before the previous one finishes.

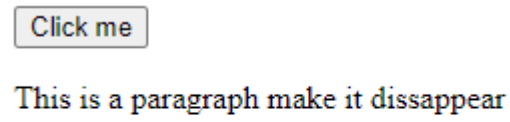
To prevent that, callback function jQuery comes in handy. It creates a queue of effects so they are run in a row.

CODE:

```
<!DOCTYPE html>
<html>
<head>
<script src="https://code.jquery.com/jquery-3.4.1.min.js"
  integrity="sha256-CSXorXvZcTkaix6Yvo6HppcZGetbYMGWSFlBw8HfCJo="
  crossorigin="anonymous"></script>
<script type="text/javascript" src="scripts.js"></script>
<link rel="stylesheet" href="styles.css">
</head>
<body>

<button>Click me</button>

<p>This is a paragraph make it dissappear</p>

</body>
</html>
```

```
$(document).ready(() => {
    $("button").click(() => {
        $("p").hide("slow", () => {
            alert("Congratulations it works");
        });
    });
});
```

Output:



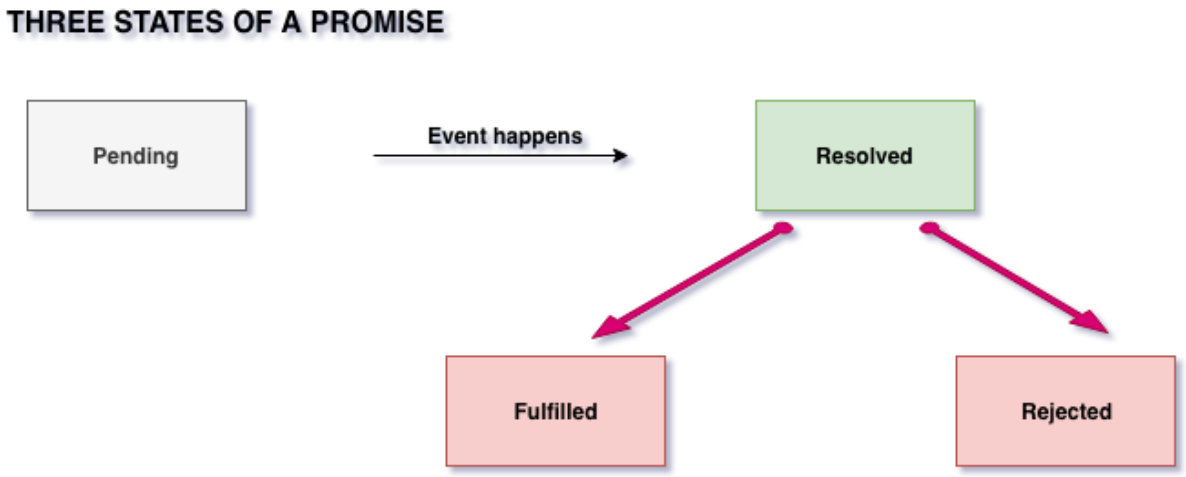
Once you click on Click ME button the texts disappears and alert box pops up:

Callback functions are simple and get the job done, but they become unmanageable as soon as you need to execute many asynchronous operations, either in parallel or in sequence. The situation where you have a lot of nested callbacks, or independent callbacks that have to be synchronized, is often referred to as the “callback hell.”

Because of this challenge, Promises were introduced to simplify deferred activities. A promise is used to handle the asynchronous result of an operation. JavaScript is designed to not wait for an asynchronous block of code to completely execute before other synchronous parts of the code can run.

In jQuery, a promise can be resolved (the promise is successful), rejected (an error occurred), or pending (the promise is neither resolved nor rejected).

A promise is an object that represents the return value or the thrown exception that the function may eventually provide. In other words, a promise represents a value that is not yet known. A promise is an asynchronous value. The core idea behind promises is that a promise represents the result of an asynchronous operation. A Promise has 3 possible states – Pending – Fulfilled – Rejected.



The Promise object is created using the new keyword and contains the promise; this is an executor function which has a resolve and a reject callback. As the names imply, each of these callbacks returns a value with the reject callback returning an error object.

Example: Simple Functional Transform

```
var author = getAuthors();

var authorName = author.name;

// becomes
```

```
var authorPromise = getAuthors().then(function (author) {
 return author.name; });
```

Method syntax: promise

promise([type][, target])

Returns a dynamically generated Promise object that's resolved once all actions of a certain type bound to the collection, queued or not, have finished. By default, type is fx, which means the returned Promise is resolved when all animations of the selected elements have completed.

Parameters

type (String) The type of queue that has to be observed. The default value is fx, which represents the default queue for the effects.

target (Object) The object onto which the promise methods have to be attached.

Returns

A Promise object.

Example:

```
$('p')
  .promise()
  .then(function(value) { console.log(value); });
```

Example:

```
const weather = true ;
const date = new Promise(function(resolve, reject)
```

```
{
if (weather) {
const dateDetails = {
name: 'Cubana Restaurant',
location: '55th Street',
table: 5
};
resolve(dateDetails)
}
else {
reject(new Error('Bad weather'))
}});
date
 .then(function(done)
 {
console.log('We are going on a date!') console.log(done)
 })
.catch(function(error)
 { console.log(error.message)
```

```
})
```

Using a promise that has been created is relatively straightforward; we chain .then() and .catch() to our Promise.

Unit 2: HTML5, JQuery and Ajax

Asynchronous Communication- XHR

AJAX is not a programming language, but a technique that incorporates a client-side script (i.e. a script that runs in a user's browser) that communicates with a web server. Further, its name is somewhat misleading: while an AJAX application might use XML to send data, it could also use just plain text or JSON text. But generally, it uses an XMLHttpRequest object in your browser to request data from the server and JavaScript to display the data.

XMLHttpRequest supports both synchronous and asynchronous communications. In general, however, asynchronous requests should be preferred to synchronous requests for performance reasons.

Synchronous requests block the execution of code which causes "freezing" on the screen and an unresponsive user experience.

Fetch API

The Fetch API provides an interface for fetching resources. It will seem familiar to anyone who has used XMLHTTPRequest, but this API provides a more powerful and flexible feature set.

AJAX can access the server both synchronously and asynchronously:

- **Synchronously**, in which the script stops and waits for the server to send back a reply before continuing.
- **Asynchronously**, in which the script allows the page to continue to be processed and handles the reply if and when it arrives.

Components of AJAX

The AJAX cannot work independently. It is used in combination with other technologies to create interactive Web pages that are described in the following list:

JavaScript:

- Loosely typed scripting language.
- JavaScript function is called when an event occurs in a page.
- Glue for the whole AJAX operation.

DOM:

- API for accessing and manipulating structured documents.
- Represents the structure of XML and HTML documents.

CSS:

- Allows for a clear separation of the presentation style from the content and may be changed programmatically by JavaScript.

XMLHttpRequest:

- JavaScript object that performs asynchronous interaction with the server.

XMLHttpRequest Methods

- **abort()**

  Cancels the current request.

- **getAllResponseHeaders()**

  Returns the complete set of HTTP headers as a string.

- **getResponseHeader(headerName)**

Returns the value of the specified HTTP header.

- **open(method, URL)**
- **open(method, URL, async)**
- **open(method, URL, async, userName)**
- **open(method, URL, async, userName, password)**

  Specifies the method, URL, and other optional attributes of a request.

  The method parameter can have a value of "GET", "POST", or "HEAD". Other HTTP methods such as "PUT" and "DELETE" (primarily used in REST applications) may be possible.

  The "async" parameter specifies whether the request should be handled asynchronously or not. "true" means that the script processing carries on after the send() method without waiting for a response, and "false" means that the script waits for a response before continuing script processing.

- **send(content)**

  Sends the request.

- **setRequestHeader(label, value)**

  Adds a label/value pair to the HTTP header to be sent.

XMLHttpRequest Properties

- **onreadystatechange**

  An event handler for an event that fires at every state change.

- **readyState**

  The readyState property defines the current state of the XMLHttpRequest object.

The following table provides a list of the possible values for the readyState property −

State	Description
0	The request is not initialized.
1	The request has been set up.
2	The request has been sent.
3	The request is in process.
4	The request is completed.

readyState = 0 After you have created the XMLHttpRequest object, but before you have called the open() method.

readyState = 1 After you have called the open() method, but before you have called send().

readyState = 2 After you have called send().

readyState = 3 After the browser has established a communication with the server, but before the server has completed the response.

readyState = 4 After the request has been completed, and the response data has been completely received from the server.

- **responseText**

  Returns the response as a string.

- **responseXML**

  Returns the response as XML. This property returns an XML document object, which can be examined and parsed using the W3C DOM node tree methods and properties.

- **status**

  Returns the status as a number (e.g., 404 for "Not Found" and 200 for "OK").

- **statusText**

  Returns the status as a string (e.g., "Not Found" or "OK").

How to choose between GET & POST?

Purpose of GET - to GET information , intended to be used when you are reading information to display on the page. Browsers will automatically cache the result from a GET request and if the same GET request is made again then they will display the cached result rather than rerunning the entire request. A GET call is retrieving data to display in the page and data is not expected to be changed on the server by such a call and so re-requesting the same data should be expected to obtain the same result.

POST method is intended to be used where you are updating information on the server . Results returned from server . A POST call will therefore always obtain the response from the server rather than keeping a cached copy of the prior response. If the value to be retrieved is expected to vary over time as a result of other processes updating it then add a current time parameter to what you are passing in your GET call. These criteria is not only for GET and POST for your Ajax calls but also to GET or POST when processing forms on your web page as well.

Unit 2: HTML5, JQuery and Ajax

AJAX load() Method

• The load() method loads data from a server and puts the returned data into the selected element.

Syntax: $(selector).load(URL, data, callback);

- The required URL parameter specifies the URL you wish to load.
- The optional data parameter specifies a set of query string key/value pairs to send along with the request.
- The optional callback parameter is the name of a function to be executed after the load() method is completed The following example loads the content of the file "test.txt" into a specific element:

$("#div1").load("test.txt");

It is also possible to add a jQuery selector to the URL parameter. The following example loads the content of the element with id="p1", inside the file "test.txt", into a specific element:

$("#div1").load("test.txt #p1");

The optional callback parameter specifies a callback function to run when the load() method is completed. The callback function can have different parameters:

- responseTxt - contains the resulting content if the call succeed
- statusTXT - contains the status of the call
- xhr - contains the XMLHttpRequest object

The following example displays an alert box after the load() method completes. If the load() method has succeed, it displays "External content loaded successfully!", and if it fails it displays an error message:

```
$("button").click(function(){
$("#div1").load("demo_test.txt",function(responseTxt,statusTxt,xhr){
if(statusTxt=="success

alert("External content loaded successfully!");

 if(statusTxt=="error")

  alert("Error: "+xhr.status+": "+xhr.statusText);  });

});
```

HTTP Request: GET vs. POST

Two commonly used methods for a request-response between a client and server.

- GET- Requests data from a specified resource
- POST - Submits data to be processed to a specified resource

- **$.get(URL,callback);**

  - The required URL parameter specifies the URL you wish to request.
  - The optional callback parameter is the name of a function to be executed if the request succeeds.

- **$.post(URL,data,callback);**

  - The required URL parameter specifies the URL you wish to request.

- The optional data parameter specifies some data to send along with the request.
- The optional callback parameter is the name of a function to be executed if the request succeeds.

Example

```
$("button").click(function(){

$.post("test_post.jsp",

{

name:“Bill Gates",

city:“Seatle"

},

function(data,status)

{

alert("Data: " + data + "\nStatus: " + status);

});

});
```

- The first parameter of $.post() is the URL we wish to request ("test_post.jsp").
- Then we pass in some data to send along with the request (name and city).

- The JSP script in "test_post.jsp" reads the parameters, process them, and return a result.
- The third parameter is a callback function. The first callback parameter holds the content of the page requested, and the second callback parameter holds the status of the request.

The $.ajax(settings) or $.ajax(url, settings)

Used for sending an Ajax request. The settings is an object of key-value pairs. The frequently-used keys are:

- *url*: The request URL, which can be placed outside the *settings* in the latter form.
- *type*: GET or POST.
- *data*: Request parameters (name=value pairs). Can be expressed as an object (e.g., {name:"peter", msg:"hello"}), or query string (e.g., "name=peter&msg=hello").
- *dataType*: Expected response data type, such as text, xml, json, script or html.
- *headers*: an object for request header key-value pairs. The header X-Requested-With:XMLHttpRequest is always added.

Ajax request, by default, is asynchronous. In other words, once the .ajax() is issued, the script will not wait for the response, but continue into the next statement, so as not to lock up and freeze the screen.

NOTE: $ is a shorthand (alias) for the jQuery object. $() is an alias for jQuery() function for Selector. $.ajax() is a global function (similar to class method in an OO language).

The Fetch API

The Fetch API provides a fetch() method defined on the window object, which you can use to perform requests. This method returns a Promise that can be used to retrieve the response of the request.

The fetch method only has one mandatory argument, which is the URL of the resource you wish to fetch. The Fetch API is a modern interface that allows you to make HTTP requests in the web browsers. The fetch() method is available in the global scope that instructs the browser to send a request to a provided URL.

Sending a Request

The fetch() has only one parameter which most of the time is the URL of the resource that you want to fetch:

```
let response = fetch(url);
```

The fetch() method returns a Promise so you can use the then() and catch() methods to handle it:

```
fetch(url)
  .then(response => {
    // handle the response
  })
  .catch(error => {
    // handle the error
  });
```

When the request completes, the resource is available. At this time, the promise will resolve into a Response object.

The Response object is the API wrapper for the fetched resource. The Response object has a number of useful properties and methods to inspect the response.

Reading a Response

If the contents of the response are in the raw text format, you can use the text() method. The text() method returns a Promise that resolves with the complete contents of the fetched resource:

```
fetch('/readme.txt')
   .then(response => response.text())
   .then(data => console.log(data));
```

Or async/await can be used:

```
async function fetchText() {
   let response = await fetch('/readme.txt');
   let data = await response.text();
   console.log(data);
}
```

Besides the text() method, the Response object has other methods such as json(), blob(), formData() and arrayBuffer() to handle respective data.

Handling status codes of a response

The Response object provides the status code and status text via the status and statusText properties. When a request is successful, the status code is 200 and status text is OK:

```
async function fetchText() {
   let response = await fetch('/readme.txt');
```

```
    console.log(response.status); // 200

    console.log(response.statusText); // OK

    if (response.status === 200) {

        let data = await response.text();

        // handle data

    }

}

fetchText();
```

Output:

200

OK

If the requested resource doesn't exist, the response code is 404:

```
let response = await fetch('/non-existence.txt');

console.log(response.status); // 400

console.log(response.statusText); // OK
```

Output:

400

Not Found

If the requested URL throws a server error, the response code will be 500.

If the requested URL is redirected to the new one with the response 300-309, the status of the Response object is set to 200. In addition the redirected property is set to true.

The fetch() returns a promise that rejects when a real failure occurs such as a web browser timeout, a loss of network connection, and a CORS violation.

Unit 2: ReactJS

MERN and REACT.js

Introduction to Web Development

Web development stack is nothing but a set of tools typically used in tandem to develop web apps. It refers to the technologies that individual developer specializes in and use together to develop new pieces of software.

Web technology sets that include all the essential parts of a modern app are as follows: the **frontend framework**, the **backend solution** and the **database (relational or document-orinted)**

Introducton to stack:

Any web application made by using multiple technologies. The combination of these technologies is called a **"stack,"** popularized by the LAMP stack, which is an acronym for Linux, Apache, MySQL, and PHP, which are all open-source components. As the web development world is continually changing, its technology stacks too changing. Top web development stacks are

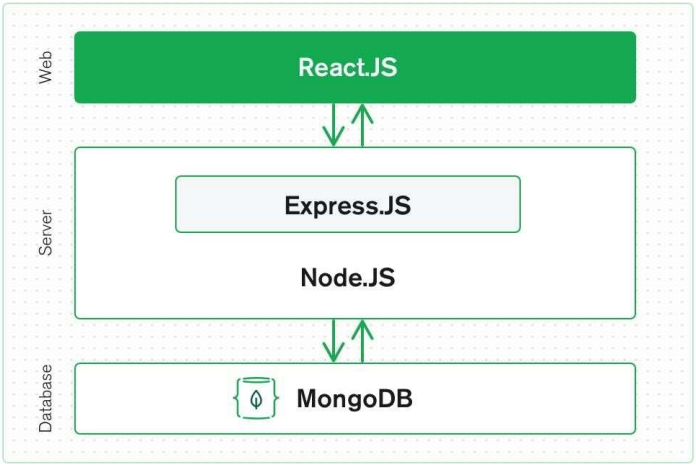
- MEAN
- MERN
- Meteor.js
- Flutter
- The serverless Technology stack
- The LAMP technology stack
- Ruby on Rails Tech Stack

Unit 2: ReactJS

MERN stands for **MongoDB, Express, React, Node**

- MongoDB - Document database
- Express(.js) - Node.js web framework
- React(.js) - A client-side JavaScript library
- Node(.js) - The premier JavaScript web server

Allows you to easily construct a 3-tier architecture (frontend, backend, database) entirely using JavaScript and JSON.



Why MERN?

Ideally suited for web applications that have a large amount of interactivity built into the front-end.

- JavaScript Everywhere
- JSON Everywhere
- Isomorphic

REACT.JS

Free and open source front end JS Library for building UI and UI Components. Maintained by Facebook, a community of individual developers and companies. This can be used as base in the development of Single page applications and mobile applications and

is concerned with state management and rendering the state to DOM.

It is the declarative JavaScript Library for creating dynamic client-side applications

Builds up complex interfaces through simple Components, connect them to data on your backend server, and render them as HTML. Provides support for forms, error handling, events and render them as HTML.

Key points about React.js

- **Properties of React:**

  Declarative, Simple, Component based, Supports server side, Mobile support, Extensive, Fast , Easy to learn

- **Single way data flow**
  - A set of immutable values are passed to the components renderer as properties in its HTML tags. The component cannot directly modify any properties but can pass a call back function with the help of which we can do modifications.
  - This complete process is known as "**properties flow down; actions flow up**".

- **Virtual DOM**
  - Creates an in-memory data structure cache which computes the changes made and then updates the browser.
  - Allows a special feature that enables the programmer to code as if the whole page is rendered on each change whereas react library only renders components that actually change

Creation of React App

Can be done in two ways

- **Using node package manager(npm):** To setup a build environment for React that typically involved use of npm (node package manager), webpack, and Babel

- **Directly importing Reactjs library in HTML Code.** Defined in two .js files (**React** and **ReactDOM**)
  - <script src="https://unpkg.com/react@17/umd/react.development.js" crossorigin ></script>
  - <script src="https://unpkg.com/react-dom@17/umd/react-dom.development.js" crossorigin> </script>
  - The files differ for development and production.
    When deploying, replace "development.js" with "production.min.js"

React is used for handling the **view layer for web and mobile apps.** Allows developers to create **large web applications that can change data**, without reloading the page. Also allows to **create reusable UI components.** The main purpose of React is to **be fast, scalable, and simple. Works only on user interfaces** in the application

React Elements:

The browser DOM is made up of DOM elements. Similarly, the React DOM is made up of React elements. DOM elements and React elements may look the same, but they are actually quite different. A React element is a description of what the actual DOM element should look like. In other words, React elements are the instructions for how the browser DOM should be created. Syntax: React.createElement(type,{props},children);

The first one is the type of element we're creating, in this case an <h1> tag. This could also be another React component. Second is the properties list in the form of objects. Third argument is the content of the element used in the first argument.

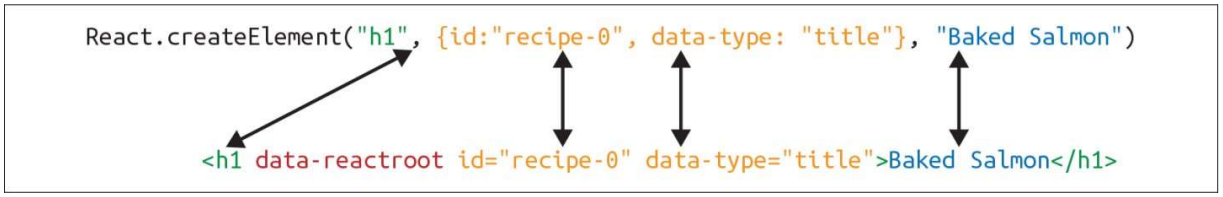
We can create a React element to represent an h1 using React.createElement

```
React.createElement("h1", null, "Baked Salmon")
```

When an element has attributes, they can be described with properties. Here is a sample of an HTML h1 tag that has id and data-type attributes.

```
React.createElement("h1", {id: "recipe-0", 'data-type': "title"}, "Baked Salmon")
```

Relationship between createElement and the DOM element



Note: data-reactroot will always appear as an attribute of the root ele- ment of your React component

React Component in brief:[In detail in L2 and L3]

A user control that has **code to represent visual interfaces and data.** An isolated piece of code which can be reused in one or the other module **.**Contains a root component in which other subcomponents are included. 2 types of components in React.js can be created.

- **Stateless Functional Component**
  - Includes simple JavaScript functions and immutable properties, i.e., the value for properties cannot be changed.

```
function Demo(props) {
  return <h1> Welcome to REACT JS, {props.Name} </h1>;
}
```

- **Stateful Class Component**
  - Classes which extend the Component class from React library. The class component must include the render method which returns HTML.

```
class Demo extends React.Component{
        render(){   return <h1>
                Welcome to REACT JS, {props.Name} </h1>;   }
}
```

Calling/rendering the components:

ReactDOM: Contains the tools necessary to render React elements in the browser. All the tools necessary to generate HTML from the virtual DOM are found in this library.

