- CT101 -
Computing Systems

Dr. Frank Glavin

Room 404, IT Building

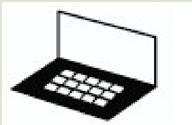
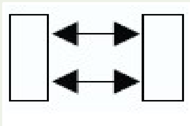
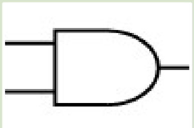
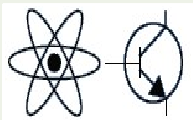
Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Introduction to Computing Systems

Computing Systems in a Nutshell

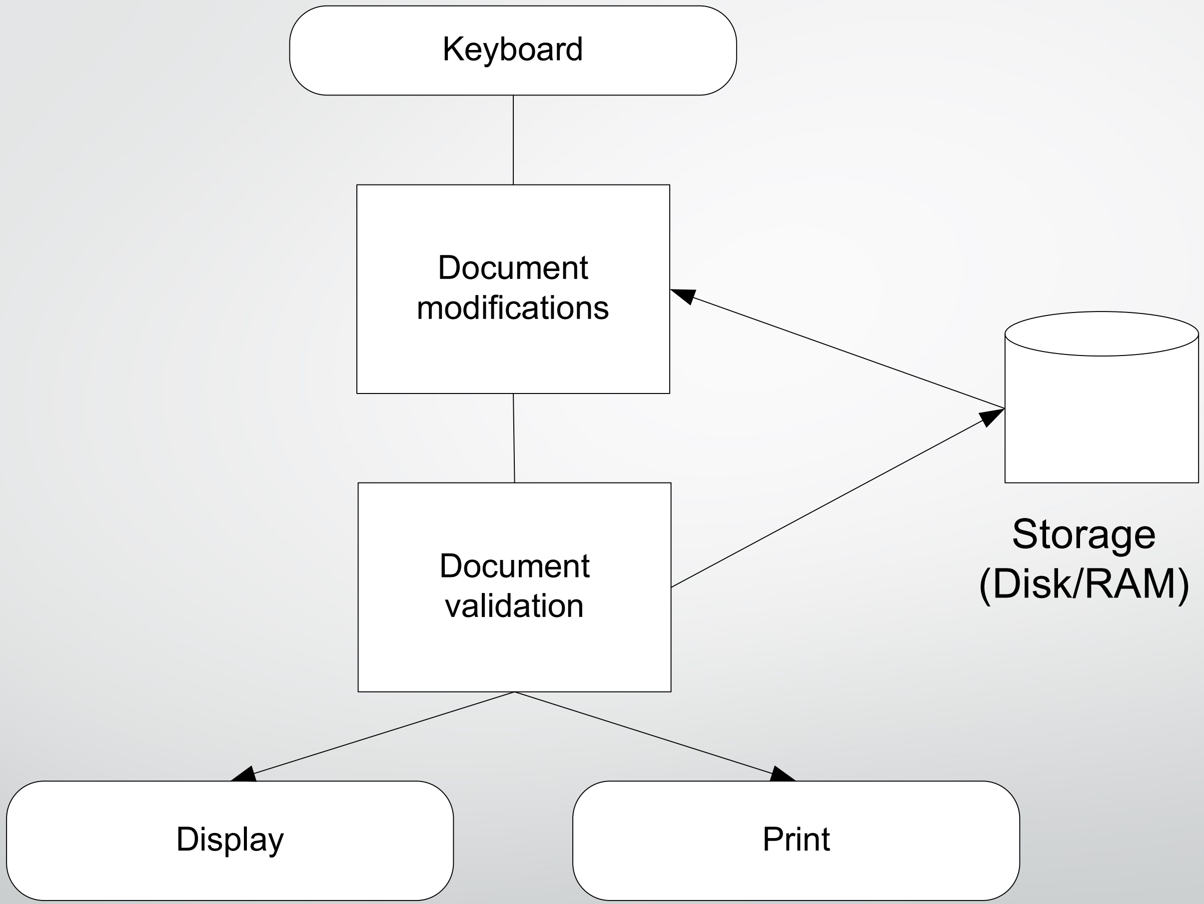
>"hello world!"	**Application Software**
	Operating Systems
	CPU Architecture & Microarchitecture
	Sequential & Combinatorial Digital Logic
	Physics & Devices