ReactDOM.render() is responsible for rendering a React component. The first

parameter is the component class name. Second parameter is the destination where the component is to be rendered.

```
ReactDOM.render(
    React.createElement(Demo, null, null),
    document.getElementById('root')
);
```

Coding Example 1: Serverless Hello world: Using react, displaying a simple page on the browser

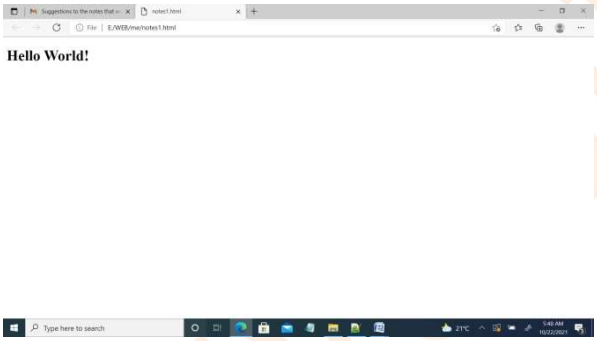
```
<!DOCTYPE HTML>
<html>
<head>
    <script crossdomain
src="https://unpkg.com/react@16/umd/react.development.js"> </script>
    <script crossdomain src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js"> </script>
    <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js">
    </script>
</head>
<body>
  <div id="contents"></div><!-- this is where our component will appear -->
  <script type="text/babel">    // Important
      var contentNode = document.getElementById('contents');
      var component = <h1>Hello World!</h1>; // A simple JSX component
      ReactDOM.render(component, contentNode);
                              // Render the component inside the content Node
  </script>
</body>
</html>
```

In the above code, <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js">

</script> , this is the **babel library which is a JSX transformer**. Then, there is the type of the script that you specify in the wrapping <script> tags around the JSX ode. The browser-based JSX compiler looks for all inline scripts of type “text/babel” and compiles the contents into the corresponding JavaScript. The other two scripts, react.js and react-dom.js, are the core React libraries that handle react component creation and rendering

```
<script type="text/babel">
```

Screenshot of example code 1 output:

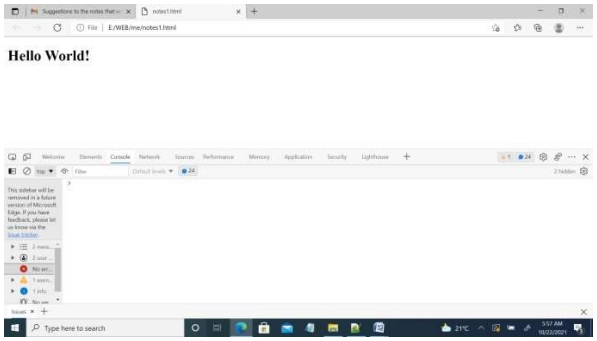


Coding Example 2: Multiple component calls/renders will render only the last one

In the above code, add these below lines and oberve the output

```
ReactDOM.render(component, contentNode);
ReactDOM.render(component, contentNode);
ReactDOM.render(component, contentNode);
ReactDOM.render(component, contentNode);
```

Screenshot of example code 1 output:



Including the JSX Code:

JSX uses a special syntax which allows you to mix HTML with JavaScript. JSX is a JavaScript XML used in React applications. An extension to JavaScript. Uses HTML syntax to create elements and components. Has tag name, attributes, and children. JSX compiles the code into pure JavaScript which can be understood by the browser.

Include the library:

```
<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"> </script>
<script type="text/babel">
                ReactDOM.render(<h1>Welcome to REACTJS</h1>,
                document.getElementById('root')
  );
```

Babel:

A JavaScript compiler that can translate markup or programming languages into JavaScript. Available for different conversions. React uses Babel to convert JSX into JavaScript.

Using React, print welcome to REACTJS on the web page. Use JSX and NonJSX both

Case 1: Using JSX Syntax

```
<html>
<head>
<script crossdomain src="https://unpkg.com/react@16/umd/react.development.js">
</script>
<script crossdomain src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js"> </script>
<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"> </script>
</head>
<body>
        <div id = "root"></div>
        <script type = "text/babel">
```

```
        ReactDOM.render(<h1>welcome</h1>,document.getElementById("root"))
        </script>
</body>
</html>
```

Case 2: Using Non-JSX Syntax: small change in the above code

```
<html>
<head>
<script crossdomain    src="https://unpkg.com/react@16/umd/react.development.js">
</script>
<script crossdomain src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js"> </script>
<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"> </script>
</head>
<body>
        <div id = "root"></div>
        <script type = "text/babel">
        ReactDOM.render(React.createElement("h1","null","welcome to
reactJS"),document.getElementById("root"))
        </script>
</body>
</html>
```

References:

i) What is the use of React.createElement ? - GeeksforGeeks

ii) Textbooks from the syllabus copy

Components and Properties

Components

It defines the **visuals and interactions** that make up what people see in app. Declares **how the view looks like, given the data**. When the data changes, if you are used to the jQuery way of doing things, you'd typically do some DOM manipulation. But the React doesn't do anything like that! The React library figures out how the new view looks, and just applies the changes between the old view and the new view. This makes the views consistent, predictable, easier to maintain, and simpler to understand.

Why Components?

There may be elements which are similar . There might be a need to make a change that will result in changes in multiple places. Similar to functions, if we could write code related to the element in one place, changes can be minimized. Solution is to have reusable piece of JavaScript code that output (via JSX) HTML elements

Types of Components

1. **Stateless Functional Component –** [Will be discussed later]
    - Includes simple JavaScript functions and immutable properties, i.e., the value for properties cannot be changed.
    - Use hooks to achieve functionality for making changes in properties using JS.
    - Used mainly for UI.

2. Stateful Class Component

- Classes which extend the Component class from React library. The class component must include the render method which returns HTML.

First Stateful component

Creation of a component:

```
class HelloWorld extends React.Component
{       render()
        {   return <p>Hello, componentized world!</p>; }
}
```

The render() method is something that that the React calls when it needs to display the component. There are other methods with special meaning to React that can be implemented, called the Lifecycle functions, which provide hooks into various stages of the component formation and events. This will be discussed later. But render() is one that must be present; otherwise you'll have a component that has no screen presence.

Calling/Rendering a component:

Case 1: Add the JSX in the render method with a element with the tag name as the Component name

```
ReactDOM.render( <HelloWorld/>,
        document.querySelector("#container")
);
```

Case 2: Add the NON-JSX in the render method with a element using React.createElement

```
ReactDOM.render( React.createElement("HelloWorld",null, null),
        document.querySelector("#container")
);
```

Parameterized components:

Passing the attributes to the components during rendering and using these attributes as properties inside the component.

Properties

Properties are ways in which React components can be customized. Immutable and same as what attributes in HTML elements. Props are arguments passed into React components and are passed via HTML attributes. 2 steps to add properties to components

- Make the function of your component read the props from the props parameter Place the props inside curly brackets – { }. In JSX, if you want something to get evaluated as an expression, you need to wrap that something inside curly brackets. If you don't do that, the raw text gets printed out.

```
return <h1>Hello, {this.props.greetingtarget}</h1>
```

- **Modify the Component Call** : When rendering the component, add the prop to the component using the attribute

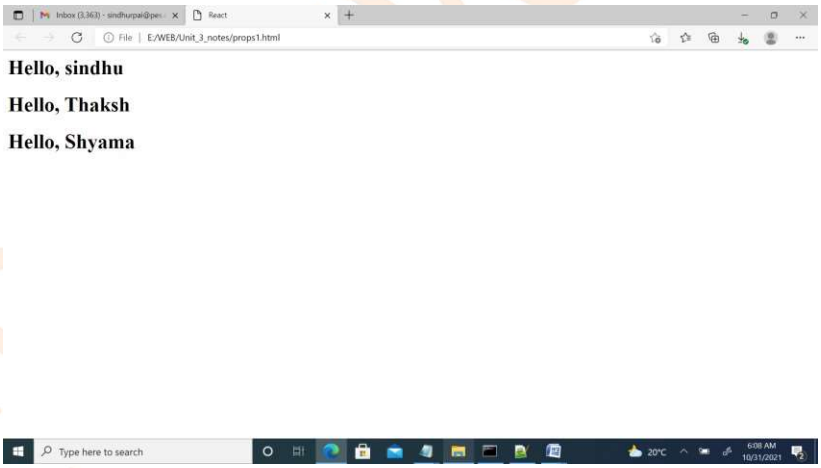
```
<Helloworld greetingtarget = "sindhu" />
```

Coding example 1: Usage of this.props

```
<html>
    <head>
        <title>React</title>
        <script crossdomain
src="https://unpkg.com/react@16/umd/react.development.js"> </script>
        <script crossdomain src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js"> </script>
        <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js">
</script>
    </head>
    <body>
        <div id="container"></div>
        <script type="text/babel">
            var dest = document.querySelector("#container");
            class Helloworld extends React.Component {
                render(){
                    return <h1>Hello, {this.props.greetingtarget}</h1>
                    }
```

```
        }
        ReactDOM.render(
            <div>
                <Helloworld greetingtarget = "sindhu" />
                <Helloworld greetingtarget = "Thaksh" />
                <Helloworld greetingtarget = "Shyama" />
            </div>,
            document.getElementById("container")
        );
    </script>
</body>
</html>
```

Output:



Usage of this.props.children

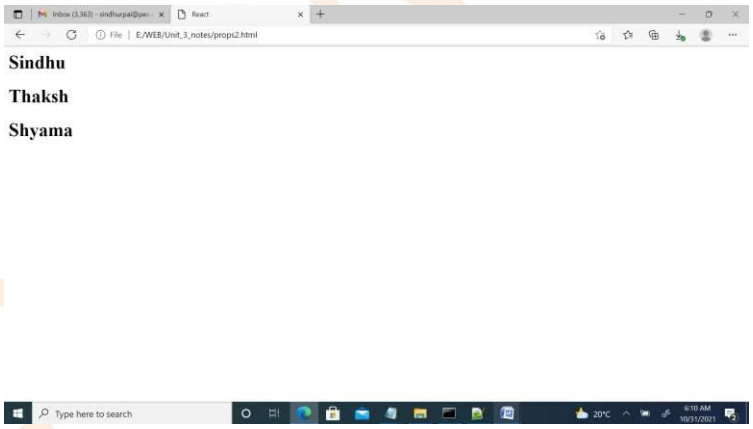
A special property that is passed to components automatically. It is used to display the data between the opening and closing JSX tags when invoking a component. Can have one element, multiple elements, or none at all. It's value will be respectively a single child node, an array of child nodes or undefined

Coding example 2:

```
<script type="text/babel">
```

```
        class Helloworld extends React.Component {
            render(){
                return  <h1>{this.props.children}</h1>
                }
          }
        ReactDOM.render(
            <div>
                <Helloworld>Sindhu</Helloworld>
                <Helloworld>Thaksh</Helloworld>
                <Helloworld>Shyama</Helloworld>
            </div>,
            document.querySelector("#container")
        );
</script>
```

Output:



Validating the property values

The properties being passed to component to can be validated against a specification. This specification is supplied in the form of a static object called propTypes in the class, with the name of the property as the key and the validator as the value, which is one of the many constants exported by React.PropTypes, for example,

React.PropTypes.string

```
Helloworld.propTypes = { greetingtarget: React.PropTypes.string.isRequired,
                         greetingtarget_id: React.PropTypes.number
                       };
```

Property validation is checked only in development mode, and a warning is shown in the console when any validation fails. Since we are in an early stage in the development of the application, we expect more changes to the properties. You can also default the property values when the parent does not supply the value. For example, if you want the greeting_target to be defaulted to something else, rather than show an empty string, you can do this: `Helloworld.defaultProps = { greetingtarget: '-- no title --', };`

Few Points to think:

- Can we have nested components?
- Can we provide style to components?
- How to send properties from one component to another?
- What are the different methods under life cycle of components?
  Refer to the further notes for more details on the above points

UNIT2: ReactJS
Styling the components and Complex Components

Styling the components

There are different ways to style the components in React.

- Inline CSS – using the style attribute
- CSS in JS - Using Libraries JSS and Styled Components
- CSS Modules - css loader and Sass & SCSS
- Stylable

Inline CSS in Detail

Consider the below code to provide red color to the contents of h1. But this is not a react way of providing the styling to the components.

```
render()
{
    return (<h1 style = {color:"red"}>welcome to styling to components</h1>)
}
```

We create objects of style and render it inside the components in style attribute using the React technique. Let us understand this through an example code.

```
<body>
    <div id = "root"></div>
    <script type = "text/babel">
        class Letter extends React.Component{
            render() {
 //Object letterstyle contains all key-value pairs of styling  to be applied to letter
                var letterstyle = {
                marginRight: "10px", // observe this comma
                textAlign: "center",backgroundColor:"blue", //observe the camelcase
                color:"olive",  padding: "20px", display:"inline"      }
                return <h1 style = {letterstyle}> {this.props.children}</h1>
            //Setting the style attribute to refer to the object
```

```
                    }
}
            ReactDOM.render(<div><Letter>A</Letter>
                    <Letter>E</Letter>
                    <Letter>I</Letter>
                    <Letter>O</Letter>
                    <Letter>U</Letter>    </div>
                    ,document.getElementById("root"))
        </script>
    <body>
```

If we execute the above code, All letters will have the same background color. To avoid this, background color must be sent suring the component rendering.

```
<script type = "text/babel">
        class Letter extends React.Component{
                render() {
 //Object letterstyle contains all key-value pairs of styling   to be applied to letter
                    var letterstyle = {
                    marginRight: "10px", // observe this comma
                    textAlign: "center", backgroundColor: this.props.bgcolor,
                    color:"olive",  padding: "20px"        }
                    return <h1 style = {letterstyle}> {this.props.children}</h1>
                    }
        }
        ReactDOM.render(<div><Letter bgcolor = "pink"> A</Letter>
                    <Letter bgcolor = "red">E</Letter>
                    <Letter bgcolor = "#0000FF">I</Letter>
                    <Letter bgcolor = "#BBEE00">O</Letter>
                    <Letter bgcolor = "#EE00BB">U</Letter>        </div>
                    ,document.getElementById("root"))
        </script>
```

Output of above code is as below:

Advantage of creation of style object is , it provides flexibility to add more styling properties as and when required.

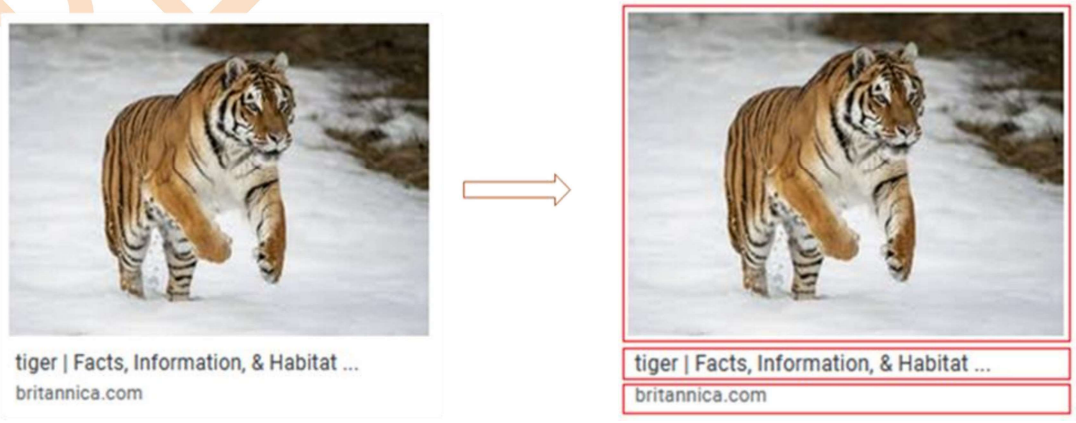
Complex Components

It is nothing but **building the component that uses other user defined components.** Also known as **component composition. React** lets you split the UI into smaller independent pieces so that you can reason about each piece in isolation. Using components rather than building the UI in a monolithic fashion also encourages reuse.. A component takes inputs (called properties) and its output is the rendered UI of the component. We will put together fine-grained components to build a larger UI.

Approach followed to build complex components:

- Identify the major visual elements
- Breaking them into individual components

Consider this example:



DIGITAL DESIGN AND COMPUTER ORGANIZATION

Basic Structure of computers T2: Chapter 1: 1.1-1.4

Department of Computer Science and Engineering

Basic Structure of computers
Outline

- Computer Types
- Functional Units:
  - Input Unit,
  - Memory Unit,
  - ALU,
  - Output Unit,
  - Control Unit,
- Basic operational concepts

DIGITAL DESIGN AND COMPUTER ORGANIZATION

Computer Types T2:Ch1 1.1

Department of Computer Science and Engineering

Basic Structure of computers
Introduction

- **What is Computer Organisation ?**
  - **Computer Organization:** This field focuses on the functional units of a computer system and how they work together to execute instructions. It deals with the internal structure and operation of a computer, including its components like the CPU, memory, I/O devices, and control unit.

- **How is Computer Organisation Different from Digital Design ?**

  While computer organization deals with the high-level structure and function of a computer, digital design focuses on the low-level implementation of its components.

Basic Structure of computers
What is a computer ?

- Simply put, a computer is a sophisticated electronic calculating machine that:
  - Accepts input information,
  - Processes the information according to a list of internally stored instructions and
  - Produces the resulting output information.

- Functions performed by a computer are:
  - Accepting information to be processed as input.
  - Storing a list of instructions to process the information.
  - Processing the information according to the list of instructions.
  - Providing the results of the processing as output.

What are the functional units of a computer?

Basic Structure of computers
Types Of Computers

Classification #1:
- Micro Computers
- Mini Computers
- Mainframes
- Super Computers

Classification #2:
- Analog Computers
- Digital Computers
- Hybrid Computers

More General Classification:
- *General Purpose Computers*
- *Special Purpose Computers*

Types Of Computers
Workstations

- Workstations are *industry* – standard desktop computers with more computational power and high-resolution graphic display with variety of graphic input/output capability.

What is the use of a workstation ?

- Workstation PC's are usually employed in the applications of Computer Aided Design and Drafting (CADD). Simulation & Modeling, Interactive Graphic design, Multimedia and other engineering applications.

Types Of Computers
Enterprise Systems or Mainframes

- Enterprise systems or mainframes are like a family of computers with tremendous computing power beyond workstations.
  - Their computational activities are distributed among one main system (usually a server) and number of child nodes or intelligent PC's with or without local computing power / processor (usually client computers) called dumb terminals.

- Mainframes are preferred at business data processing corporate offices. They are quite expensive with several hard disks, RAID's and backup storage units.

Types Of Computers
Super Computers

- One of the fastest computer's type currently available, *They are problem scalable.*
- Computational speed of a super computer is measured in terms of Floating Point Operations Per Second or **FLOPS**.
- Examples: Cray X/MP-14 , Param – 8000, Param - Padma

DIGITAL DESIGN AND COMPUTER ORGANIZATION

Functional Units of A Computer T2:Ch1 1.2

Department of Computer Science and Engineering

Basic Structure of computers
Functional Units

A computer in its simplest form comprises five functional units:

1) Input Unit

2) Output Unit

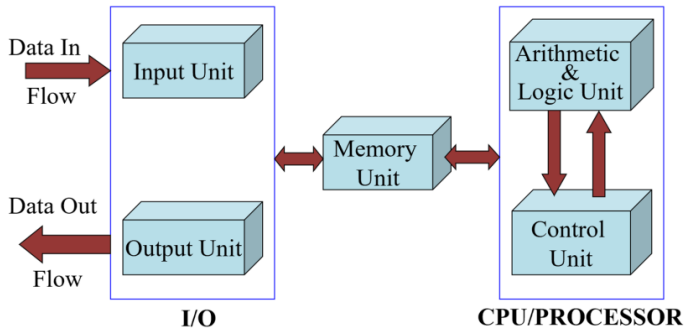
3) Memory Unit

4) Arithmetic & Logic Unit

5) Control Unit

Basic Structure of computers
Functional Units

The following figure depicts functional units of a computer:



Functional Units of a Computer
Input Unit

- Computer accepts encoded information through input unit. The standard input device is a keyboard of a video monitor or terminal.
- Whenever a key is pressed, keyboard controller sends the scanned code of that letter, digit or symbol to CPU/Memory.
- Examples include Mouse, Joystick, Tablet or Digitizer, Scanner etc.

Functional Units of a Computer
Memory Unit

Memory unit stores the program instructions, data operands on and results of computations etc. Memory unit is classified as:

1. Primary /Main Memory

2. Secondary /Auxiliary Memory

Functional Units of a Computer
Memory Unit

Primary memory is the computer memory that is directly accessible by CPU. It is comprised of DRAM and provides the actual working space to the processor. It holds the data and instructions that the processor is currently working on.

- **Advantages of Primary Memory**
  - **a. Speed**: Provides fast access to data and instructions.
  - **b. Direct Access:** Allows the CPU to quickly read from and write to memory.
- **Disadvantages of Primary Memory**
  - **a. Limited Size:** Is smaller compared to secondary memory

Functional Units of a Computer
Memory Unit

Secondary Memory / Mass Storage

The contents of the secondary memory first get transferred to the primary memory and then are accessed by the processor, this is because the processor does not directly interact with the secondary memory.

- **Advantages of Secondary Memory**
  - **Persistence:** Retains data even without power.
  - **Large Capacity:** Typically offers much more storage space than primary memory.
- **Disadvantages of Secondary Memory**
  - **Speed:** Slower access compared to primary memory.

Functional Units of a Computer
Memory Unit

1. Main memory is classified again as RAM and ROM.

- **RAM** is termed as Read/Write memory or user memory that holds run time program instruction and data.
- **ROM** holds system programs and firmware routines such as BIOS, POST, I/O Drivers that are essential to manage the hardware of a computer.

RAM

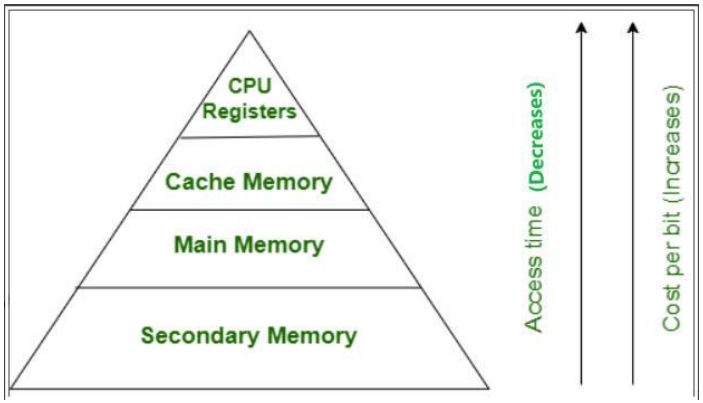
ROM

Functional Units of a Computer
Memory Unit

2. Secondary /Auxiliary Memory:

- While primary storage is essential, it is ***volatile*** in nature (i.e. its contents will be lost in the absence of power) and expensive too. Additional requirement of memory would be supplied as auxiliary memory at **cheaper cost.**
- Secondary memory are *magnetic memories* viz Floppy disk, Hard disk, Magnetic tape, CD-ROM etc.
- Secondary memories are ***non volatile*** in nature (contents will not be lost in the absence of power).

Functional Units of a Computer
Memory Hierarchy



Functional Units of a Computer
ALU

What is an ALU?

- An ALU is a digital circuit that performs arithmetic and logic operations on binary numbers.
- ALUs are essential components of computer systems, as they enable the execution of various computational tasks.

How does an ALU work?

- ALUs process binary data using logic gates, which perform operations like AND, OR, NOT, and XOR. The combination of these gates allows ALUs to execute complex arithmetic and logic operations.

Functional Units of a Computer
Functions of an ALU

- Operations are **executed** in the Arithmetic and Logic Unit (ALU).
  - Arithmetic operations such as addition, subtraction.
  - Logic operations such as comparison of numbers.
- In order **to execute** an instruction, **operands** need to be brought into the ALU **from the memory.**
  - Operands are stored in general purpose registers available in the ALU.
  - Access times of general purpose registers are faster than the cache.
- **Results** of the operations are **stored back** in the memory or retained in the processor for immediate use.

Functional Units of a Computer
Output Unit

- Computer **returns** the computed results, error messages, etc., via **output unit.**
- The standard output device is a video monitor, LCD/TFT monitor.
- Other output devices are *printers, plotters* to take a paper copy of the results, programs, graphs called *print out* or a hardcopy
- Printers types: Dot Matrix printer, Inkjet printer, Laser printer. . .

Functional Units of a Computer
Control Unit

- **Coordinates activities:** The control unit oversees the operations of all components within the CPU.
- **Issues timing signals:** It sends signals like MEMR (Memory Read), MEMW (Memory Write), IOR (Input Output Read), and IOW (Input Output Write) to control data transfers.
- **Governs data transfers:** These timing signals determine when specific operations should occur.
- **Interprets instructions:** The control unit decodes instructions to determine the required actions.

Functional Units of a Computer
Control Unit

Operation of a computer can be summarized as:

- **Accepts** information from the input units **(Input unit)**.
- **Stores** the information **(Memory)**.
- **Processes** the information **(ALU).**
- Provides **processed results** through the output units **(Output unit).**
- **Operations** of Input unit, Memory, ALU and Output unit are **coordinated** by **Control unit.**
  - **Instructions control "what"** operations take place (e.g. data transfer, processing).
  - **Control unit** generates timing signals which determines **"when"** a particular operation takes place.

Functional Units of a Computer
Control Unit

Main conceptual events/operations of a computer

1.A set of instructions which perform a given task, called a *program* must reside in the main memory of computer during its execution.

2.The CPU fetches those instructions sequentially one-by-one from the main memory, decodes them and perform the specified operation on associated data operands in ALU.

3. Processed data i.e. useful information will be displayed on an output unit.

4. All activities pertaining to processing and data movement inside the computer machine are governed by *control unit*.

Functional Units of a Computer
Control Unit

Information Handled by a Computer:

Instructions/machine instructions

- Govern the transfer of information within a computer as well as between the computer and its I/O devices
- Specify the arithmetic and logic operations to be performed
- List of instructions is called a Program(in memory)

Data (numbers or encoded chars)

- Used as operands by the instructions. Eg(Binary, BCD, ASCII,EBCDIC)

Functional Units of a Computer
Control Unit

Information in a computer -- Instructions
Instructions specify commands to:

- Transfer information within a computer (e.g., from memory to ALU)
- Transfer of information between the computer and I/O devices (e.g., from keyboard to computer, or computer to printer)
- Perform arithmetic and logic operations (e.g., Add two numbers, Perform a logical AND).

A sequence of instructions to perform a task is called a program, which is stored in the memory.
Processor fetches instructions that make up a program from the memory and performs the operations stated in those instructions.
What do the instructions operate upon?

DIGITAL DESIGN AND COMPUTER ORGANIZATION

Basic Operational Concepts T2:Ch1 1.3

Department of Computer Science and Engineering

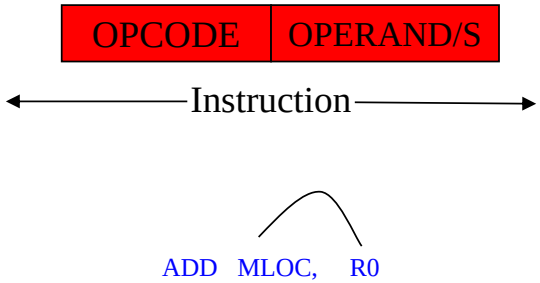
Functional Units of a Computer
Basic Operational Concepts

Review

- Activity in a computer is governed by **instructions.**
- To perform a task, an appropriate program consisting of a list of instructions is stored in the **memory.**
- **Individual instructions** are brought from the **memory into the processor**, which executes the specified operations.
- Data to be used as **operands** are also stored in the **memory.**

Functional Units of a Computer
Basic Operational Concepts

An Instruction consists of two parts: An Operation code and operand/s.



Functional Units of a Computer
Basic Operational Concepts

Execution steps: To add two operands

Step 1: **Fetch** the **instruction** from main memory into the processor
Step 2: **Fetch** the **operand** at location MLOC (memory location)from main memory into the processor
Step 3: **Add** the memory **operand** (i.e. fetched contents of MLOC() to the contents of **register R0**
Step 4: **Store** the resulting **sum in R0** itself.

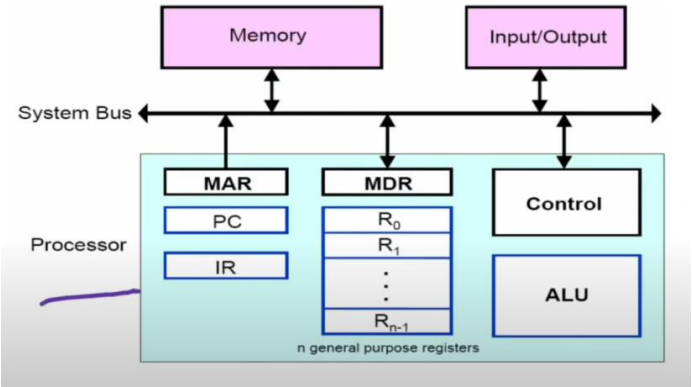
The original contents of R0 is overwritten.

Functional Units of a Computer
Basic Operational Concepts

- Prev example combines a memory access operation with an ALU operation
- This can be realized as
  Load LOCA,R1
  Add R1,R0
- Transfers b/w the memory and the cpu are started by sending the address of the location to memory unit and issuing the appropriate control signals

Functional Units of a Computer
Basic Operational Concepts

Processor and Main Memory Interaction



Functional Units of a Computer
Registers

What is a Register?

Definition: A register is a small, fast storage location within the CPU used to temporarily hold data and instructions during processing.

Key Characteristics:

- **Fast Access:** Registers are the fastest form of memory, as they are located directly within the CPU.
- **Limited Size:** Registers usually have a very small storage capacity, typically 32-bit or 64-bit, depending on the CPU architecture.
- **Temporary Storage:** Used to store data temporarily for fast retrieval during computations.

Functional Units of a Computer
Types of Registers

- **Instruction Register (IR)**
  - **Purpose:** Temporarily holds the instruction currently being executed by the processor.
  - **Function:** Decodes the instruction, allowing the processor to understand and execute the required operation.

- **Program Counter (PC)**
  - **Purpose:** Tracks the address of the next instruction to be executed in the program sequence.
  - **Function:** Automatically increments to point to the next instruction address after each execution.

Functional Units of a Computer
Types of Registers

- **General-Purpose Registers (R0 – Rn-1)**
  - **Purpose:** Temporarily store operands for quick access during instruction execution, supporting various operations like arithmetic and data transfer.
  - **Function:** Hold and manipulate data during calculations, minimizing memory access for faster CPU performance.

- **Memory Address Register (MAR)**
  - **Purpose:** Holds the memory address of the location that the CPU is about to read from or write to.
  - **Function:** Communicates directly with the memory bus for fetching or storing data.

Functional Units of a Computer
General Purpose Register

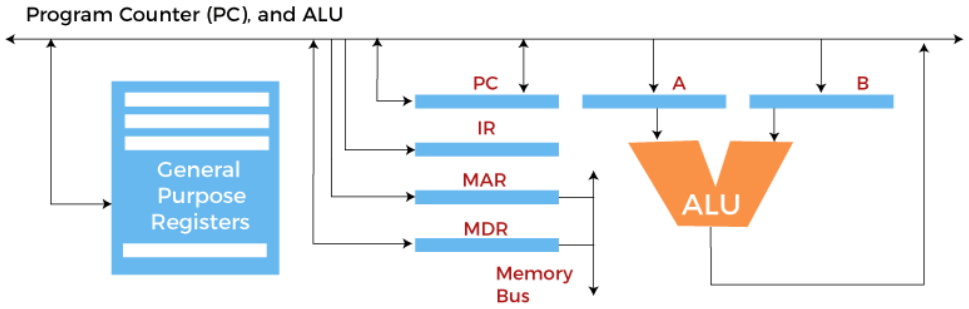
	7 0	Addr.	
	R0	0x00	
	R1	0x01	
	R2	0x02	
	...		
	R13	0x0D	
General	R14	0x0E	
Purpose	R15	0x0F	
Working	R16	0x10	
Registers	R17	0x11	
	...		
	R26	0x1A	X-register Low Byte
	R27	0x1B	X-register High Byte
	R28	0x1C	Y-register Low Byte
	R29	0x1D	Y-register High Byte
	R30	0x1E	Z-register Low Byte
	R31	0x1F	Z-register High Byte

Functional Units of a Computer
Types of Registers

- **Memory Data Register (MDR)**
  - **Purpose:** Contains the actual data being transferred to or from the addressed memory location.
  - **Function:** Acts as a buffer between the CPU and memory.

Functional Units of a Computer
Basic Operational Concepts

Operational Steps Associated with Processor & Main Memory Interaction



One-bus datapath

Functional Units of a Computer
Basic Operational Concepts

How is an instruction executed ?

A fetch-decode cycle is performed for each instruction in a program.

This means that every time the CPU needs to execute a new instruction, it goes through this cycle:

1. **Fetch:** The CPU retrieves the next instruction from memory using the program counter (PC).
2. **Decode:** The CPU interprets the instruction to determine what operation it needs to perform and what data it needs to use.
3. **Execute:** The CPU carries out the instruction, which might involve performing calculations, manipulating data, or controlling other parts of the computer.

Functional Units of a Computer
Basic Operational Concepts

How is the next instruction executed ?

After the execution of an instruction, the PC is typically incremented to point to the next instruction in the program sequence, and the cycle repeats. This continuous process allows the CPU to execute a program's instructions one by one until the program is finished.

Let us now look at this process in detail.

Functional Units of a Computer
Basic Operational Concepts

1. Programs reside in memory through input devices:

Programs are loaded into the computer's memory from input devices like keyboards or storage devices (e.g., hard drives, SSDs). This step typically happens before the program starts executing.

2. PC is set to point to the first instruction:

The Program Counter (PC) holds the address of the next instruction to be executed. At the start, it is initialized to point to the first instruction in the program.

3. The contents of PC are transferred to MAR:

The Memory Address Register (MAR) holds the address of the memory location that is being accessed. The address from the PC is copied to MAR.

Functional Units of a Computer
Basic Operational Concepts

4. A Read signal is sent to the memory:

A signal is sent to the memory to indicate that data should be read from the address specified by the MAR.

5. The first instruction is read out and loaded into MDR:

The instruction located at the address in MAR is read from memory and placed into the Memory Data Register (MDR).

6. The contents of MDR are transferred to IR:

The Instruction Register (IR) holds the current instruction to be executed. The data in MDR (which is the fetched instruction) is transferred to IR.

Functional Units of a Computer
Basic Operational Concepts

7. The instruction is ready to be decoded and executed:

The instruction in IR is decoded to understand what action needs to be performed.

8.If the instruction involves an operation to be performed by the ALU:

- If the instruction requires arithmetic or logical operations, the necessary operands must be obtained.
- **Get operands:**
- **General-purpose register:** If the operands are in general-purpose registers, these registers are accessed directly.

Functional Units of a Computer
Basic Operational Concepts

8.

- **Memory:** If the operands are in memory, the following steps are performed:
  - The address of the operand is transferred to MAR.
  - A Read signal is sent to memory to fetch the operand.
  - The operand is read into MDR and then transferred to the ALU.

9.Perform operation in ALU:
The Arithmetic Logic Unit (ALU) performs the operation specified by the instruction using the operands it received.

Functional Units of a Computer
Basic Operational Concepts

10.Store the result back:

- **To general-purpose register:** If the result needs to be stored in a register, it is written to the appropriate general-purpose register.
- **To memory:** If the result needs to be stored in memory:
  - The address for storing the result is transferred to MAR.
  - The result is placed in MDR.
  - A Write signal is sent to memory to store the result.

11.During the execution of the current instruction, PC is incremented to the next instruction

Functional Units of a Computer
Interrupt

- **What is an interrupt ?**
  - An interrupt is a request from an I/O device for service by the processor.
- **Why is an interrupt necessary?**
  - Normal execution of programs may be **interrupted** if some device requires **urgent** servicing.
  - To deal with the situation immediately, the normal execution of the current program must be interrupted.

Functional Units of a Computer
Procedure of interrupt operation

1. The I/O device raises an **interrupt signal**.
1. The processor provides the requested service by executing an appropriate **interrupt-service routine**.
1. The **state of the processor is first saved** before servicing the interrupt Normally, the contents of the PC, the general registers, and some control information are stored in memory.
1. When the interrupt-service routine is completed, the **state of the processor is restored** so that the interrupted program may continue.

DIGITAL DESIGN AND COMPUTER ORGANIZATION

Bus Structures T2:Ch1 1.4

Department of Computer Science and Engineering

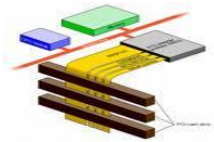
Functional Units of a Computer
Bus Structures

What is Bus ?

- A bus is a collection of wires that connects different parts of a computer in an organized manner for the purpose of communicating information such as memory address, I/O address, Data, Control bits etc.

Why is a bus required ?

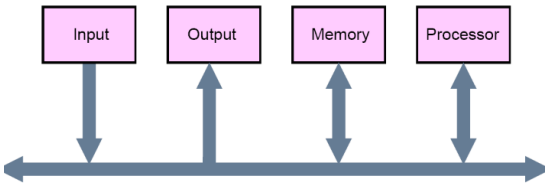
- With the bus structure, speed of operation can be achieved since collection of wires in a bus permits for transferring all bits of information at a time i.e., in parallel. One byte or one full word at a time simultaneously on different wires.



Functional Units of a Computer
Bus Structures

A group of lines that serves a connecting path for several devices is called a bus

- In addition to the lines that carry the data, the bus must have lines for address and control purposes
- The simplest way to interconnect functional units is to use a single bus, as shown below



Functional Units of a Computer
Bus Structures

Single-bus

- Simplest way of interconnecting functional units.
- All units are connected to this bus.
- Can be used for only one transfer at a time, thereby only 2 units can actively use the bus at a time.
- Low Cost and flexibility for attaching peripheral devices.

Functional Units of a Computer
Bus Structures

Drawbacks of the Single Bus Structure

- The devices connected to a bus vary widely in their speed of operation
  - Some devices are relatively slow, such as printer and keyboard
  - Some devices are considerably fast, such as optical disks
  - Memory and processor units operate are the fastest parts of a computer

Solutions

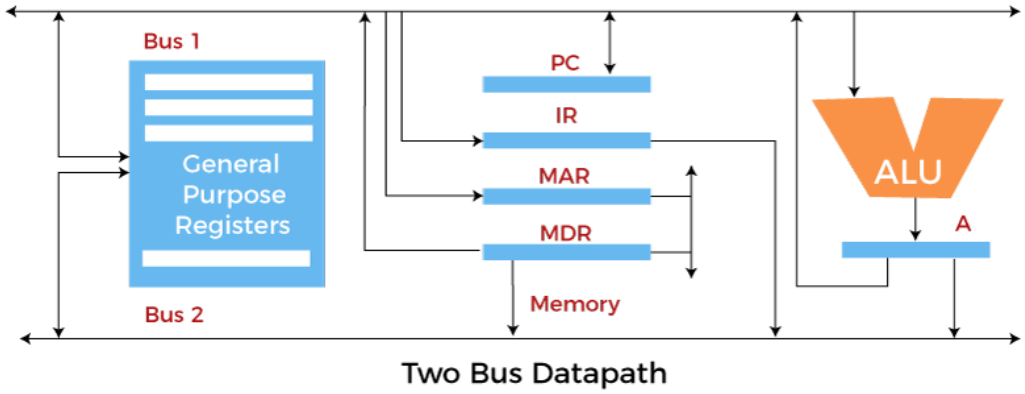
- Efficient transfer mechanism is needed to cope with this timing problem
  - A common approach is to include **buffer registers**.
  - An another approach is to use **two-bus structure** and an additional transfer mechanism(One bus can be used to fetch instruction other can be used to fetch data, required for execution).

Functional Units of a Computer
Buffer Register

- **Working :**
- The approach is to include buffer registers with the devices to hold the information during transfers which smoothes out the timing difference between the cpu,memory and i/o devices

- A buffer register holds data temporarily. Buffer register compensate the timing differences among processor, memory and slow speed peripherals like Printer, Key-board, magnetic tape etc., during the transfers over a common communication path i.e., single bus.

Functional Units of a Computer
Two Bus Architecture



Two Bus Datapath

THANK YOU

Team DDCO
Department of Computer Science

DIGITAL DESIGN AND COMPUTER ORGANIZATION

Number, Arithmetic Operations, And Characters
Memory Locations And Address

T2: Chapter 2: 2.1-2.2

Department of Computer Science and Engineering

Number, Arithmetic Operations, And Characters
Introduction

- Computers are built using logic circuits that operate on information represented by two- valued electrical signals
- We label the two values as 0 and 1
- we define the amount of information represented by such a signal as a bit of information, where bit stands for binary digit.
- The most natural way to represent a number in a computer system is by a string of bits, called a binary number.
- A text character can also be represented by a string of bits called a character code.

Number, Arithmetic Operations, And Characters
Outline

1. **Number Representation**
2. **Addition of Positive Numbers**
3. **Addition And Subtractions of Signed Numbers**
4. **Overflow In Integer Arithmetic**
5. **Characters**

DIGITAL DESIGN AND COMPUTER ORGANIZATION

1. Number Representation

T2:Ch2 2.1

Department of Computer Science and Engineering

Number Representation
Signed Integer

- 3 major representations:
  - Sign and magnitude
  - One's complement
  - Two's complement

➢In all three systems, the leftmost bit is 0 for positive and 1 for negative
➢Positive numbers have identical representations in all systems, but negative values have different representations

Assumptions:
- 4-bit machine word
- 16 different values can be represented
- Roughly half are positive, half are negative

Number Representation
Binary Signed Integer Representation

B	Values represented		
$b_3\,b_2\,b_1\,b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	- 0	- 7	- 8
1 0 0 1	- 1	- 6	- 7
1 0 1 0	- 2	- 5	- 6
1 0 1 1	- 3	- 4	- 5
1 1 0 0	- 4	- 3	- 4
1 1 0 1	- 5	- 2	- 3
1 1 1 0	- 6	- 1	- 2
1 1 1 1	- 7	- 0	- 1

Figure 2.1. Binary, signed-integer representations.