Input -> Process -> Output model

OÉ Gaillimh
NUI Galway

- Computer system is supposed to perform a useful operation, such as word processing, retrieval and manipulation of data, bookkeeping, etc.
  - i.e. a credit card transaction operation
- Regardless of the type of operation to be performed, the work of a computer can be characterized as an

  input->process->output model:
  - The program retrieves ***input*** from a disk file, mouse, keyboard or other type of input,
  - ***Processes*** the input
  - Produces the ***output*** to a disk, terminal, printer or some other type of output device
- All of the above operations are repetitive in nature



File Edit Workflow

Computing System Components

OÉ Gaillimh
NUI Galway

- ***Hardware*** – provides the physical mechanisms to input and output data, manipulating data and controlling the various input, output, storage and communication components

- ***Software*** – both application and system, which provides instructions that tell the hardware exactly what tasks are to be performed and in what order

- ***Data being manipulated*** – can be alphanumeric, graphic or any other form. In all cases it is represented in a form that the computer will understand and manipulate

Architecture versus Organization

OÉ Gaillimh
NUI Galway

- **Architecture**
  - Refers to those attributes of a system visible to a programmer
  - The architecture of a CPU is actually its instruction set, number of bits used for data representation, addressing techniques, etc…
- **Organization**
  - Refers to the operational units and their interconnections that realize the architectural specifications
  - Hardware details transparent to the programmer, such as control signals between different functional units, memory type (i.e. dynamic RAM or static RAM, etc…), registers type (static or dynamic), etc..
- It is an architectural issue whether a computer will or will not have a specific instruction (i.e. multiply), but it is an organization issue whether that instruction will be implemented by a special arithmetic unit or it will be implemented using the adder of the system by repetitive add operations

Computing Systems Description

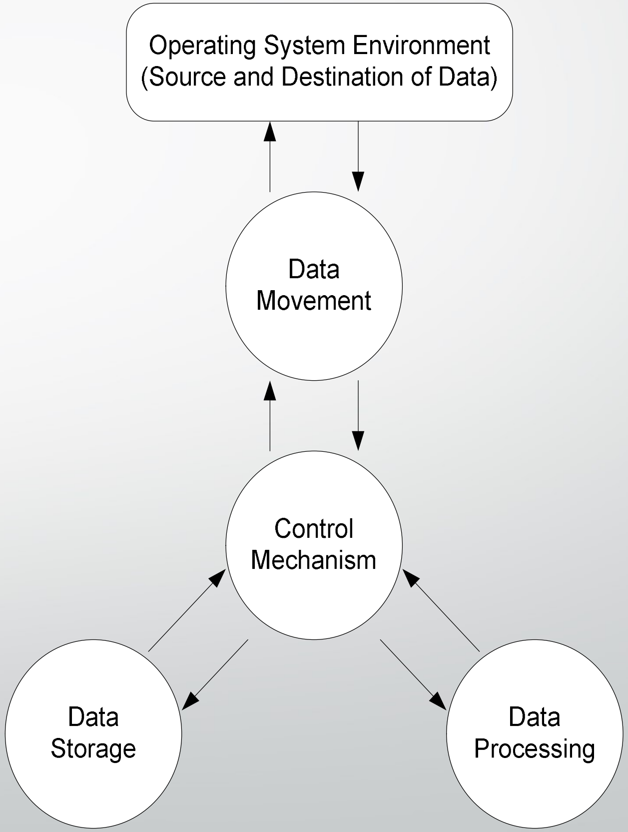
- Top down approach
  - Starting from a top view and decomposing the system into its subparts
- Bottom up approach
  - Starting from the bottom and building up a complete description
- Top-down approach seems to be the clearest and most effective.
  - However we will use both approaches trying to apply the best approach to a specific area