Number Representation
Summary of the table

- SIGN & MAGNITUE SYSTEM: Negative value is obtained by changing the sign bit (MSB)

  Range: **$-(2^{n-1}) - 1$ to $+2^{n-1} - 1$**

- SIGNED 1'S COMPLEMENT: Negative number is obtained by complementing each bit of the corresponding positive number i.e $(2^n-1) - N$

  Range: **$-(2^{n-1}) - 1$ to $+2^{n-1} - 1$**

- SIGNED 2'S COMPLEMENT: Negative number is obtained by taking 2's complement of positive number i.e $2^n - N$
- Range: $-(2^{n-1})$ to $+(2^{n-1}-1)$

DIGITAL DESIGN AND COMPUTER ORGANIZATION

2. Addition of Positive Number T2:Ch2 2.1

Department of Computer Science and Engineering

Addition of Positive Number

```
  0        1        0        1
+ 0      + 0      + 1      + 1
------   -------  -------  --------
  0        1        1        1 0
------   --------  -------  --------
```

Addition of 1-bit numbers

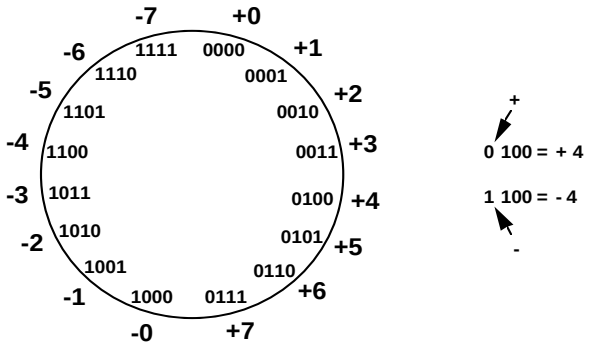
Carry-out

DIGITAL DESIGN AND COMPUTER ORGANIZATION

3. Addition And Subtraction of Signed Numbers T2:Ch2 2.1

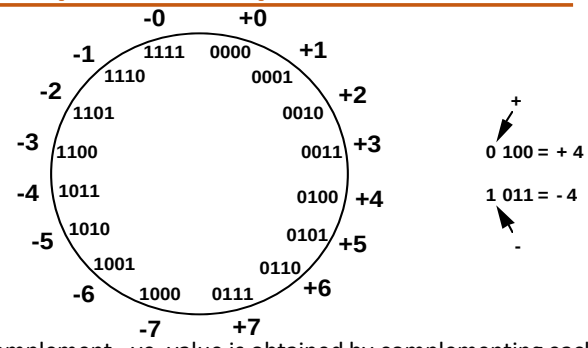
Department of Computer Science and Engineering

Addition And Subtraction of Signed Numbers
Sign and Magnitude Representation



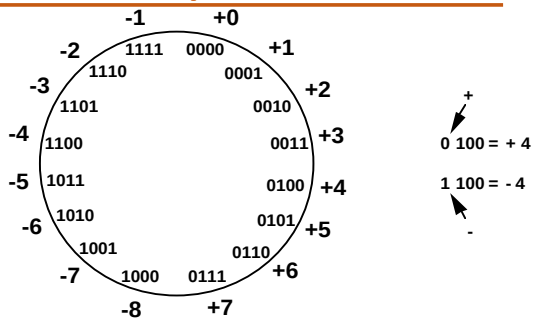
High order bit is sign: 0 = positive (or zero), 1 = negative
Three low order bits is the magnitude: 0 (000) thru 7 (111)
Range = $-(2^{n-1})-1$ to $+2^{n-1}-1$
Two representations for 0

Addition And Subtraction of Signed Numbers
One's Complement Representation



- In 1's complement -ve value is obtained by complementing each bit of the corresponding +ve value
- The operation of forming 1's complementing is equivalent to subtracting that no. from $2^n - 1$
- Still two representations of 0! This causes some problems

Addition And Subtraction of Signed Numbers
Two's Complement Representation



like 1's comp except shifted one position clockwise

- The operation of forming 2's complementing is equivalent to subtracting that no. from 2^n
- 2's complement of a no. can be obtained by adding 1 to the 1's complement of that no.
- Only one representation for 0
- One more negative number than positive number

Addition And Subtraction of Signed Numbers

- For signed arithmetic operations 2's complement format is used
- For signed arithmetic operation sign and magnitude and 1's complement format is not suitable

Addition And Subtraction of Signed Numbers
Addition using 2's Complement

```
  0111
+ 1101
---------
(1)0100
--------
```

The carry out should be ignored

Example of addition of n – bit signed numbers using 2's complement representation

```
+2      0010          +4      0100
+3    + 0011          -6    +1010
----  ----------      -----  ---------
+5      0101          -2      1110
----  ----------      -----  ----------

-4      1100          -6      1010
-3    +  1101         +3    +  0011
---   --------        ----    --------
-7    (1)1001         -3      1101
---   ----------      ----    ---------
```

Example of subtraction of n – bit signed numbers using 2's complement representation

```
  +4        0100                        0100
-  +3     - 0011   ➔                  +  1101
------------  --------                 ---------
  +1                                   (1)0001
------------                           ---------

   +5        0101                       0101
-  -2      - 1110   ➔                 + 0010
-------    ---------                   --------
   +7                                   0111
--------                               ---------
```

Example of subtraction of n – bit signed numbers using 2's complement representation

```
    -3          1101                     1101
-   -6       -  1010   ➔        +        0110
------------   --------                  -------
    +3                                 (1) 0011
------------                             ---------

    -4          1100                     1100
-   +2        - 0010   ➔            +     1110
-------       ---------                  --------
    -6                                 (1)1010
--------                                 ---------
```

Example of addition and subtraction of n – bit signed numbers using 2's complement representation leading to overflow

```
  +3           0011                       0011
-    - 6     -  1010   ➔           +     0110
------------   --------                  ---------
  +9                                    1001 ➔ -7
------------                            ---------

  -4                 1100
+ -5               + 1011
-------              --------
   -9                1  0111   ➔ +7
--------            ---------
```

DIGITAL DESIGN AND COMPUTER ORGANIZATION

4. Overflow In Integer Arithmetic T2:Ch2 2.1

Department of Computer Science and Engineering

Overflow In Integer Arithmetic

- Overflow will not occur when two numbers having opposite signs added
- Overflow can occur only if two numbers having same signs are added
- The carry out signal from the sign bit position is not a sufficient indicator of overflow when adding signed numbers
- A simple way to detect the overflow is to examine the signs of the two numbers X and Y and the sign of the result. When both the operands X and Y have the same sign, an over flow occurs when the sign of S is not the same as the signs of X and Y

DIGITAL DESIGN AND COMPUTER ORGANIZATION

5. Characters T2:Ch2 2.1

Department of Computer Science and Engineering

Characters

- In addition to numbers, computers must be able to handle nonnumeric text information consisting of characters.
- Characters can be letters of the alphabet, decimal digits, punctuation marks, and so on.
- They are represented by codes that are usually eight bits long. One of the most widely used such codes is the American Standards Committee on Information Interchange (ASCII) code

DIGITAL DESIGN AND COMPUTER ORGANIZATION

Memory Locations And Addresses T2: Chapter 2: 2.2

Department of Computer Science and Engineering

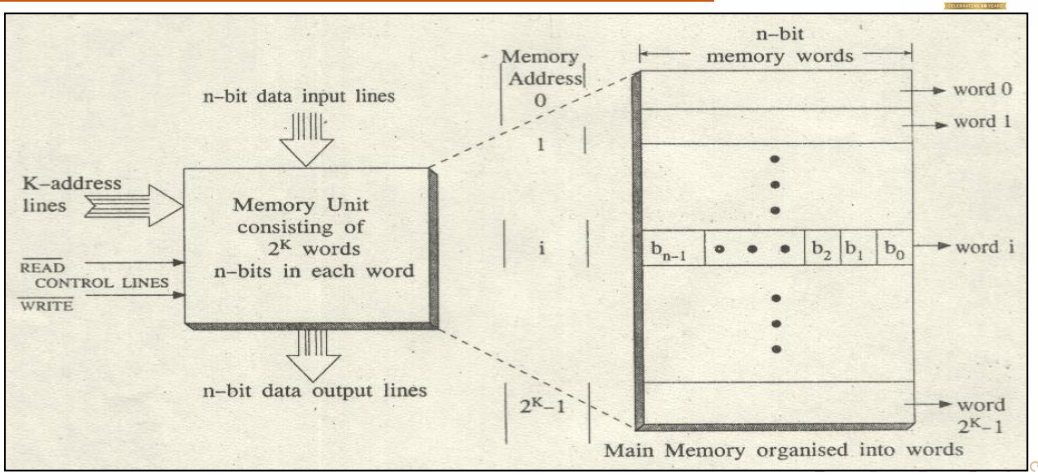
Memory Locations And Addresses
Introduction

- Main memory is a collection of semiconductor storage cells,
- Each Cell can store one bit of information say a “1” or a “0” bit.
- Memory stores binary information in groups of bits called *words*.
- Memory word is an addressable location to store a number, characters as data or an instruction in binary.
- The length of a memory word can be specified as n-bits per word called the word length.
- Modern computers have word length of the order 16-bits to 64-bits.

Memory Locations And Addresses
Memory Addresses

- Each word of information to be stored or retrieved from memory requires an identifying name called its Address.
- Special input lines called Address lines (wires) select each and every word in memory distinctly.
- For which each word in memory is assigned an identification number, called as address starting from 0 and continues 1,2,3,…….,up to 2^k where k is the number of address lines.
- The selection of a specific memory word is achieved by applying the K-bit binary address to the address lines.
- The address range from 0 to $2^K - 1$ constitute address space comprising 2K memory words.

Memory Locations And Addresses
Memory Organization and Addressing



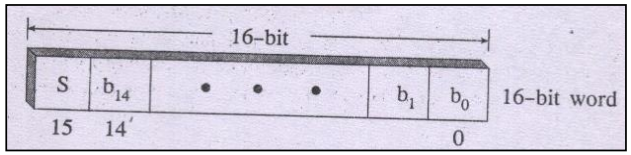
Memory Locations, Addresses, and Operation

- It is impractical to assign distinct addresses to individual bit locations in the memory.
- The most practical assignment is to have successive addresses refer to successive byte locations in the memory – byte-addressable memory.
- Byte locations have addresses 0, 1, 2, … If word length is 32 bits, they successive words are located at addresses 0, 4, 8,…

Memory Locations, Addresses, and Operation
Encoding of Information

Numbers

➢ The length of the operand depends on the specific data type. 16 - bit **16 – bit memory word pattern** to hold integer numbers.

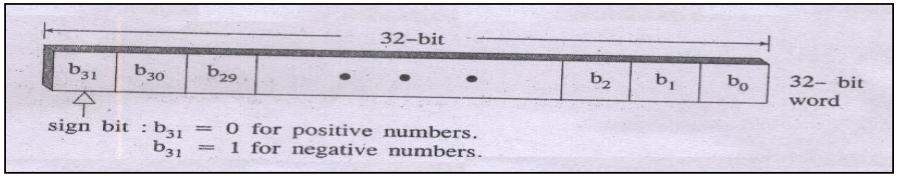


Bit 15 indicate sign bit→ If it is 0, number is +ve
If it is 1, number is –ve

Binary number 0000 0010 0001 1101 is interpreted as a decimal +541

Memory Locations, Addresses, and Operation
Encoding of Information

➢ **32-bit memory word pattern** to hold integer numbers



➢Magnitude of the number can be computed using binary positional weight notation as:

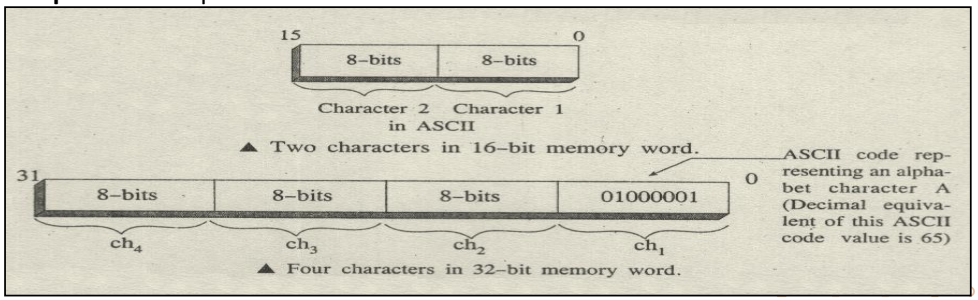
$$= b_{30} \times 2^{30} + b_{29} \times 2^{29} + \ldots\ldots\ldots\ldots + b_1 \times 2^1 + b_0 \times 2^0$$

Magnitude represented in 32 –bit memory words using signed – magnitude representation can range from 0 to $2^{31} - 1$.

Memory Locations, Addresses, and Operation
Characters

Characters

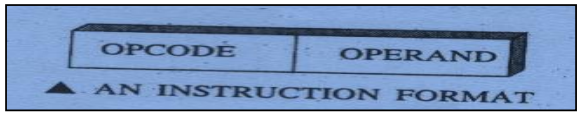
➢ A memory word can also store character information such as alphabets, decimal digits, symbols like +, -, $, ?, punctuation marks like; :,' " etc., Such characters can be represented using a 7-bit ASCII code. **Memory Word Bit pattern** to represent characters



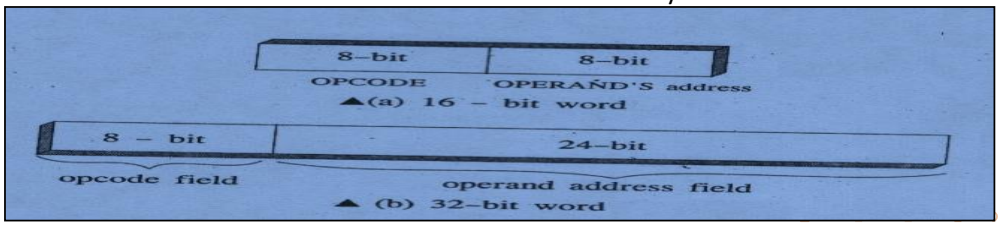
Memory Locations, Addresses, and Operation
Instructions

Instructions

➢ Program instructions are also stored in main memory words. An instruction has two parts : operation code (opcode) and operand field:



➢A machine instruction format in 16-bit and 32-bit memory words:



Memory Locations, Addresses, and Operation Instructions

➢The magnitude of the number in 16-bit memory word is computed as

$= b_{14} \times 2^{14} + b_{13} \times 2^{13} + \ldots\ldots\ldots\ldots + b_1 \times 2^1 + b_0 \times 2^0$

For a number in signed – magnitude representation, where in bit 15(b_{15}) specify a sign bit, a 0 or 1.

➢ Now Binary number 0000 0010 0001 1101 has the equivalent magnitude in decimal as:

$= 0x2^{14} + 0x2^{13} + 0x2^{12} + 0x2^{11} + 0x2^{10} + 1x2^9 + 0x2^8 + 0x2^7$
$+ 0x2^6 + 0x2^5 + 1x2^4 + 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0$

$= 512 + 16 + 8 + 4 + 1 = +541$

Memory Locations And Address Outline

1. **Byte Addressability**
2. **Big-Endian And Little-Endian Assignments**
3. **Word Alignment**
4. **Accessing Numbers, Characters, And Character Strings**

DIGITAL DESIGN AND COMPUTER ORGANIZATION

1. Byte Addressability T2:Ch2 2.2

Department of Computer Science and Engineering

Byte Addressability

➢ If the smallest addressable unit in a computer is a word, then such computers are said to be **word addressable computer**.

➢A computer in which each byte stored can be addressed individually is called as **byte addressable computer.**

➢ If the word length of the given computer machine is 32 –bit words, then sequence of memory words can be located subsequently at addresses 0 (word 0), 4 (word 1), 8 (word 2)…

➢ Each such word contains 4 bytes.

DIGITAL DESIGN AND COMPUTER ORGANIZATION

2. Big-Endian & Little-Endian Assignments T2:Ch2 2.2

Department of Computer Science and Engineering

Big-Endian And Little-Endian Assignment

Two kinds of representations in assigning bytes in a memory word are:

(1) Big-endian assignment

(2) Little-endian assignment

➢**Big – endian assignment**: Here the bytes are assigned a number starting with most significant byte (left most byte) and successively next bytes in that word. Also the word is given the same memory address as its most significant byte.

➢Big – Endian scheme of address assignment is used in 68000*(CISC processor)* *processor* and *Power PC (RISC processor) processors.*

Big-Endian Assignment

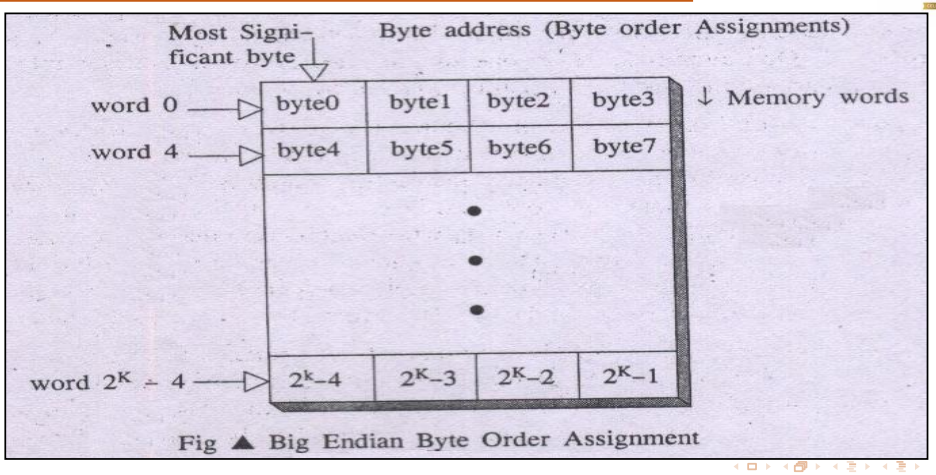


Fig ▲ Big Endian Byte Order Assignment

Little-Endian Assignment

In this scheme byte 0 appears as the right most byte of word 0.
Low order bytes represents least significant bytes or right most bytes.

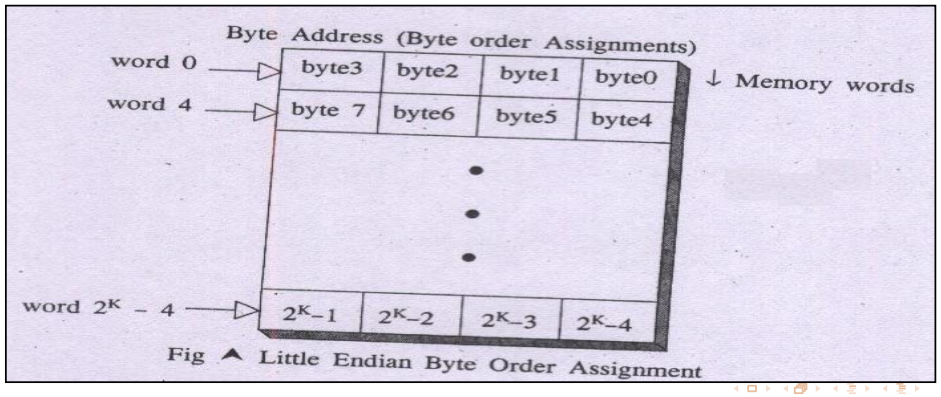
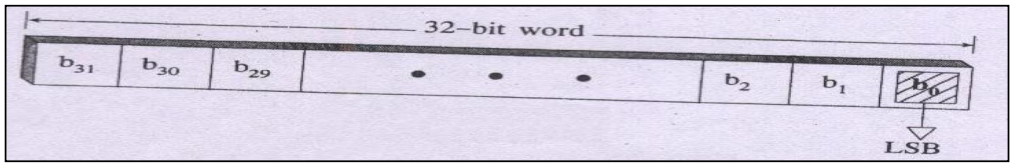


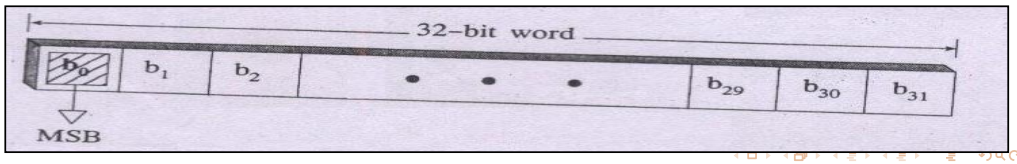
Fig ▲ Little Endian Byte Order Assignment

Conventions for numbering bits in a memory word

For example 68000 processor uses a convention in which bit 0 (b0) is treated as least significant bit (LSB) of a word.



➢ Using the other convention, Power PC for example treats bit 0 (b0) as the most significant bit (MSB) of a memory word.



Word Alignment

It refers to grouping of bytes in sequence in a given memory unit.

➢ For a computer with 32-bit word length

word 0 at address 0 (comprising bytes 0,1,2,3),
word 1 at address 4 (comprising bytes 4,5,6,7)
word 2 at address 8 (comprising bytes 8,9,10,11) . . .

➢ Similarly for a computer with 16-bit word length:

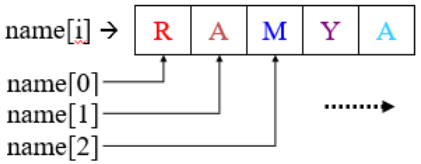
word 0 at address 0 (comprising bytes 0,1),
word 1 at address 2 (comprising bytes 2,3)
word 2 at address 4 (comprising bytes 4,5) . . .

➢ Similarly for a computer with 64-bit word length:

word 0 at address 0 (comprising bytes 0,1,2,3,4,5,6,7),
word 1 at address 8 (comprising bytes 8,9,10,11,12,13,14,15)
word 2 at address 16 (comprising bytes 16,17,18,19,20,21,22,23) . . .