Structure versus Function

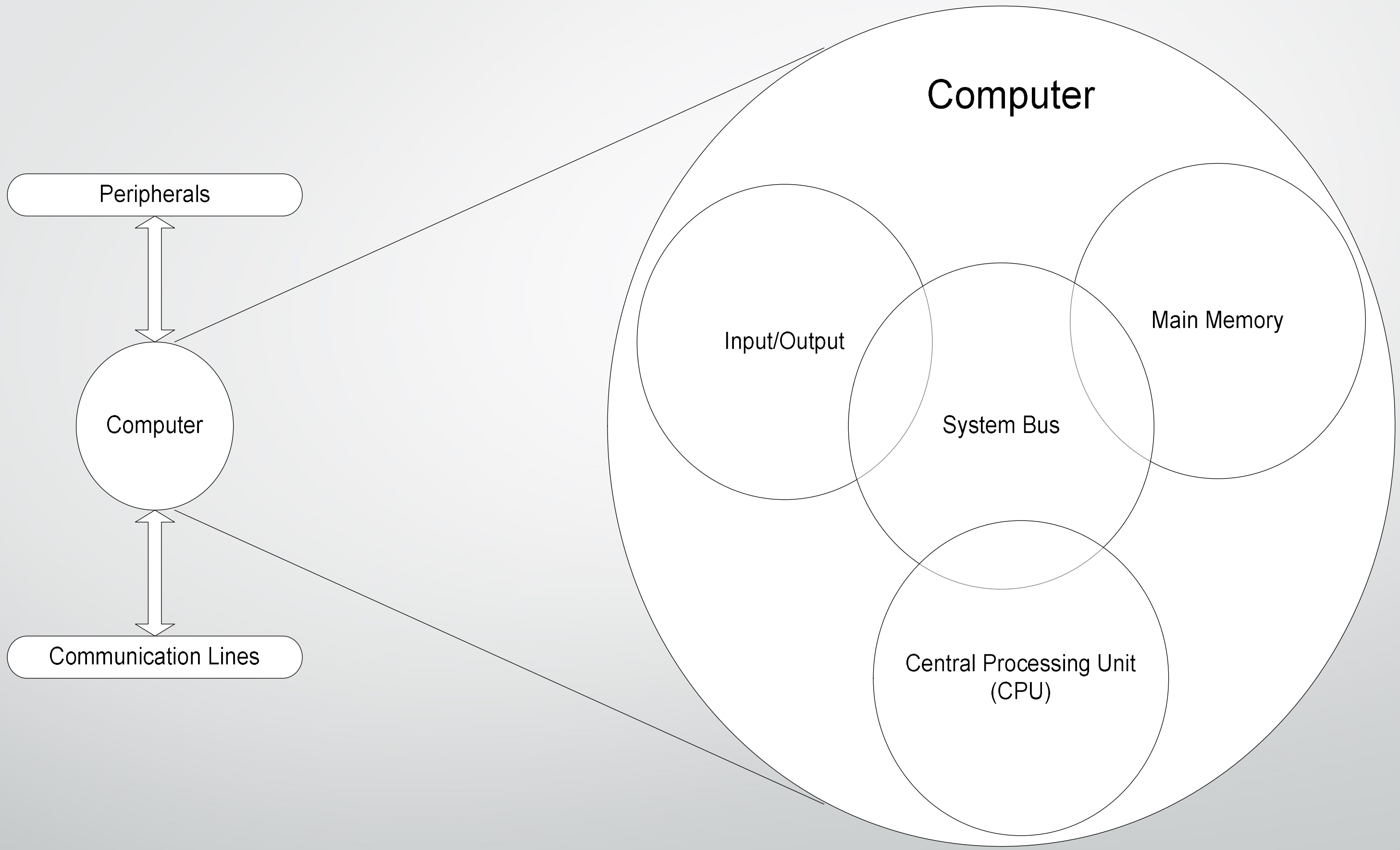
- Computing systems are complex machines made out of millions and millions of different components.
  - How can one clearly describe them??
  - The key is to recognize the hierarchical nature of most complex systems, including the computer.
  - Hierarchical system organized in a number of levels. Each level is characterized by structure and function:
- **Structure**: the way the components are interconnected
- **Function**: the operation of each individual component as part of the structure

Computing Systems Function

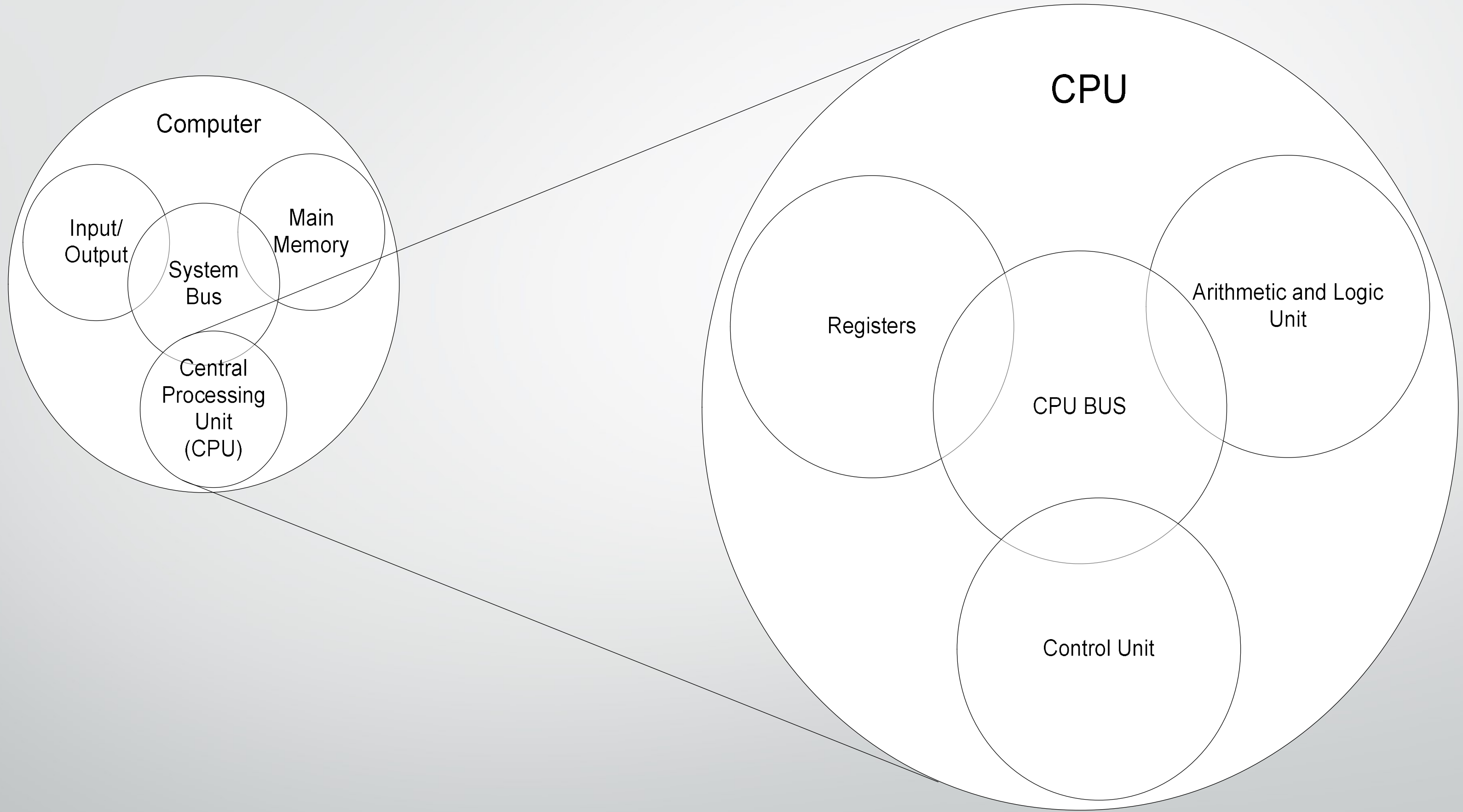
- Data Processing
  - Fundamental types of data
  - Fundamental types of processing
- Data Storage
  - Short term storage
  - Long term storage
- Data Movement
  - Input/Output for devices directly connected (peripherals)
  - Data communication for moving data over long distances
- Control
  - External (users)
  - Internal (manage resources)