Numbers, Character and Character Strings

- Numbers can be accessed from memory unit by specifying its word address or multiple byte address in memory.
- Character constants occupy *single byte* in memory. Therefore they can be accessed specifying their **byte address.**
- For example a character string RAMYA in an array of char can be accessed at byte addresses in memory say bytes 0,1,2,3 & 4



THANK YOU

Team DDCO
Department of Computer Science

DIGITAL DESIGN AND COMPUTER ORGANIZATION

Memory Operations
Instructions And Instruction Sequencing

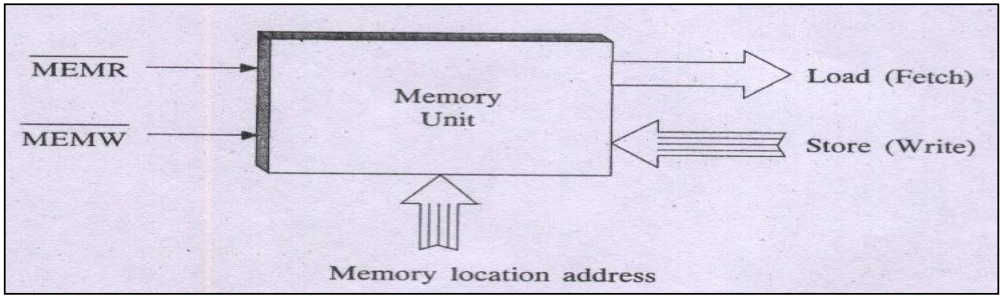
T2: Chapter 2: 2.3-2.4

Department of Computer Science and Engineering

Memory Operations
Introduction

While executing a program, two distinct operations associated with Processor-Memory interaction are:

- Load operation / Memory Read - (Instruction fetch / Operand fetch)
- Store operation / Memory Write - (Write computed results)



Memory Operations
Load (Read Or Fetch)

- Transfers a copy of the contents of a specific memory location to the processor
- Processor sends address of the desired location to memory and requests that its contents be read(Issues read signal)
- Memory reads the data and sends it to the processor
- Memory contents remains unchanged

Memory Operations
Store Or Write

- Transfers an item of info from the CPU to a specific memory location, destroying the former contents of that location
- Processor sends the address of the desired location to the memory, together with the data to be written
- Issues write signal
- Sends the data via data bus and write into the selected particular memory location

Instructions And Instruction Sequencing
Introduction

- An instruction is a command to the processor to perform a given task on data operands.
- A program is a set of instructions that specify operations, operands & the sequence by which processing has to occur.
- Thus operation of a computer is controlled by stored program
- In general a computer must support 4 categories of operations:
  - 1.Data Movement : To conduct I/O transfers
  - 2.Data Storage : Across Memory and processor registers.
  - 3.Data Processing : Arithmetic and logical operations
  - 4. Program sequencing and control : Test and Branch.

Instructions And Instruction Sequencing
Outline

1. **Register Transfer Notation**
2. **Assembly Language Notation**
3. **Basic Instruction Types**
4. **Instruction Execution And Straight-Line Sequencing**
5. **Branching**
6. **Condition Codes**
7. **Generating Memory Addresses**

DIGITAL DESIGN AND COMPUTER ORGANIZATION

1. Register Transfer Notation T2:Ch2 2.4

Department of Computer Science and Engineering

Register Transfer Notation

- Data operands & Instructions are situated in main memory locations & processor registers.
- Like mnemonics for Opcodes, Symbolic names or label identifiers are used to name or identify such locations.
- For instances memory location names include M, LOC, A, B, C, SUM, N1, VAR1, VAR2, NUM1, etc.,
- Similarly processor register names include R0, R1, R2, R3, R4, R5, etc., and registers of I/O subsystem may be designated as DATAIN, DATAOUT etc.

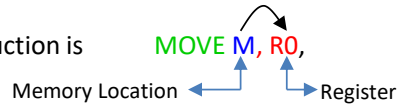
Register Transfer Notation

- While depicting operations, the contents of a memory location or a register are denoted by placing the corresponding name in a pair of square brackets.

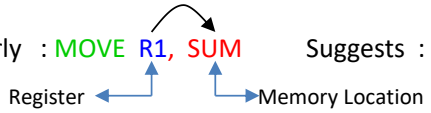
For instance : R0 ← [M]

Suggests to copy contents of memory location M to register R0

- The corresponding instruction is MOVE M, R0,



- Similarly : MOVE R1, SUM Suggests : SUM ← [R1]



- The instruction : MOVE B, LOC Means : LOC ← [B]

Register Transfer Notation

- Similar operations that involve Registers only are:

R1 ← [R2] MOVE R2, R1

- R1 ← [R1] + [R3] ADD R3, R1

Suggest to add contents of register R1 and register R3 and rewrite the R1 contents to store the sum.

- All such notations with leftward arrow (←) assignment operations involving register locations give rise to what is known as **Register Transfer Notation**

DIGITAL DESIGN AND COMPUTER ORGANIZATION

2. Assembly Language Notation T2:Ch2 2.4

Assembly Language Notation

- An assignment statement in a High Level Language Program:

  S = P + Q

- Here **P**, **Q** & **S** are variables.
- Variable name refers to *symbolic address of memory location* from which data operand can be read or written.

  S ← [P] + [Q]

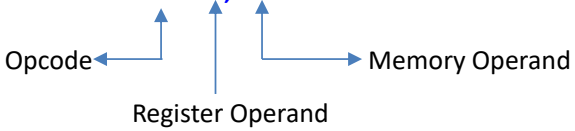
- This operation suggests contents of two memory locations **P** and **Q** are fetched from memory and transferred into processor where sum of two numbers is performed. The resulting sum is then sent back to memory and stored in memory location S.
- An instruction to this effect in Assembly Language Notation:

  Add P, Q, S

Assembly Language Notation

- Consider **R1 ← [R1] + [R3]**
- Equivalent Assembly Language instruction is **Add R3, R1**.
- Here the operation Code Mnemonic is Add
- Register Locations **R1**, **R3** represent *operand fields*

- Similarly, in the instruction **ADD R1, SUM**



- i.e. SUM ← [SUM] + [R1]

DIGITAL DESIGN AND COMPUTER ORGANIZATION

3. Basic Instruction Types T2:Ch2 2.4

Basic Instruction Types

- Three – Address Instruction
- Two – Address Instruction
- One – Address Instruction
- Zero – Address Instruction

➢ **Three – Address Instruction**

Suppose we would like to use a single machine instruction to perform the following addition operation:

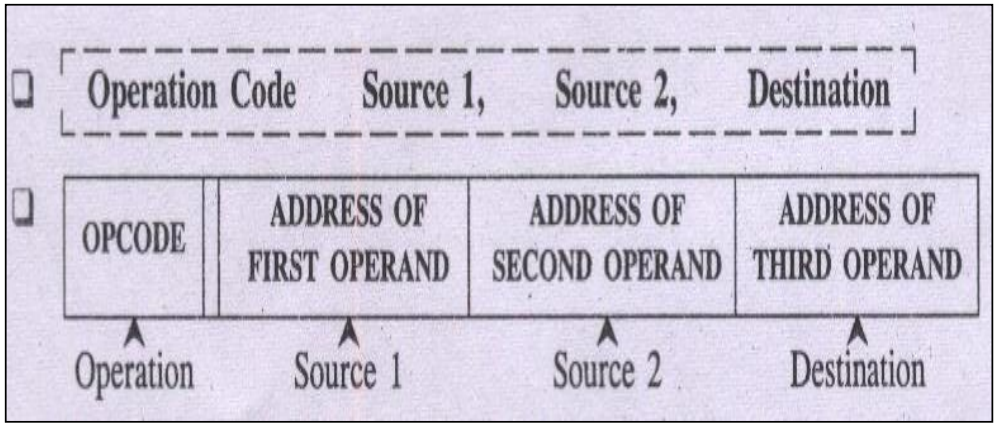
S ← [P] + [Q]

We will have to use a **three – address instruction** to carry out addition and to store the sum in a third variable S.

➢ Then three address instruction to perform addition is:

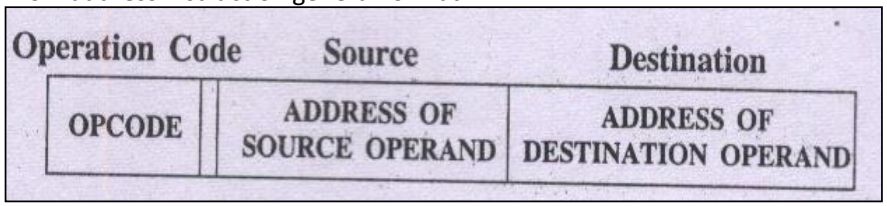
Add P, Q, S

Basic Instruction Types
General form of three – address instruction



Basic Instruction Types
Two – Address Instruction

➢ Two – address instruction general format:



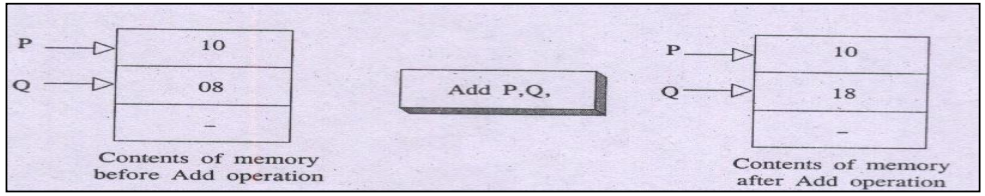
➢ Example of two – address instruction Add P, Q

➢ To perform operation of addition as : Q ← [P] + [Q]

Destination Operand ← Q ; [P] + [Q] → Source Operand/s

Basic Instruction Types

➢ Here one of the operand i.e. destination operand Q acts as source as well as destination.



➢ To perform S ← [P] + [Q]

➢ Use two – address instructions sequence:

MOVE Q, S
ADD P, S

Basic Instruction Types
One-Address Instruction

- One –address instruction format specify a single operand along with operation code.
- The requirement of second operand is fulfilled by an implicit processor register known as Accumulator (AC).
- Example of one-address instruction: ADD Q
- It Specify: AC ← [AC] + [Q]
- Load P Suggest AC ← [P] (Fetch P into AC)
- Store S Indicate S ← [AC] (Write AC into S)
- To perform S ← [P] + [Q]
- Use one – address instructions : Load P; AC ← [P]
  Add Q; AC ← [AC] + [Q]
  Store S; S ← [AC]

Basic Instruction Types
Zero – Address Instruction

- Stack – organized computer make use of a special memory structure called push down stack to store operands. In such computer machines it is possible to use instructions that contain only operation codes and no explicit operands.
- The name Zero – address specifies the absence of an address field of operands in machine instructions.
- Example of Zero – address instruction: ADD
- To perform operation of addition as TOS ← (P + Q)

Basic Instruction Types

- If processor supports ALU operations where one data may be in memory and other in register then the instruction sequence is for

ADD A,B,C :

MOVE A, Ri

ADD B, Ri

MOVE Ri,C

DIGITAL DESIGN AND COMPUTER ORGANIZATION

4. Instruction Execution And Straight-Line Sequencing

T2:Ch2 2.4

Department of Computer Science and Engineering

Instruction Execution And Straight-Line Sequencing

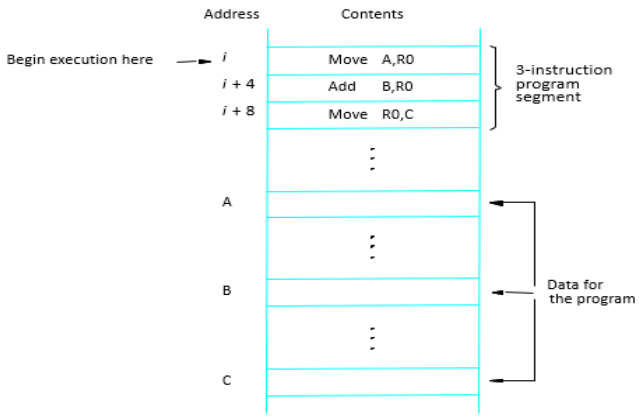


Figure 2.8. A program for C ← [A] + [B].

Assumptions:
- One memory operand per instruction
- 32-bit word length
- Memory is byte addressable
- Full memory address can be directly specified in a single-word instruction

Two-phase procedure
-Instruction fetch
-Instruction execute

Instruction Execution And Straight-Line Sequencing

➢PC – Program counter: hold the address of the next instruction to be executed

➢Straight line sequencing: If fetching and executing of instructions is carried out one by one from successive addresses of memory, it is called straight line sequencing.

➢Major two phase of instruction execution

- ❖ Instruction fetch phase: Instruction is fetched from memory and is placed in instruction register IR
- ❖Instruction execute phase: Contents of IR is decoded and processor carries out the operation either by reading data from memory or registers.

Instruction Execution And Straight-Line Sequencing

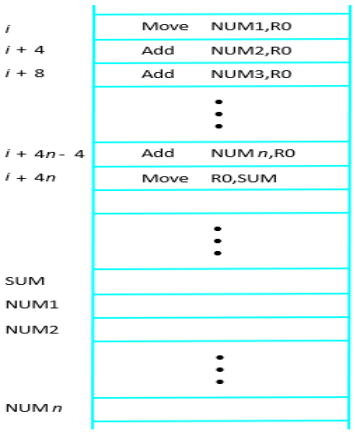


Figure 2.9. A straight-line program for adding n numbers.

DIGITAL DESIGN AND COMPUTER ORGANIZATION

5. Branching T2:Ch2 2.4

Department of Computer Science and Engineering

Branching

➢Branch instruction are those which changes the normal sequence of execution.

➢Sequence can be changed either conditionally or unconditionally.

➢Accordingly we have conditional branch instructions and unconditional branch instruction.

➢conditional branch instruction changes the sequence only when certain conditions are met.

➢Unconditional branch instruction changes the sequence of execution irrespective of condition of the results.

Branching

➢ Consider branch instruction in a program:

Branch > 0 Loop

➢ Program execution continues in a straight line sequence until encountering of **Branch > 0 Loop** instruction

➢ Execution results in a jump to location of Loop.

➢ That is a branch instruction loads a new value in PC - the memory location address (loop) at which program control is to be transferred to start/resume execution.

➢ As a result processor is prohibited from fetching and executing next instruction in straight – line sequence.

Branching

Branch target

Conditional branch

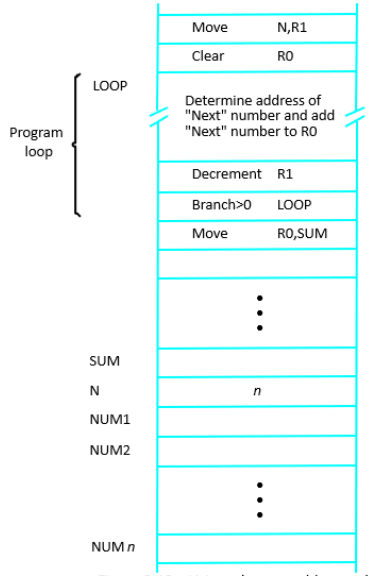


Figure 2.10. Using a loop to add *n* numbers.

DIGITAL DESIGN AND COMPUTER ORGANIZATION

6. Condition Codes T2:Ch2 2.4

Department of Computer Science and Engineering

Condition Codes

- The data conditions or status, after an arithmetic or logical operation are indicated by setting or clearing the flip – flops called *flags* or *condition codes*.
- Each flip-flop holding a data condition code is a one –bit storage cell (logic circuit) that can be set to a 1 or reset to 0 value.
- These condition codes are set/reset as a result of arithmetic and logical operations in the ALU.
- Condition code bits or flag bits are accommodated in a groups of 4 – bit, 8 bit or 16 – bit flag register or a status register in CPU.

Condition Codes

- Results of various instructions are stored for subsequent use by conditional instructions
- This is done by recording the required info in individual bits, called as condition code flags – grouped together in special processor register called as condition code register or status register

Condition Codes

- CONDITIONAL CODE FLAGS:

N – Negative	1 if results are Negative 0 if results are Positive
Z – Zero	1 if results are Zero 0 if results are Non zero
V – Overflow	1 if arithmetic overflow occurs 0 no overflow occurs
C – Carry	1 if carry from MSB bit 0 if there is no carry from MSB bit

Condition Codes

- Pentium processor makes use of following condition codes:

C (carry flag)
P (parity flag)
A (Auxiliary carry flag)
Z (zero flag)
S (sign flag)
O (over flow flag)

Condition Codes

List the steps needed to execute the machine instruction Add LOC, R0 in terms of transfers between memory and processor and some simple control commands. Assume that the instruction itself is stored in the memory at location INSTR and that this address is initially in register PC.

Condition Codes

- Transfer the contents of register PC to register MAR
- Issue a Read command to memory, and then wait until it has transferred the requested word into register MDR
- Transfer the instruction from MDR into IR and decode it
- Transfer the address LOCA from IR to MAR
- Issue a Read command and wait until MDR is loaded
- Transfer contents of MDR to the ALU
- Transfer contents of R0 to the ALU
- Perform addition of the two operands in the ALU and transfer result into R0
- Transfer contents of PC to ALU
- Add 1 to operand in ALU and transfer incremented address to PC

Condition Codes

Repeat the problem for machine instruction
Add R1,R2,R3

Condition Codes

- Transfer the contents of register PC to register MAR
- Issue a Read command to memory, and then wait until it has transferred the requested word into register MDR
- Transfer the instruction from MDR into IR and decode it
- Transfer contents of R1 and R2 to the ALU
- Perform addition of two operands in the ALU and transfer answer into R3
- Transfer contents of PC to ALU
- Add 1 to operand in ALU and transfer incremented address to PC

Condition Codes

Give a short sequence of machine instructions for the task: “Add the contents of memory location A to those of location B, and place the answer in location C.”

Instructions Load LOC, R_i

and Store R_i, LOC

are the only instructions available to transfer data between the memory and general purpose register R_i. Do not destroy the contents of either location A or B.

Condition Codes

Load A,R0
Load B,R1
Add R0,R1
Store R1,C

Condition Codes

Suppose that move and add instructions are available in the format
Move/Add loc1,loc2

Loci can be either memory or reg. Is it possible to use fewer instructions?
Give the seq

Condition Codes

Move B,C
Add A,C

DIGITAL DESIGN AND COMPUTER ORGANIZATION

7. Generating Memory Addresses T2:Ch2 2.4

Department of Computer Science and Engineering

Generating Memory Addresses

- The instruction block at **LOOP** needs to add a different number from a list during each iteration.
- The **Add** instruction must refer to a different memory address during each pass through the loop.
- The memory operand address cannot be directly given in a single **Add** instruction because it would require modification on each pass.
- A possible solution is to use a processor register, such as **Ri**, to hold the memory address.
- **Ri** can be initialized with the address **NUMI** before the loop and incremented by 4 after each iteration to point to the next memory address.
- This situation highlights the need for flexible ways to specify operand addresses.
- These methods are known as **addressing modes**, which vary in implementation but share common underlying concepts across different computers.

THANK YOU

Team DDCO
Department of Computer Science

DIGITAL DESIGN AND COMPUTER ORGANIZATION

Addressing Modes T2: Chapter 2: 2.5

Department of Computer Science and Engineering

Generating Memory Addresses

- Situations like program looping, branching demand convenient ways of locating memory words or memory operands.
- This convenience is being offered by different instructions (of a particular instruction set as defined by the Microprocessor) via various Addressing Modes they support.
- Suppose a given memory operand address is too large to fit into a given instruction format (i.e., in operand field), then it is the addressing mode (say Register Indirect Mode) that can resolve this problem.

Addressing Modes
Introduction

What is Addressing Mode . . . ?

- The different ways in which the location of an operand is specified in an instruction are referred to as *addressing modes*

Addressing Modes
Introduction

What is Addressing Mode . . . ?

- *The addressing mode specifies a rule or method for interpreting or modifying the address field of the instruction before the operand is actually accessed for manipulation.*
- *It **is concerned with the different ways in which the location of an operand can be specified in a given instruction.***
- *Various schemes for specifying addresses of operands in an instruction have been introduced. Such schemes are collectively known as Addressing Modes.*

Addressing Modes
Introduction

☺Advantages of Addressing modes :

- *To be able to reference large number of memory location in main memory or for some system in virtual memory.*
- *To reduce number of bits in the address field of an instruction*
- *To extend programming convenience to the user / programmer by providing such facilities as indexing of set of data values, counters for loop control, pointers to memory locations*

Addressing Modes
Outline

1. Immediate mode
2. Absolute (Direct) mode
3. Register mode
4. Indirect mode
5. Indexed mode
6. Relative mode
7. Auto increment mode
8. Auto decrement mode

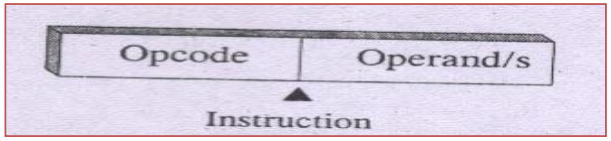
DIGITAL DESIGN AND COMPUTER ORGANIZATION

1. Immediate mode T2:Ch2 2.5

Department of Computer Science and Engineering

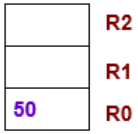
Immediate mode

➢ Here the operand is specified in the instruction itself. An instruction that follows immediate mode has an operand field rather than an address field.