Computing System Structure



CPU Structure

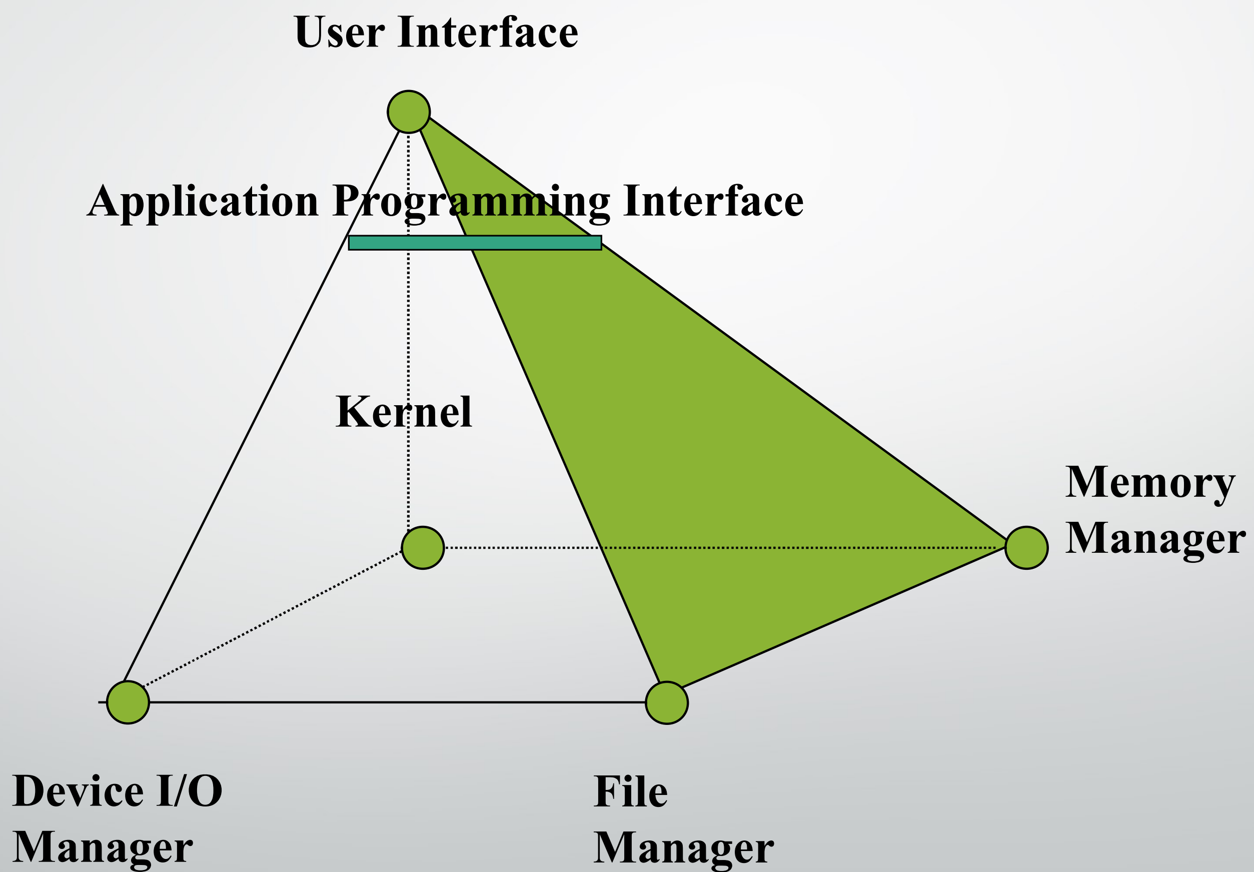


Computing Systems Software

OÉ Gaillimh
NUI Galway

- Application software
  - Performs specific tasks for users: spreadsheets, database systems, desktop publishing, program development, games, etc…
- System software
  - Provides infrastructure for application software
  - Consists of **operating system** and **utility software**

Operating System Components



Review Question 1

OÉ Gaillimh
NUI Galway

Architecture of a CPU refers to:

A. Its instruction set, number of bits used for data representation, addressing techniques, etc
B. Details on how the instructions are implemented
C. Details on how various subsystems (Arithmetic and Logic Unit, Registers and Control Unit) are interconnected
D. The operations of the control unit

Review Question 2

Out of the options below, identify one that is NOT a function of a computing system

A. Data storage
B. Power consumption
C. Data processing
D. Data movement

Review Question 3

OÉ Gaillimh
NUI Galway

Out of the options below, identify the one that is NOT part of a computing system structure

A. CPU
B. Memory
C. Buses
D. Data

References

- "The Architecture of Computer Hardware and Systems Software", Irv Englander, ISBN: 0-471-36209-3
- "Computer Systems", J Stanley Warford, ISBN: 0-7637-16633-2