For example:

Move $50_{immediate}$, R0



➢ A common convention say, a pound symbol # has to precede the value of an immediate operand

Move #50, R0

Immediate mode
Example

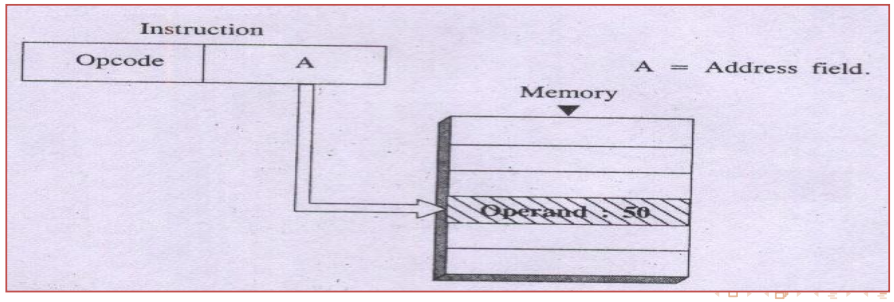
A=B+6
Move B,R1
Add #6,R1
Move R1,A

DIGITAL DESIGN AND COMPUTER ORGANIZATION

2. Absolute (Direct) mode T2:Ch2 2.5

Absolute (Direct) mode

➢ Here operand resides in Memory and its address is given explicitly in the address field of an instruction. This scheme need only one memory reference in addition to instruction fetch cycle and no further calculation is required to compute operand address.



Absolute (Direct) mode

➢ Direct addressing scheme is simple to use and easy to implement without the requirement for additional hardware.
➢However it offers only *limited memory address space*.

➢ Examples

Move LOC,R0
Load P
Store S

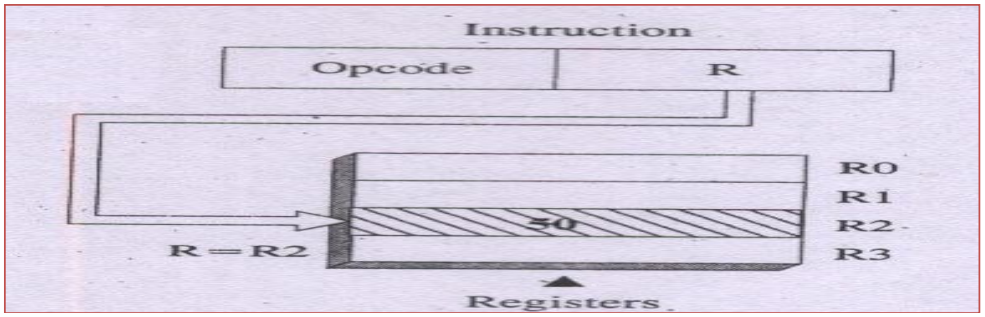
DIGITAL DESIGN AND COMPUTER ORGANIZATION

3. Register mode T2:Ch2 2.5

Register Addressing mode

➢ In this scheme, the operand is the contents of a register appearing in the instruction i.e A=R

➢ example
Add R1,R2



Register Addressing mode

Advantages of this scheme :

✓ No memory reference

✓ A few bit address to indicate register location

✓ Speedy execution since register is inside the processor & has low access time.

Disadvantages of this scheme :

✓ Limited address space as number of registers are less in many of the processors.

DIGITAL DESIGN AND COMPUTER ORGANIZATION

4. Indirect mode T2:Ch2 2.5

Indirect Addressing mode

➢ The effective address of the operand is the contents of a register or memory location whose address appears in the instruction

➢ By referring to this address (EA), the required operand can be fetched from memory.

➢ Example: **Add** (A), **R0** i.e. EA = (A) i.e. contents of **A** is **B**
Here **B** is effective address of desired operand in memory

Indirect Addressing mode

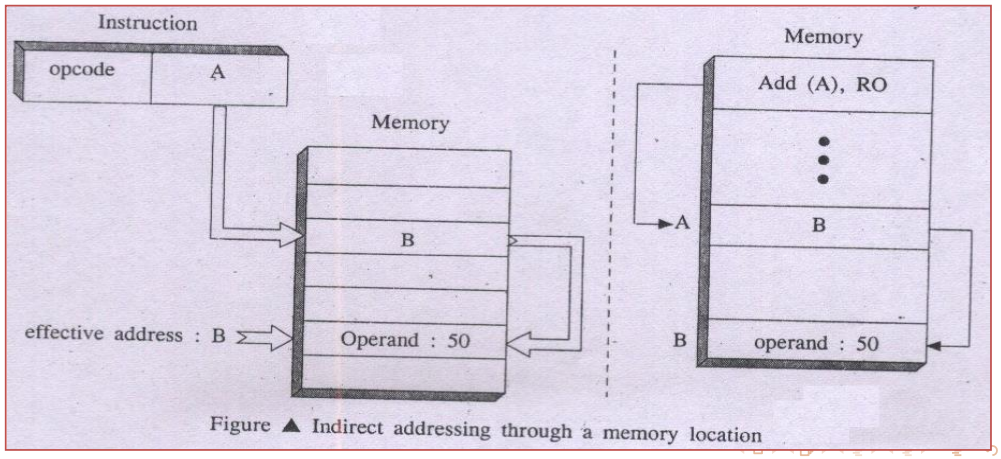


Figure ▲ Indirect addressing through a memory location

Register Indirect Addressing mode

- Here, instruction specifies a register in the CPU whose contents give the effective address of the operand in Memory.
- For example **Add (R1), R0 i.e. EA = (R1) i.e. contents of R1 is B**

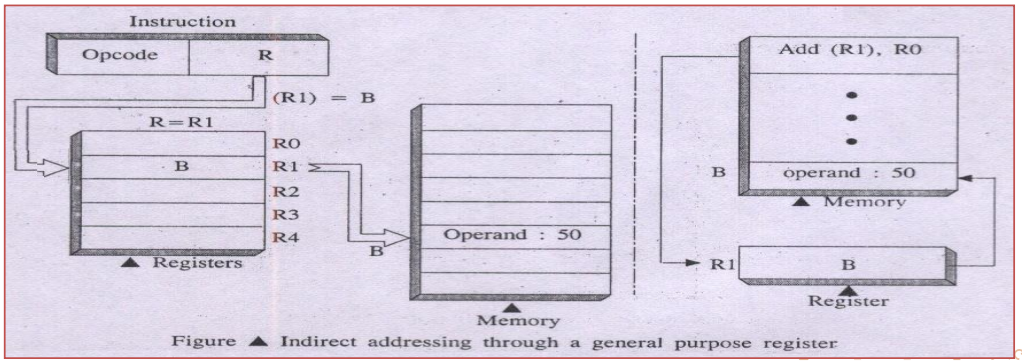


Figure ▲ Indirect addressing through a general purpose register

Indirect Addressing mode

Address	Contents		
	Move		N, R1
	Move		#NUM, R2
	Clear		R0
LOOP	Add		(R2), R0
	Add		#4, R2
	Decrement	R1	
	Branch > 0	LOOP	
	Move		R0, SUM

Indirect Addressing mode

The advantage of Register Indirect Addressing:

- It uses one less memory reference (memory read operation)
- Address field of the instruction uses a fewer bits to specify a register
- Register indirect addressing can be specified with Effective Address **EA** = (**R**) i.e. **B**

Advantages of Indirect addressing:

a wider address range to refer to a large number of memory locations.

Disadvantage of Indirect addressing:

2 or more memory references (memory read operations) required to fetch the desired operand in memory.

DIGITAL DESIGN AND COMPUTER ORGANIZATION

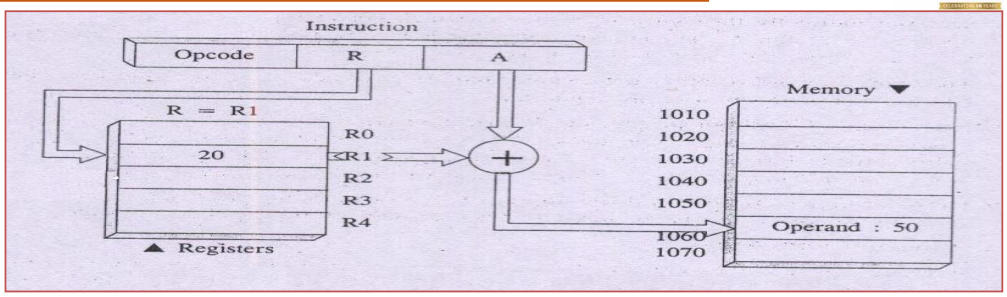
5. Indexed mode T2:Ch2 2.5

Department of Computer Science and Engineering

Indexed Addressing mode

- Here, effective address of the operand is generated by adding a constant value to the contents of a register.
- This constant value may be either an *offset value* called *displacement* or *beginning address of data/operand array* in main memory (Base).
- Indexed addressing mode is symbolically represented as **X(R)**
  - Here X denote a constant and **R** is name of the register involved in Indexing.
- Effective address EA of the operand is given as **EA = X + [R]**
  - Contents of index register are not changed during the process of address generation.

Indexed Addressing mode



A = 1040 = X; constant X corresponds to a memory location

(R) = 20 i.e. Index or offset value

EA = A + (R) = 1040 + 20

EA = 1060 at which you will find desired operand 50

Indexed Addressing mode

You can use indexed addressing mode in two ways

I) A constant value X defines here beginning address of operand in memory and index register Ri contains offset value (Displacement)

- For example 1040 in the following instruction:

Add 1040 (R1), R2	X=1040 R1=20 offset

II) A constant value X defines an offset and index register Ri contains beginning address of operand in memory.

- For example 20 in the following instruction:

Add 20(R1), R2	X=20 R1=1040

Indexed Addressing mode

- Index addressing is fast and is excellent for manipulating data structures such as arrays as all you need to do is set up a base address then use the index in your code to access individual elements.
- Another advantage of indexed addressing is that if the array is re-located in memory at any point then only the base address needs to be changed. The code making use of the index can remain exactly the same.

Indexed Addressing mode
Example

Example: Add 20(R1),R2

Add 2000(R3),R4

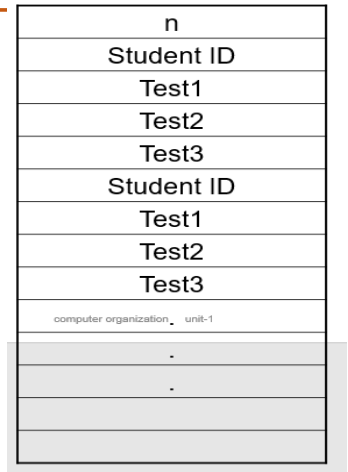
Assume that a list of scores of the student beginning at location LIST as shown in the following diagram

A 4 word memory comprises a record that stores relevant info for each student

Compute the sum of all scores obtained on each of the test and store these sums in memory locations SUM1,SUM2,SUM3

Indexed Addressing mode

N	5000
LIST	5004
LIST+4	5008
LIST+8	5012
LIST+12	5016
LIST+16	5020



Indexed Addressing mode

```
                Move            #LIST, R0
                Clear           R1
                Clear           R2
                Clear           R3
                Move            N, R4
LOOP            Add             4(R0), R1
                Add             8(R0), R2
                Add             12(R0), R3
                Add             #16, R0
                Decrement       R4
                Branch>0        LOOP
                Move            R1, SUM1
                Move            R2, SUM2
                Move            R3, SUM3
```

Indexed Addressing mode

❑ *Advantages of Index mode*

is the flexibility it offers to access relative memory locations

❑ *Disadvantages of Index mode*

* Is the complexity of computing effective address.

* The instruction requires to have two address fields

at least one of which is an explicit number.

Indexed Addressing mode

- several variations of this basic form provide a very efficient access to memory operands in practical programming situations.
- For example, a second register may be used to contain the offset X, in which case we can write the Index mode as

(Ri, Rj)

The effective address is the sum of the contents of registers Ri and Rj.
- The second register is usually called the base register.
- This form of indexed addressing provides more flexibility in accessing operands, because both components of the effective address can be changed.

Indexed Addressing mode

Another version of the Index mode uses two registers plus a constant, which can be denoted as

X(Ri, Rj)

In this case, the effective address is the sum of the constant X and the contents of registers Ri and Rj.

This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (Ri, Rj) part of the addressing mode.

DIGITAL DESIGN AND COMPUTER ORGANIZATION

6. Relative mode T2:Ch2 2.5

Department of Computer Science and Engineering

Relative mode

- This scheme supplies the relative position of the memory operand to be located.
- It's like index mode only but program counter register PC substitutes for base address contents
- Commonly used to specify the target address in branch instruction
- Relative Mode specify Effective Address by a notation:

  X(PC)
- Effective address is **EA = [PC] + X**

- **Branch > 0 loop** Here jump value / displacement **± X**

DIGITAL DESIGN AND COMPUTER ORGANIZATION

7. Auto increment mode T2:Ch2 2.5

Department of Computer Science and Engineering

Auto increment mode

- The EA of the operand is the contents of a register specified in the instruction
- After accessing the operand, the contents of this register are incremented automatically to point to the next operand in contiguous memory locations.
- Notation for Auto Increment Mode: **(R_i) +**
- For example: **Add (R2) + R0**
- Use of Auto Increment mode instruction eliminates the use of explicit increment instruction

Auto increment mode

Program to add n numbers using autoincrement

```
        MOVE N,R1
        MOVE #NUM1,R2
        CLEAR R0
LOOP    ADD (R2)+,R0
        DECREMENT R1
        BRANCH >0 LOOP
        MOVE R0,SUM
```

DIGITAL DESIGN AND COMPUTER ORGANIZATION

8. Auto decrement mode T2:Ch2 2.5

Department of Computer Science and Engineering

Auto decrement mode

- Here, content of a register specified in the instruction is decremented to denote effective address of next operand in successive memory location.
- Notation for Auto Decrement Mode: - (R_i)
- For instance

  Add – (R2), R0
- It allows accessing of operands in decreasing order of memory address..

Addressing Mode

- The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes.

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC
Indirect	(R*i*) (LOC)	EA = [R*i*] EA = [LOC]
Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Base with index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*]; Increment R*i*
Autodecrement	–(R*i*)	Decrement R*i*; EA = [R*i*]

THANK YOU

Team DDCO
Department of Computer Science

DIGITAL DESIGN AND COMPUTER ORGANIZATION

Machine Instructions and Programs

Department of Computer Science and Engineering

DIGITAL DESIGN AND COMPUTER ORGANIZATION

Machine Instructions and Programs T2:Ch2 2.6,2.7

Department of Computer Science and Engineering

Basic Structure of computers
Outline

- Assembly Languages
  - Assembly Language Syntax
  - Assembler Directives
  - Assembly and Execution of Programs
  - Number Notation
- I/O Operations

DIGITAL DESIGN AND COMPUTER ORGANIZATION

Assembly Language T2:Ch2 2.6

Department of Computer Science and Engineering

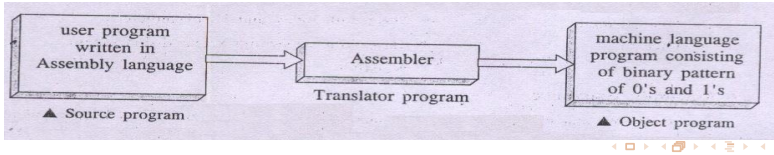
Machine Instructions and Programs
Assembly Language

- Here, symbolic codes are used to represent binary pattern of machine instructions. These symbolic codes are called as *mnemonics*.
- *Mnemonics* are abbreviations that represent operation code of an instruction in a compact and meaningful symbolic form.
- For instance mnemonics for few operation codes include:

INC / INR- Increment
ADDI - To add immediate operand
ADD - To add
LOAD - To load operand from memory
STORE- To store operand to memory
MOVE - To transfer data from one location to another location/Register.

Machine Instructions and Programs
Assembly Language

- A complete set of such mnemonics, symbolic names for register & memory locations and a list of rules for their use forms a programming language called an *Assembly Language*.
- The basic unit of assembly language program is a *line of code*.
- Here every line of code has symbolic code called *mnemonic* **opcode** and symbolic name to represent address of memory location / register as the ***operand field***.
- The translator program is called **an Assembler.**



Machine Instructions and Programs
Assembly Language Syntax

- For example, assembly language syntax suggests that a
- move instruction to appear as follows:
  - **MOVE R0, SUM**
- The opcode mnemonic **MOVE** is followed by at least one blank space.
- The source operand is in register R0 (register operand).
- The destination operand is in the memory location **SUM**
- For instance symbolic name of operand memory location **SUM** suggest **absolute mode** or **direct addressing mode** is used in the instruction.

Machine Instructions and Programs
Assembly Language Syntax

- Pound sign **#** usually denotes an ***immediate operand*** in the instruction using **Immediate Addressing mode**
- For example: **ADD #10, R2 or ADDI 10, R2**
  - **MOVE #20, R3**
- **Indirect addressing** is usually specified by placing parentheses around the name or symbol

For example **ADD (A), R0**

Format of Assembly Language Statements

LABEL	OPCODE	OPERAND(S)	; COMMENTS

Machine Instructions and Programs
Assembler Directives

- *Assembler Directives are the assembler commands to the assembler concerning the program being assembled. These commands are neither translated into machine opcode nor assigned any memory location in the object program.*
- Thus an assembly language program is said to be a complete one and acceptable to an assembler for translation and for further assignment of memory locations – if it is associated with necessary assembler directives.
- Examples of Assembler Directives:

  S EQU 150

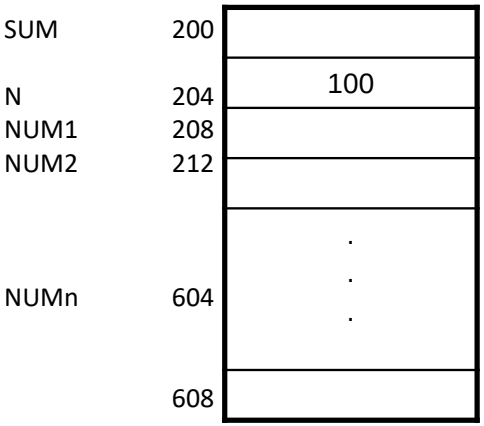
  - The assembler command **EQU** directs the assembler during translation that the symbolic name **S** must be replaced by value **150**,

Machine Instructions and Programs
Assembly Language

➢Memory arrangement for the program

```
100    Move N,R1
104    Move #NUM1,R2
108    Clear R0
112    Add (R2), R0
116    Add #4, R2
120    Decrement R1
124    Branch>0  LOOP
128    Move R0, SUM
```

Machine Instructions and Programs
Memory arrangement for the program



Machine Instructions and Programs
AL representation for the program

```
SUM     EQU       200
        ORIGIN    204
N       DATAWORD  100
NUM1    RESERVE   400
        ORIGIN    100
START   MOVE      N,R1
        MOVE      #NUM1,R2
        CLR R0
LOOP    ADD       (R2),R0
        ADD       #4,R2
        DEC       R1
        BGTZ      LOOP
        MOVE      R0,SUM
        RETURN
        END       START
```

Machine Instructions and Programs
Assembler Directives

- **ORIGIN 204**

- Instruct assembler to place data block at main memory locations starting from 204

- **N DATAWORD 100**

- Inform the assembler that value of N i.e. data value 100 is to be placed in the memory location 204.

- **ORIGIN 100**

- The second ORIGIN directive states that assembler directive must load machine instructions of the object program in the main memory starting from location 100.

Machine Instructions and Programs
Assembler Directives

- **NUM1 RESERVE 400**

This directive declares that a memory block of 400 bytes is to be reserved for data, and that the name NUM1 is to be associated with address 208

Machine Instructions and Programs
Assembler Directives

RETURN

- This is an assembler directive that identifies the point at which execution of the program should be terminated.
- It causes the assembler to insert an appropriate machine instruction that returns control to the operating system of the computer.

Machine Instructions and Programs
Assembler Directives

END START

The last statement in the source program is the assembler directive END, which tells the assembler that this is the end of the source program text.

The END directive includes the label START, which is the address of the location at which execution of the program is to begin.

Machine Instructions and Programs
Assembler Directives

How does an assembler directive differ from a regular instruction?

- Unlike regular instructions, which are translated into machine code and executed by the CPU, assembler directives are not converted into machine opcodes and are not assigned any memory location in the object program. They are essentially instructions to the assembler itself, rather than to the computer's processor.

Machine Instructions and Programs
ASSEMBLY AND EXECUTION OF PROGRAMS

What steps are involved in the assembly and execution of programs, and why might these steps be crucial for efficient program operation?

Machine Instructions and Programs
ASSEMBLY AND EXECUTION OF PROGRAMS

- A source program written in an assembly language must be assembled into a machine language object program before it can be executed -performed by assembler.
- As the assembler scans through a source programs, it keeps track of all names and the numerical values that correspond to them in a symbol table .

Machine Instructions and Programs
ASSEMBLY AND EXECUTION OF PROGRAMS

- When a name appears a second time, it is replaced with its value from the table .
- A problem arises when a name appears as an operand before it is given a value. For example, this happens if a forward branch is required.
- A simple solution to this problem is to have the assembler scan through the source program twice.
- During the first pass, it creates a complete symbol table. At the end of this pass, all names will have been assigned numerical values.
- The assembler then goes through the source program a second time and substitutes values for all names from the symbol table. Such an assembler is called a two-pass assembler.

Machine Instructions and Programs
ASSEMBLY AND EXECUTION OF PROGRAMS

- The assembler stores the object program on a magnetic disk.
- The object program must be loaded into the memory of the computer before it is executed.
- This task is performed by an utility program called a loader.

Machine Instructions and Programs
NUMBER NOTATION

ADD #93, R1
or
ADD #%01011101, R1
or
ADD #$5D, R1

DIGITAL DESIGN AND COMPUTER ORGANIZATION

I/O Operations T2:Ch2 2.7

Department of Computer Science and Engineering

Machine Instructions and Programs
BASIC INPUT/OUTPUT OPERATIONS

How is the interaction between the central processing unit (CPU) and input/output (I/O) devices is managed in terms of data transfer and control?