- CT101 -
Computing Systems

Dr. Fatemeh Ahmadi Zeleti

Fatemeh.ahmadizeleti@NUIGalway.ie

School of Computer Science,

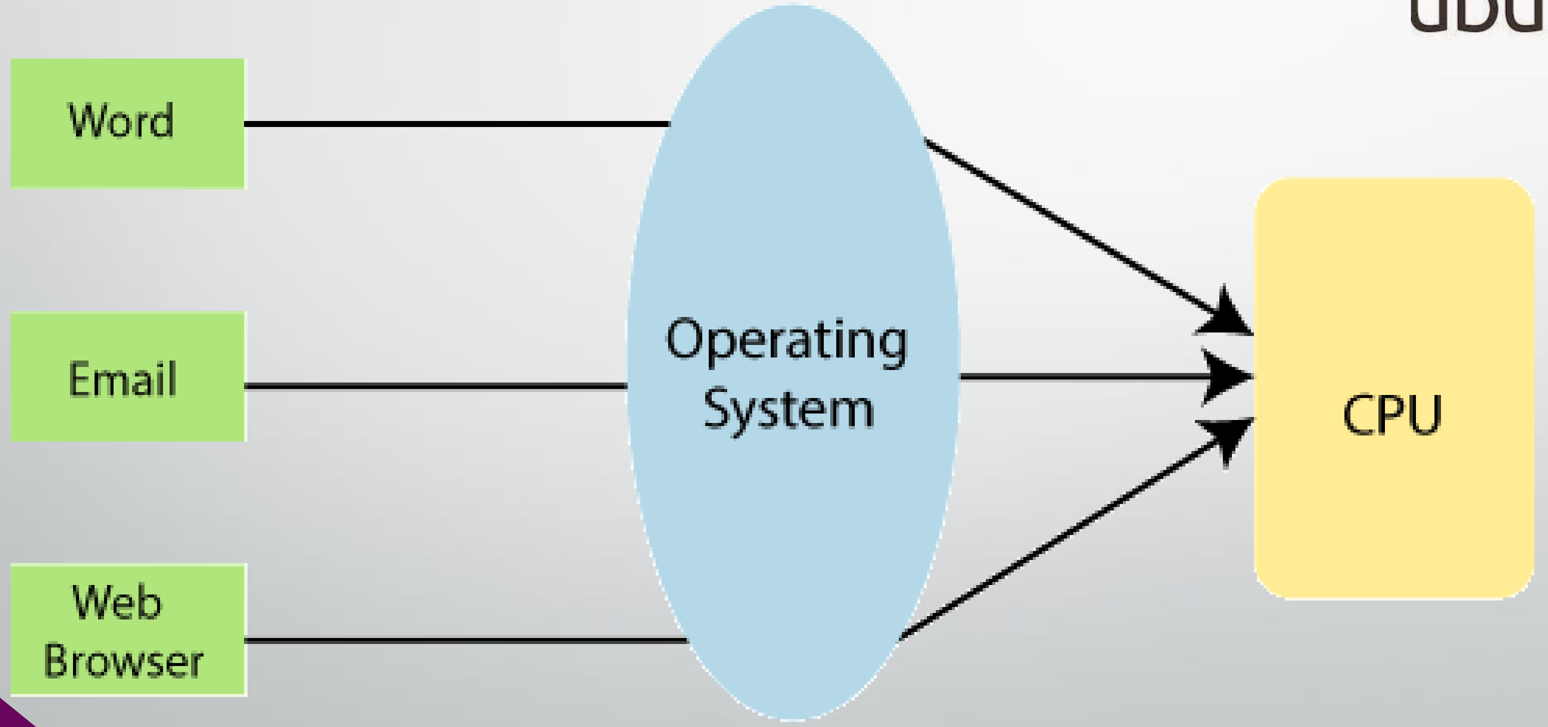
National University of Ireland, Galway

Introduction to Operating Systems

Contents

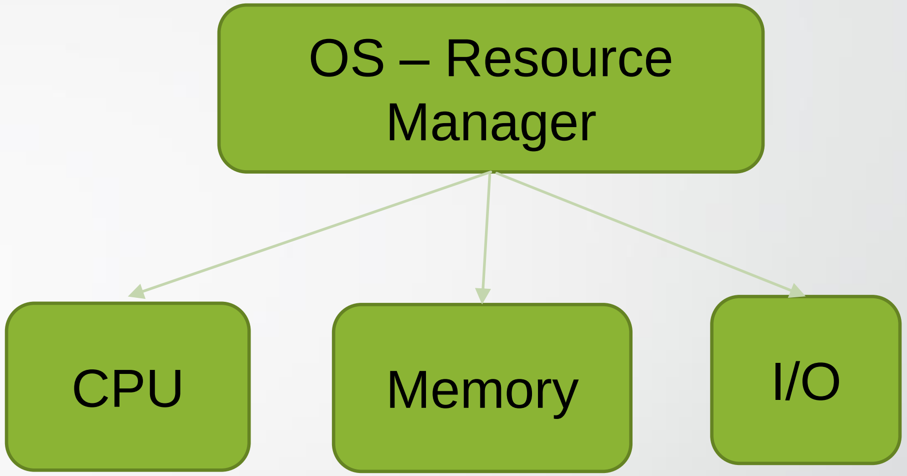
Operating System

- Interface between the user and the hardware



Functions of Operating Systems

- Oversee operation of computer
- Store and retrieve files
- Schedule programs for execution
- Coordinate the execution of programs