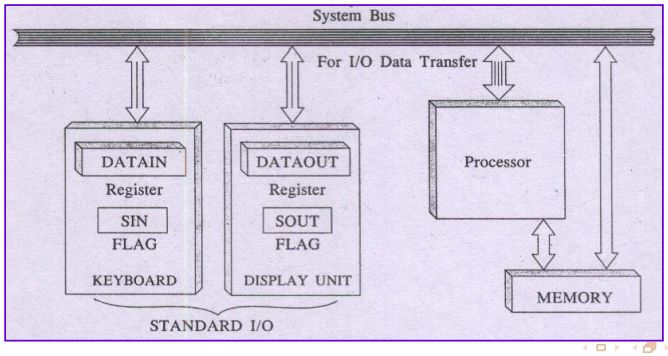
Machine Instructions and Programs
BASIC INPUT/OUTPUT OPERATIONS

Mechanism of I/O transfer
between processor and keyboard & video monitor:

- A character key when pressed from the keyboard its scan code is sent to an 8-bit buffer register DATAIN in the keyboard.
- Processor is informed about a valid character data presence in DATAIN register by setting a synchronization flag SIN to 1.
- I/O driver program continuously monitors contents of SIN flag, & when SIN is set to 1, it reads the contents of DATAIN.
- Thus character stored in DATAIN register is transferred to processor over a system bus and SIN content is automatically reset to 0.

Machine Instructions and Programs
ASSEMBLY AND EXECUTION OF PROGRAMS

Mechanism of I/O transfer
between Processor and Keyboard & Video monitor



Machine Instructions and Programs
ASSEMBLY AND EXECUTION OF PROGRAMS

- A similar set of events take place while transferring a character data from processor to the display screen.
- Here a **DATAOUT** register that holds a character's code to be displayed when synchronization control flag **SOUT** is set to **1**. When **SOUT** equals 1, the display device is ready to receive a character from processor.
- The transfer of a character to **DATAOUT** resets **SOUT** to **0**.
- I/O driver program instructions control the status of **SOUT** flag.

👉The buffer registers **DATAIN**, **DATAOUT**, and control flags **SIN**, **SOUT** in this hardware setup forms parts of a connectivity circuits commonly known as ***device interface*** *or* ***interface hardware.***

Machine Instructions and Programs
ASSEMBLY AND EXECUTION OF PROGRAMS

- I/O driver program instructions for I/O data transfer

```
READWAIT :  Branch to READWAIT if SIN  = 0
            Input from DATAIN to R0 (if SIN = 1)
```

```
WRITEWAIT   Branch to WRITEWAIT if SOUT = 0
            Output from R0 to DATAOUT (if SOUT = 1)
```

- **SOUT** is set to 1 when display terminal is free to display next character.
- The wait loop is executed repeatedly until the control flag **SOUT** is set to 1 by the display terminal.
- Initial state of SIN is 0 and SOUT is 1

Machine Instructions and Programs
ASSEMBLY AND EXECUTION OF PROGRAMS

- Memory mapped i/o – some memory addresses refer to peripheral device buffer registers, such as DATAIN,DATAOUT
- Data can be transferred b/w the cpu and these registers using same set of instructions and same status flags
- Ex: MoveByte DATAIN,R0
- MoveByte R0,DATAOUT
- Status flags SIN and SOUT can be included device status register
- Assume that the bit b3 of the status registers INSTATUS,OUTSTATUS corresponds to SIN and SOUT

Machine Instructions and Programs
ASSEMBLY AND EXECUTION OF PROGRAMS

Then read and write operation can be implemented as:

```
READWAIT    TestBit   #3,INSTATUS
            Branch=0 READWAIT
            MoveByte DATAIN,R0
```

```
WRITEWAIT  TestBit    #3,OUTSTATUS
                      Branch=0  WRITEWAIT
                       MoveByte  R0,DATAOUT
```

Machine Instructions and Programs
ASSEMBLY AND EXECUTION OF PROGRAMS

- Program to read a line of chars and to display it

```
            Move #LOC,R0
READ        TestBit  #3,INSTATUS
            Branch=0 READ
            MoveByte DATAIN,(R0)
ECHO        TestBit #3,OUTSTATUS
            Branch=0 ECHO
            MoveByte (R0),DATAOUT
            Compare #CR,(R0)+

            branch!=0 READ
```

Machine Instructions and Programs
ASSEMBLY AND EXECUTION OF PROGRAMS

	Move	#LOC,R0	Initialize pointer register R0 to point to the address of the first location in memory where the characters are to be stored.
READ	TestBit	#3,INSTATUS	Wait for a character to be entered
	Branch=0	READ	in the keyboard buffer DATAIN.
	MoveByte	DATAIN,(R0)	Transfer the character from DATAIN into the memory (this clears SIN to 0).
ECHO	TestBit	#3,OUTSTATUS	Wait for the display to become ready.
	Branch=0	ECHO	
	MoveByte	(R0),DATAOUT	Move the character just read to the display buffer register (this clears SOUT to 0).
	Compare	#CR,(R0)+	Check if the character just read is CR (carriage return). If it is not CR, then
	Branch≠0	READ	branch back and read another character. Also, increment the pointer to store the next character.

THANK YOU

Team DDCO
Department of Computer Science

DIGITAL DESIGN AND COMPUTER ORGANIZATION

Machine Instructions and Programs

Department of Computer Science and Engineering

DIGITAL DESIGN AND COMPUTER ORGANIZATION

Input/Output Organisation T2:Ch4 4.1,4.2 (except 4.2.4)

Department of Computer Science and Engineering

Basic Structure of computers
Outline

- Input/Output Organisation
  - Accessing I/O Devices
- Interrupts
  - Interrupt Hardware
  - Enabling and Disabling Interrupts
  - Handling Multiple Devices
  - Exceptions

DIGITAL DESIGN AND COMPUTER ORGANIZATION

Accessing I/O Devices T2:Ch4 4.1

Department of Computer Science and Engineering

Machine Instructions and Programs
Accessing I/O Devices

- Multiple I/O devices may be connected to the processor and the memory via a bus.
- Bus consists of three sets of lines to carry address, data and control signals.
- Each I/O device is assigned an unique address.
- To access an I/O device, the processor places the address on the address lines.
- The device recognizes the address, and responds to the control signals.

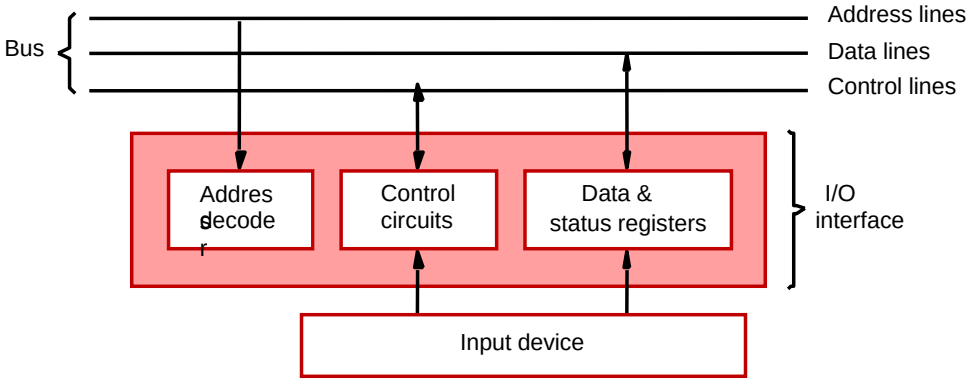
Machine Instructions and Programs
Accessing I/O Devices

- I/O devices and the memory may share the same address space:
  - Memory-mapped I/O.
  - Any machine instruction that can access memory can be used to transfer data to or from an I/O device.
  - Ex: Move DATAIN,R 1
    Move R2,DATAOUT
  - Simpler software.

Machine Instructions and Programs
Accessing I/O Devices

- I/O devices and the memory may have different address spaces:
  - Special instructions to transfer data to and from I/O devices.
  - I/O devices may have to deal with fewer address lines.
  - I/O address lines need not be physically separate from memory address lines.
  - In fact, address lines may be shared between I/O devices and memory, with a control signal to indicate whether it is a memory address or an I/O address.

Machine Instructions and Programs
Accessing I/O Devices



Machine Instructions and Programs
Accessing I/O Devices

- I/O device is connected to the bus using an I/O interface circuit which has:
  - Address decoder, control circuit, and data and status registers.
- Address decoder decodes the address placed on the address lines thus enabling the device to recognize its address.
- Data register holds the data being transferred to or from the processor.
- Status register holds information necessary for the operation of the I/O device.

Machine Instructions and Programs
Accessing I/O Devices

- Data and status registers are connected to the data lines, and have unique addresses.
- The Address decoder, the data & status register and the control circuitry required to coordinate I/O transfer constitute the Device's interface circuit

Machine Instructions and Programs
Accessing I/O Devices

- Recall that the rate of transfer to and from I/O devices is slower than the speed of the processor. This creates the need for mechanisms to synchronize data transfers between them.
- Program-controlled I/O:
  - Processor repeatedly monitors a status flag to achieve the necessary synchronization.
  - Processor polls the I/O device.

Machine Instructions and Programs
Accessing I/O Devices

- Two other mechanisms used for synchronizing data transfers between the processor and memory:
  - Interrupts.
  - Direct Memory Access.

Machine Instructions and Programs
Accessing I/O Devices

- Program-controlled I/O:
- A status flag i,e SIN and SOUT is included in the interface circuit as part of the status register.
- This flag is set to 1 when a char is entered at the keyboard and and set to 0 when the char is read by the processor .
- Same for SOUT.

Machine Instructions and Programs
Accessing I/O Devices

- To review the concept, let us consider a example of I/O operations involving a keyboard and a display device.
  - The 4 regs involved in the data transfer operations are : DATAIN, DATAOUT, STATUS, CONTROL
  - STATUS reg contains two control flags SIN and SOUT and two flags KIRQ, DIRQ which are used in conjunction with interrupts

Machine Instructions and Programs
Accessing I/O Devices

```
                Move            #line,R0
WAITK           Testbit         #0,status
                Branch=0        WAITK
                Move
                DATAIN,R1
WAITD           Testbit         #1,status
                Branch=0        WAITD
                Move            R1,
DATAOUT
                Move            R1,(R0)+
                Compare         #$0D, R1
                Branch!= 0      WAITK
                CALL            PROCESS
```

DIGITAL DESIGN AND COMPUTER ORGANIZATION

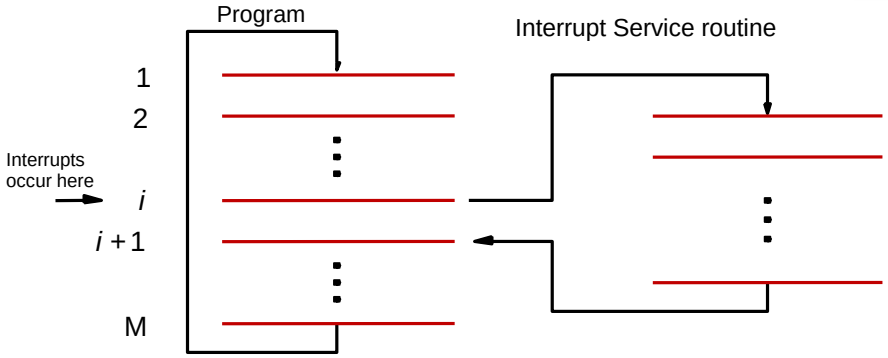
Interrupts T2:Ch4 4.2

Department of Computer Science and Engineering

Machine Instructions and Programs
Interrupts

- In program-controlled I/O, when the processor continuously monitors the status of the device, it does not perform any useful tasks.
- An alternate approach would be for the I/O device to alert the processor when it becomes ready.
  - Do so by sending a hardware signal called an interrupt to the processor.
  - At least one of the bus control lines, called an interrupt-request line is dedicated for this purpose.
- Processor can perform other useful tasks while it is waiting for the device to be ready.

Machine Instructions and Programs
Interrupts



Machine Instructions and Programs
Interrupts

- Processor is executing the instruction located at address i when an interrupt occurs.
- Routine executed in response to an interrupt request is called the interrupt-service routine.
- When an interrupt occurs, control must be transferred to the interrupt service routine.

Machine Instructions and Programs
Interrupts

- But before transferring control, the current contents of the PC (i+1), must be saved in a known location.
- This will enable the return-from-interrupt instruction to resume execution at i+1.
- Return address, or the contents of the PC are usually stored on the processor stack.

Machine Instructions and Programs
Comparision of Interrupts and Subroutine

- Treatment of an interrupt-service routine is very similar to that of a subroutine.

  However there are significant differences:
  - A subroutine performs a task that is required by the calling program.
  - Interrupt-service routine may not have anything in common with the program it interrupts.
  - Interrupt-service routine and the program that it interrupts may belong to different users.

Machine Instructions and Programs
Interrupt latency

- Saving and restoring information can be done automatically by the processor
- Saving and restoring registers involves memory transfers:
  - Increases the total execution time.
  - Increases the delay between the time an interrupt request is received, and the start of execution of the interrupt-service routine. This delay is called interrupt latency.
- In order to reduce the interrupt latency, most processors save only the minimal amount of information:
  - This minimal amount of information includes Program Counter and processor status registers.
- Any additional information that must be saved, must be saved explicitly by the program instructions at the beginning of the interrupt service routine.

Machine Instructions and Programs
Interrupt Enable & Disable

- Interrupt-requests interrupt the execution of a program, and may alter the intended sequence of events:
  - Sometimes such alterations may be undesirable, and must not be allowed.
- Processors generally provide the ability to enable and disable such interruptions as desired.
- One simple way is to provide machine instructions such as *Interrupt-enable* and *Interrupt-disable* for this purpose.

Machine Instructions and Programs
Avoidance of successive interrupts

- The interrupt request signal will be active until it learns that the processor has responded to its request. It is essential to ensure that this active signal does not lead successive interrupts.
- Several mechanism are available to solve this problem:
- The interrupt be disabled/enabled in the interrupt-service routine.

Machine Instructions and Programs
Avoidance of successive interrupts

- The processor automatically disable interrupts before starting the execution of the interrupt-service routine.
- The processor has a special interrupt request line through which it receives only one interrupt request regardless how long the line is activated.

Machine Instructions and Programs
Interrupts

The sequence of events involved in handling an interrupt request from a single device :

1. The device raises an IR
2. The processor interrupt the program currently being executed.
3. Interrupts are disabled by changing bit in PS
4. The device is informed that its request has been recognized & in response, it deactivates the IR signal
5. The action requested by the interrupt is performed by ISR
6. Interrupts are enabled & execution of the interrupted program is resumed.

Machine Instructions and Programs
Handling Multiple Devices

- Multiple I/O devices may be connected to the processor and the memory via a bus. Some or all of these devices may be capable of generating interrupt requests.
  - Each device operates independently, and hence no definite order can be imposed on how the devices generate interrupt requests?
- How does the processor know which device has generated an interrupt?
- How does the processor know which interrupt service routine needs to be executed?
- When the processor is executing an interrupt service routine for one device, can other device interrupt the processor?
- If two interrupt-requests are received simultaneously, then how to break the tie?

Machine Instructions and Programs
Handling Multiple Devices

- When a request is received over the common interrupt-request line, additional information is needed to identify the particular device that activated the line.
- The additional information is available in its status register. KIRQ & DIRQ.
- Then ISR polls all the I/O devices connected to the bus.
- The first device with its IRQ bit set is the device that should be serviced.

Machine Instructions and Programs
Handling Multiple Devices

- An appropriate subroutine is called to provide the request service.
- The polling scheme is easy to implement , but it spends time in interrogating the IRQ bits of all the devices that may not be requesting any service.
- An alternative approach is to use vectored interrupts.

Machine Instructions and Programs
Vectored Interrupts

- To reduce the time involved in polling process, a device requesting an interrupt may identify itself directly to the processor.
- Then the processor can immediately start executing the corresponding ISR.
- A device requesting an interrupt can identify itself by sending a special code to the processor over the bus.
- The code supplied by the device may represent the starting address of the ISR for that device.

Machine Instructions and Programs
Vectored Interrupts

- This arrangement implies that the ISR for a given device must always start at the same location.
- Flexibility can be provided by storing the starting address of the ISR in the location pointed to by the interrupting device
- The processor reads this address, called the ***interrupt vector*** and loads it into the PC.

Machine Instructions and Programs
Vectored Interrupts

- The interrupt vector may also include a new value for the processor status register.
- When a device generates an interrupt request, the processor may not be ready to receive the interrupt_vector code immediately.
- Ex. It must first complete the execution of current instruction which may require the use of bus

Machine Instructions and Programs
Vectored Interrupts

- The interrupting device must wait to put data on the bus only when the processor is ready to receive it
- When the processor is ready to receive the interrupt_vector code, it activates the interrupt_acknowledgement line, INTA.
- The I/O device responds by sending the interrupt_vector code and de-activating the INTR signal.

Machine Instructions and Programs
Interrupt Nesting

- The interrupts should be disabled during the execution of an ISR, to ensure that a request from one device will not cause more than one interruption.
- This may delay in responding to an interrupt request from other devices.
- For some devices, a long delay may lead to erroneous operation.

  Ex. Processor Clock.

Machine Instructions and Programs
Interrupt Nesting

- This example suggests that I/O devices should be organized in a priority structure.
- An interrupt request from a high priority device should be accepted while the processor is servicing another request from a lower priority device.
- To implement this scheme , we can assign a priority level to the processor that can be changed under program control.

Machine Instructions and Programs
Interrupt Nesting

- The priority level of the processor is the priority of the program that is currently being executed.
- This action disables interrupt from devices at the same level of priority or lower.
- However, interrupt requests from higher priority devices will continue to be accepted.

Machine Instructions and Programs
Interrupt Nesting

- The processor's priority is usually encoded in a few bits of the processor status word.
- This can be changed by privileged instructions, which are executed only while the processor is serving in the supervisor mode.

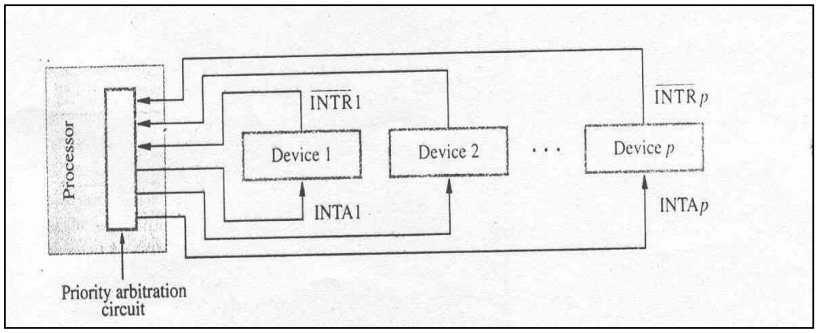
Machine Instructions and Programs
Interrupt Nesting

- The processor is in the supervisor mode only when executing operating system routines.
- It switches to the user mode before beginning to execute application programs.

Machine Instructions and Programs
Interrupt Nesting

- If a user program accidentally or intentionally, changes the priority of the processor and disrupt the system operation, will lead to a special type of interrupt called a ***privilege exception.***
- Implementation of interrupt priority using individual interrupt-request & acknowledgement lines is as shown

Machine Instructions and Programs
Interrupt Nesting



Machine Instructions and Programs
Simultaneous Requests

- Consider the problem of simultaneous arrivals of interrupt requests from two or more devices.
- The processor must have some means of deciding which request to service first.
- If all the devices have individual interrupt-request lines then using a priority scheme, the processor accepts the request having the highest priority.

Machine Instructions and Programs
Simultaneous Requests

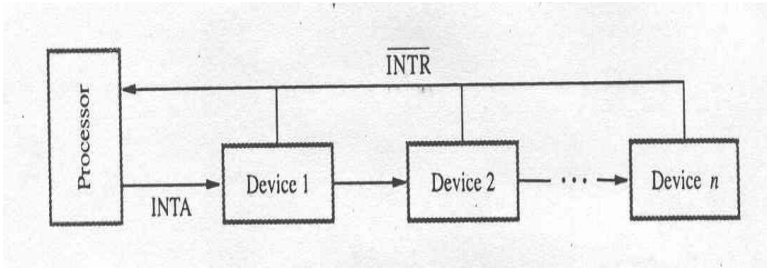
- If all the devices share a common interrupt-request line, some other mechanism is needed.

  1. Polling the status registers of the I/O devices, in which priority is determined by the order in which the devices are polled.

  2. In vectored interrupts, we must ensure that only one device is selected to read its vectored interrupt code.

Machine Instructions and Programs
Simultaneous Requests

- For this a widely used scheme is to connect the devices to form a ***daisy chain*** as shown



Machine Instructions and Programs
Simultaneous Requests

- the interrupt request line INTR is common to all devices
- the interrupt acknowledge line INTA is connected in a daisy chain fashion , such that the INTA signal propagates serially through the devices
- When INTR is active the processor responds by setting INTA to 1

Machine Instructions and Programs
Simultaneous Requests

- This signal is received by device1 , and passes it to device2 only if it does not require any service.
- If the Device 1 has pending request for interrupt, it blocks the INTA signal and proceeds to put its identifying code on the data lines
- The device closest to cpu has higher priority

Machine Instructions and Programs
Simultaneous Requests

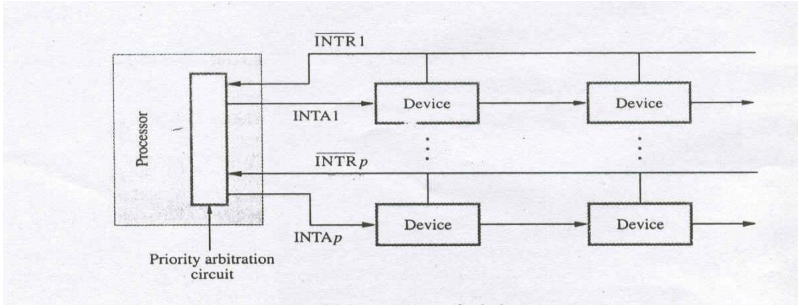
- The interrupt priority scheme is combined with daisy chain scheme as shown below

- Devices are organized in groups and each group is connected at a different priority level.

- Within a group devices are inter-connected in a daisy chain

Machine Instructions and Programs
Simultaneous Requests

- Arrangement of priority groups as shown



Machine Instructions and Programs
Exceptions

- Interrupts caused by interrupt-requests sent by I/O devices.
- Interrupts could be used in many other situations where the execution of one program needs to be suspended and execution of another program needs to be started.
- In general, the term exception is used to refer to any event that causes an interruption.
  - Interrupt-requests from I/O devices is one type of an exception.
- Other types of exceptions are:
  - Recovery from errors
  - Debugging
  - Privilege exception

Machine Instructions and Programs
Exceptions

- Many sources of errors in a processor. For example:
  - Error in the data stored.
  - Error during the execution of an instruction.
- When such errors are detected, exception processing is initiated.
  - Processor takes the same steps as in the case of I/O interrupt-request.
  - It suspends the execution of the current program, and starts executing an exception-service routine.

Machine Instructions and Programs
Exceptions

- Difference between handling I/O interrupt-request and handling exceptions due to errors:
  - In case of I/O interrupt-request, the processor usually completes the execution of an instruction in progress before branching to the interrupt-service routine.
  - In case of exception processing however, the execution of an instruction in progress usually cannot be completed.

Machine Instructions and Programs
Exceptions

- Debugger uses exceptions to provide important features:
  - Trace,
  - Breakpoints.
- Trace mode:
  - Exception occurs after the execution of every instruction.
  - Debugging program is used as the exception-service routine.

Machine Instructions and Programs
Exceptions

- Breakpoints:
  - Exception occurs only at specific points selected by the user.
  - Debugging program is used as the exception-service routine.

Machine Instructions and Programs
Exceptions

- Certain instructions can be executed only when the processor is in the supervisor mode. These are called privileged instructions.
- If an attempt is made to execute a privileged instruction in the user mode, a privilege exception occurs.
- Privilege exception causes:
  - Processor to switch to the supervisor mode,
  - Execution of an appropriate exception-servicing routine.

THANK YOU

Team DDCO
Department of Computer Science

DIGITAL DESIGN AND COMPUTER ORGANIZATION
Machine Instructions and Programs

Department of Computer Science and Engineering

DIGITAL DESIGN AND COMPUTER ORGANIZATION

Standard Input and Output

Department of Computer Science and Engineering

Basic Structure of computers
Outline

- Standard I/O Interfaces
  - Peripheral Component Interconnect Bus
  - SCSI Bus
  - Universal Serial Bus

DIGITAL DESIGN AND COMPUTER ORGANIZATION

Standard I/O Interfaces T2:Ch4 4.7

Department of Computer Science and Engineering

Standard Input and Output
Standard I/O Interfaces

- I/O device is connected to a computer using an interface circuit.
- Do we have to design a different interface for every combination of an I/O device and a computer?
- A practical approach is to develop standard interfaces and protocols.
- A personal computer has:
  - A motherboard which houses the processor chip, main memory and some I/O interfaces.
  - A few connectors into which additional interfaces can be plugged.
- Processor bus is defined by the signals on the processor chip.
  - Devices which require high-speed connection to the processor are connected directly to this bus.

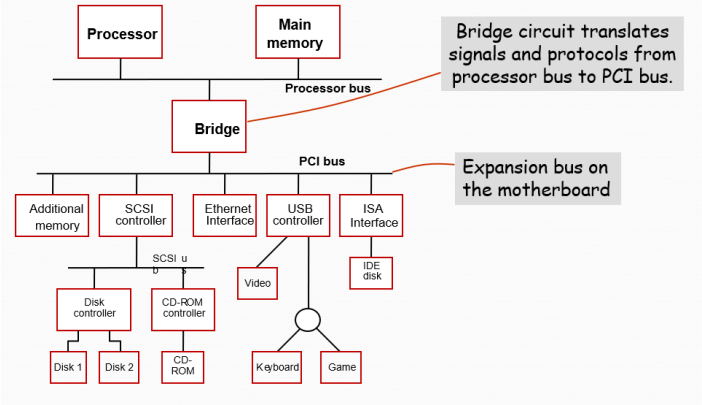
Standard Input and Output
Standard I/O Interfaces

- Because of electrical reasons only a few devices can be connected directly to the processor bus.
- Motherboard usually provides another bus that can support more devices.
  - Processor bus and the other bus (called as expansion bus) are interconnected by a circuit called “bridge”.
  - Devices connected to the expansion bus experience a small delay in data transfers.
- Design of a processor bus is closely tied to the architecture of the processor.
  - No uniform standard can be defined.
- Expansion bus however can have uniform standard defined.

Standard Input and Output
Standard I/O Interfaces

- A number of standards have been developed for the expansion bus.
  - Some have evolved by default.
  - For example, IBM’s Industry Standard Architecture.
- Three widely used bus standards:
  - PCI (Peripheral Component Interconnect)
  - SCSI (Small Computer System Interface)
  - USB (Universal Serial Bus)

Standard Input and Output
Standard I/O Interfaces



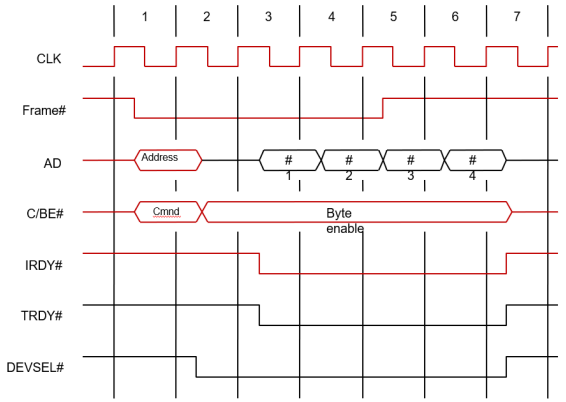
Standard Input and Output
PCI BUS

- *Peripheral Component Interconnect*
- Introduced in 1992
- Low-cost bus
- Processor independent
- Plug-and-play capability
- In today’s computers, most memory transfers involve a burst of data rather than just one word. The PCI is designed primarily to support this mode of operation.

Standard Input and Output
PCI BUS

- The bus supports three independent address spaces: memory, I/O, and configuration.
- we assumed that the master maintains the address information on the bus until data transfer is completed. But, the address is needed only long enough for the slave to be selected. Thus, the address is needed on the bus for one clock cycle only, freeing the address lines to be used for sending data in subsequent clock cycles. The result is a significant cost reduction.
- A master is called an initiator in PCI terminology. The addressed device that responds to read and write commands is called a target.

Standard Input and Output
PCI Bus



A read operation on the PCI bus

Standard Input and Output
PCI BUS – Device Configuration

- When an I/O device is connected to a computer, several actions are needed to configure both the device and the software that communicates with it.
- PCI incorporates in each I/O device interface a small configuration ROM memory that stores information about that device.
- The configuration ROMs of all devices are accessible in the configuration address space. The PCI initialization software reads these ROMs and determines whether the device is a printer, a keyboard, an Ethernet interface, or a disk controller. It can further learn bout various device options and characteristics.

Standard Input and Output
PCI BUS – Device Configuration

- Devices are assigned addresses during the initialization process.
- This means that during the bus configuration operation, devices cannot be accessed based on their address, as they have not yet been assigned one.
- Hence, the configuration address space uses a different mechanism. Each device has an input signal called Initialization Device Select, IDSEL#
- Electrical characteristics:
- PCI bus has been defined for operation with either a 5 or 3.3 V power supply.

Standard Input and Output
SCSI Bus

- The acronym SCSI stands for Small Computer System Interface.
- It refers to a standard bus defined by the American National Standards Institute (ANSI) under the designation X3.131 .
- In the original specifications of the standard, devices such as disks are connected to a computer via a 50-wire cable, which can be up to 25 meters in length and can transfer data at rates up to 5 megabytes/s.
- The SCSI bus standard has undergone many revisions, and its data transfer capability has increased very rapidly, almost doubling every two years.
- SCSI-2 and SCSI-3 have been defined, and each has several options.
- Because of various options SCSI connector may have 50, 68 or 80 pins.

Standard Input and Output
SCSI Bus

- Devices connected to the SCSI bus are not part of the address space of the processor
- The SCSI bus is connected to the processor bus through a SCSI controller. This controller uses DMA to transfer data packets from the main memory to the device, or vice versa.
- A packet may contain a block of data, commands from the processor to the device, or status information about the device.
- A controller connected to a SCSI bus is one of two types – an initiator or a target.

Standard Input and Output
SCSI Bus

- An initiator has the ability to select a particular target and to send commands specifying the operations to be performed. The disk controller operates as a target. It carries out the commands it receives from the initiator.
- The initiator establishes a logical connection with the intended target.
- Once this connection has been established, it can be suspended and restored as needed to transfer commands and bursts of data.
- While a particular connection is suspended, other device can use the bus to transfer information.
- This ability to overlap data transfer requests is one of the key features of the SCSI bus that leads to its high performance.

Standard Input and Output
SCSI Bus

- Data transfers on the SCSI bus are always controlled by the target controller.
- To send a command to a target, an initiator requests control of the bus and, after winning arbitration, selects the controller it wants to communicate with and hands control of the bus over to it.
- Then the controller starts a data transfer operation to receive a command from the initiator.

Standard Input and Output
SCSI Bus

- Data transfers on the SCSI bus are always controlled by the target controller.
- To send a command to a target, an initiator requests control of the bus and, after winning arbitration, selects the controller it wants to communicate with and hands control of the bus over to it.
- Then the controller starts a data transfer operation to receive a command from the initiator.

Standard Input and Output
SCSI Bus

Assume that processor needs to read block of data from a disk drive and that data are stored in disk sectors that are not contiguous.

The processor sends a command to the SCSI controller, which causes the following sequence of events to take place:

1. The SCSI controller, acting as an initiator, contends for control of the bus.
2. When the initiator wins the arbitration process, it selects the target controller and hands over control of the bus to it.
3. The target starts an output operation (from initiator to target); in response to this, the initiator sends a command specifying the required read operation.

Standard Input and Output
SCSI Bus

4. The target, realizing that it first needs to perform a disk seek operation, sends a message to the initiator indicating that it will temporarily suspend the connection between them. Then it releases the bus.
5. The target controller sends a command to the disk drive to move the read head to the first sector involved in the requested read operation. Then, it reads the data stored in that sector and stores them in a data buffer. When it is ready to begin transferring data to the initiator, the target requests control of the bus. After it wins arbitration, it reselects the initiator controller, thus restoring the suspended connection.

Standard Input and Output
SCSI Bus

6. The target transfers the contents of the data buffer to the initiator and then suspends the connection again
7. The target controller sends a command to the disk drive to perform another seek operation. Then, it transfers the contents of the second disk sector to the initiator as before. At the end of this transfers, the logical connection between the two controllers is terminated.
8. As the initiator controller receives the data, it stores them into the main memory using the DMA approach.
9. The SCSI controller sends as interrupt to the processor to inform it that the requested operation has been completed

Operation of SCSI bus from H/W point of view

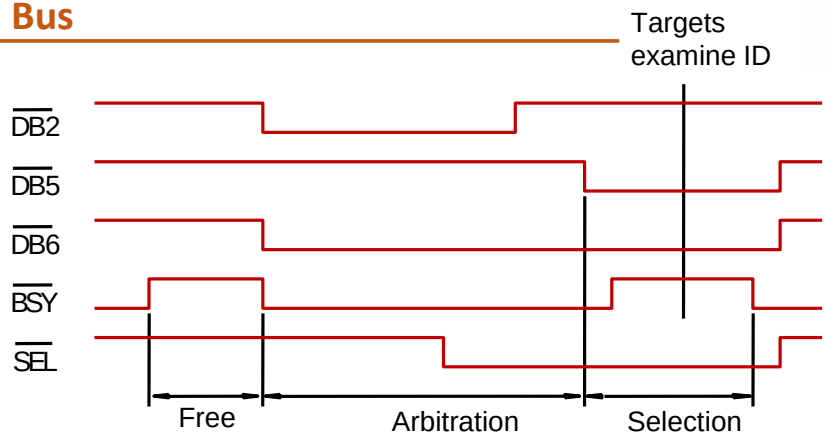
Category	Name	Function
Data	– DB(0) to – DB(7)	Data lines: Carry one byte of information during the information transfer phase and identify device during arbitration, selection and reselection phases
	– DB(P)	Parity bit for the data bus
Phase	– BSY	Busy: Asserted when the bus is not free
	– SEL	Selection: Asserted during selection and reselection
Information type	– C/D	Control/Data: Asserted during transfer of control information (command, status or message)
	– MSG	Message: indicates that the information being transferred is a message

Standard Input and Output
SCSI Bus

Table 4. The SCSI bus signals.(*cont.*)

Category	Name	Function
Handshake	– REQ	Request: Asserted by a target to request a data transfer cycle
	– ACK	Acknowledge: Asserted by the initiator when it has completed a data transfer operations
Direction of transfer	– I/O	Input/Output: Asserted to indicate an input operation (relative to the initiator)
Other	– ATN	Attention: Asserted by an initiator when it wishes to send a message to a target
	– RST	Reset: Causes all device controls to disconnect from the bus and assume their start-up state

Standard Input and Output
SCSI Bus

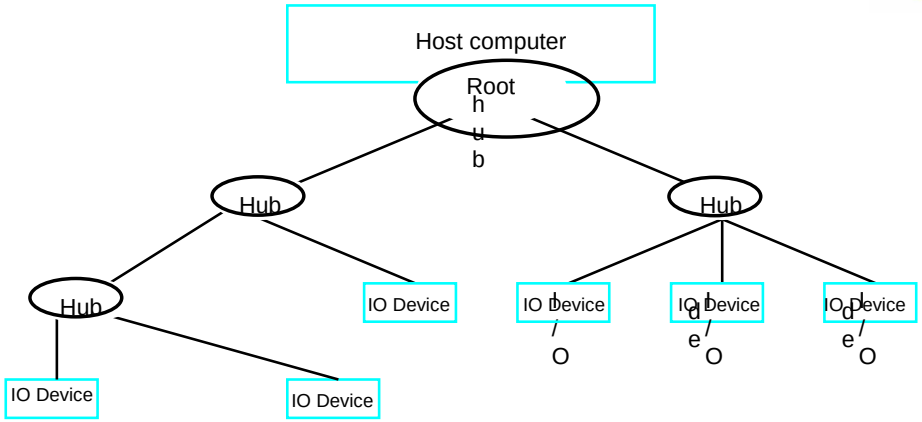


Standard Input and Output
USB

- Universal Serial Bus (USB) is an industry standard developed through a collaborative effort of several computer and communication companies, including Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, Nortel Networks, and Philips.

- Speed
  - Low-speed(1.5 Mb/s)
  - Full-speed(12 Mb/s)
  - High-speed(480 Mb/s)
- Port Limitation
- Device Characteristics
- Plug-and-play

Standard Input and Output
USB



Standard Input and Output
USB

- To accommodate a large number of devices that can be added or removed at any time, the USB has the tree structure as shown in the figure.
- Each node of the tree has a device called a hub, which acts as an intermediate control point between the host and the I/O devices. At the root of the tree, a root hub connects the entire tree to the host computer. The leaves of the tree are the I/O devices being served (for example, keyboard, Internet connection, speaker, or digital TV)
- In normal operation, a hub copies a message that it receives from its upstream connection to all its downstream ports. As a result, a message sent by the host computer is broadcast to all I/O devices, but only the addressed device will respond to that message. However, a message from an I/O device is sent only upstream towards the root of the tree and is not seen by other devices. Hence, the USB enables the host to communicate with the I/O devices, but it does not enable these devices to communicate with each other.

Standard Input and Output
USB Addessing

- When a USB is connected to a host computer, its root hub is attached to the processor bus, where it appears as a single device. The host software communicates with individual devices attached to the USB by sending packets of information, which the root hub forwards to the appropriate device in the USB tree.
- Each device on the USB, whether it is a hub or an I/O device, is assigned a 7-bit address. This address is local to the USB tree and is not related in any way to the addresses used on the processor bus.

Standard Input and Output
USB Addessing

- A hub may have any number of devices or other hubs connected to it, and addresses are assigned arbitrarily. When a device is first connected to a hub, or when it is powered on, it has the address 0. The hardware of the hub to which this device is connected is capable of detecting that the device has been connected, and it records this fact as part of its own status information. Periodically, the host polls each hub to collect status information and learn about new devices that may have been added or disconnected.
- When the host is informed that a new device has been connected, it uses a sequence of commands to send a reset signal on the corresponding hub port, read information from the device about its capabilities, send configuration information to the device, and assign the device a unique USB address. Once this sequence is completed the device begins normal operation and responds only to the new address.

Standard Input and Output
USB Protocols

- All information transferred over the USB is organized in packets, where a packet consists of one or more bytes of information. There are many types of packets that perform a variety of control functions.
- The information transferred on the USB can be divided into two broad categories: control and data.
- Control packets perform such tasks as addressing a device to initiate data transfer, acknowledging that data have been received correctly, or indicating an error.
- Data packets carry information that is delivered to a device.

Standard Input and Output
USB Protocols

- A packet consists of one or more fields containing different kinds of information. The first field of any packet is called the packet identifier, PID, which identifies the type of that packet.
- They are transmitted twice. The first time they are sent with their true values, and the second time with each bit complemented
- The four PID bits identify one of 16 different packet types. Some control packets, such as ACK (Acknowledge), consist only of the PID byte.

Standard Input and Output
USB Protocols

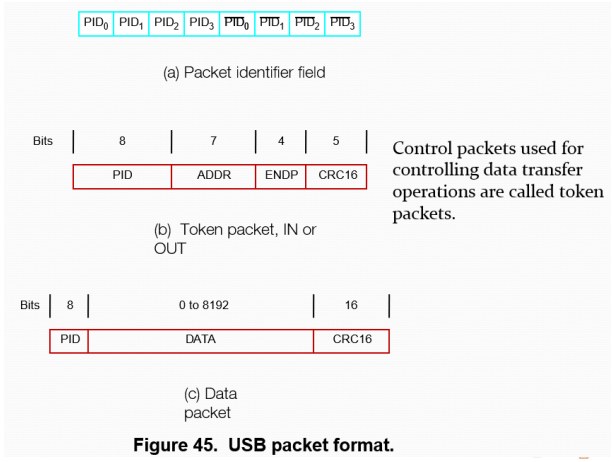


Figure 45. USB packet format.

Standard Input and Output
USB Protocols

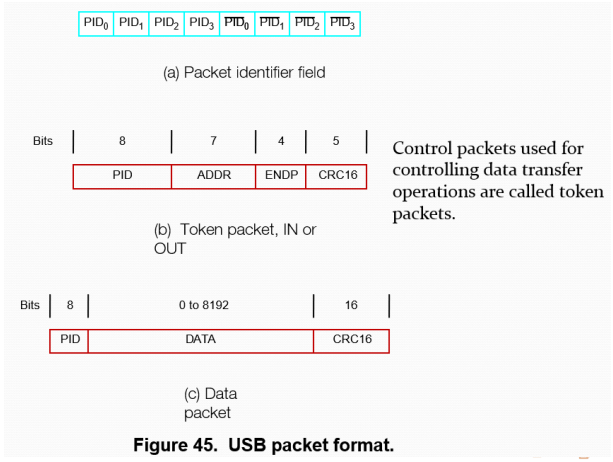


Figure 45. USB packet format.

Standard Input and Output
Inochronous Traffic on USB

- One of the key objectives of the USB is to support the transfer of isochronous data.
- Devices that generates or receives isochronous data require a time reference to control the sampling process.
- To provide this reference. Transmission over the USB is divided into frames of equal length.
- A frame is 1ms long for low-and full-speed data.
- The root hub generates a Start of Frame control packet (SOF) precisely once every 1 ms to mark the beginning of a new frame.

Standard Input and Output
Inochronous Traffic on USB

- The arrival of an SOF packet at any device constitutes a regular clock signal that the device can use for its own purposes.
- To assist devices that may need longer periods of time, the SOF packet carries an 11-bit frame number.
- Following each SOF packet, the host carries out input and output transfers for isochronous devices.
- This means that each device will have an opportunity for an input or output transfer once every 1 ms.

Standard Input and Output
Electrical Characterestics of USB

- The cables used for USB connections consist of four wires.
- Two are used to carry power, +5V and Ground.
- Thus, a hub or an I/O device may be powered directly from the bus, or it may have its own external power connection.
- The other two wires are used to carry data.
- Different signaling schemes are used for different speeds of transmission.
- At low speed, 1s and 0s are transmitted by sending a high voltage state (5V) on one or the other o the two signal wires. For high-speed links, differential transmission is used.

THANK YOU

Team DDCO
Department of Computer Science