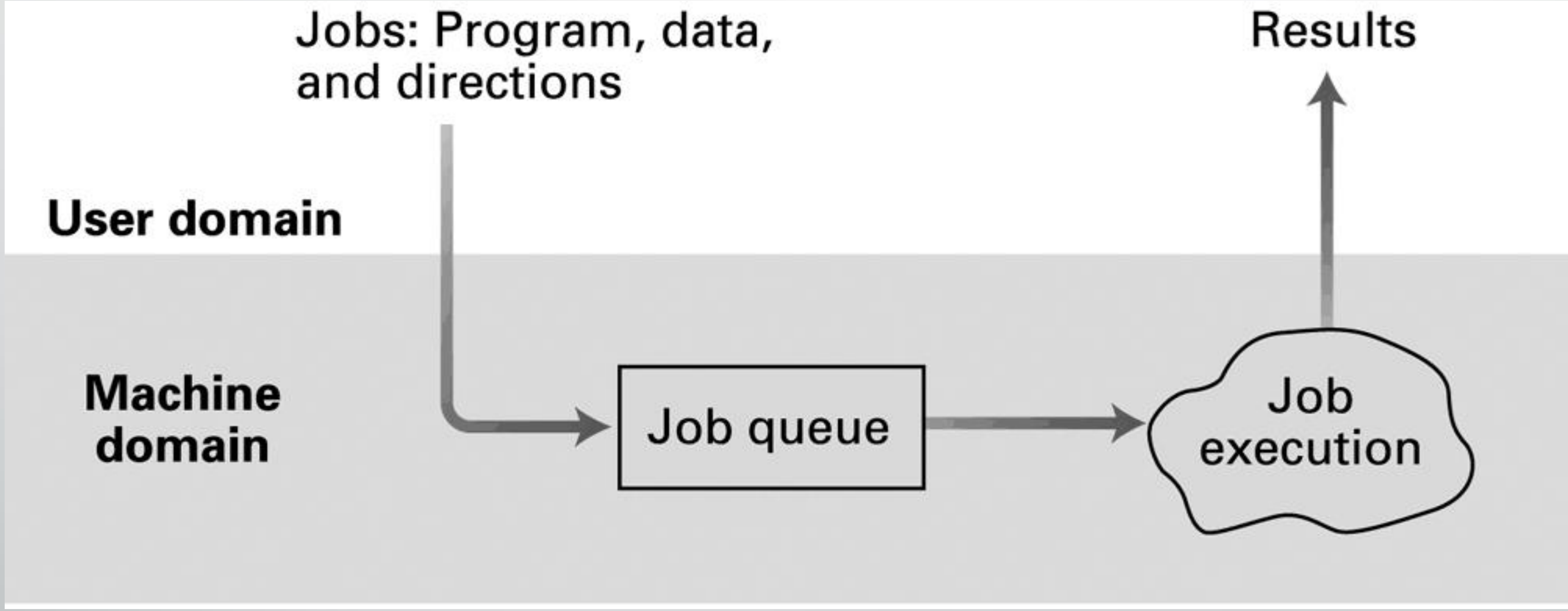
Other important yet hidden functions of an OS

- manage the computer's resources, such as the central processing unit, memory, disk drives, and printers,
- establish a user interface, and
- execute and provide services for applications software

Evolution of Shared Computing

- Batch processing OS
  - Requires batches of jobs of the same type
- Multiprogramming OS
  - Implemented by Multiprogramming
- Multitasking OS / Time-sharing / Fair share
  - An extension of multiprogramming
- Multiprocessing OS
  - Require multiprocessor machines
- Interactive processing OS/ Real Time OS
  - Requires real-time processing
- Embedded OS
- FYI other OSs: Distributes OS, Multiuser OS…

Batch Processing



Multiprogramming OS and Time Sharing / Multitasking OS

- Users seeking services from same machine at the same time – time sharing
  - Implemented using a technique called multiprogramming (time is divided into multiple intervals, execution of one job is limited to a single time interval)
- Multiple terminals connected to same machine
  - Driven by the fact that in the past computers were very expensive
- When multiprogramming is applied to single-user environments is usually called multitasking

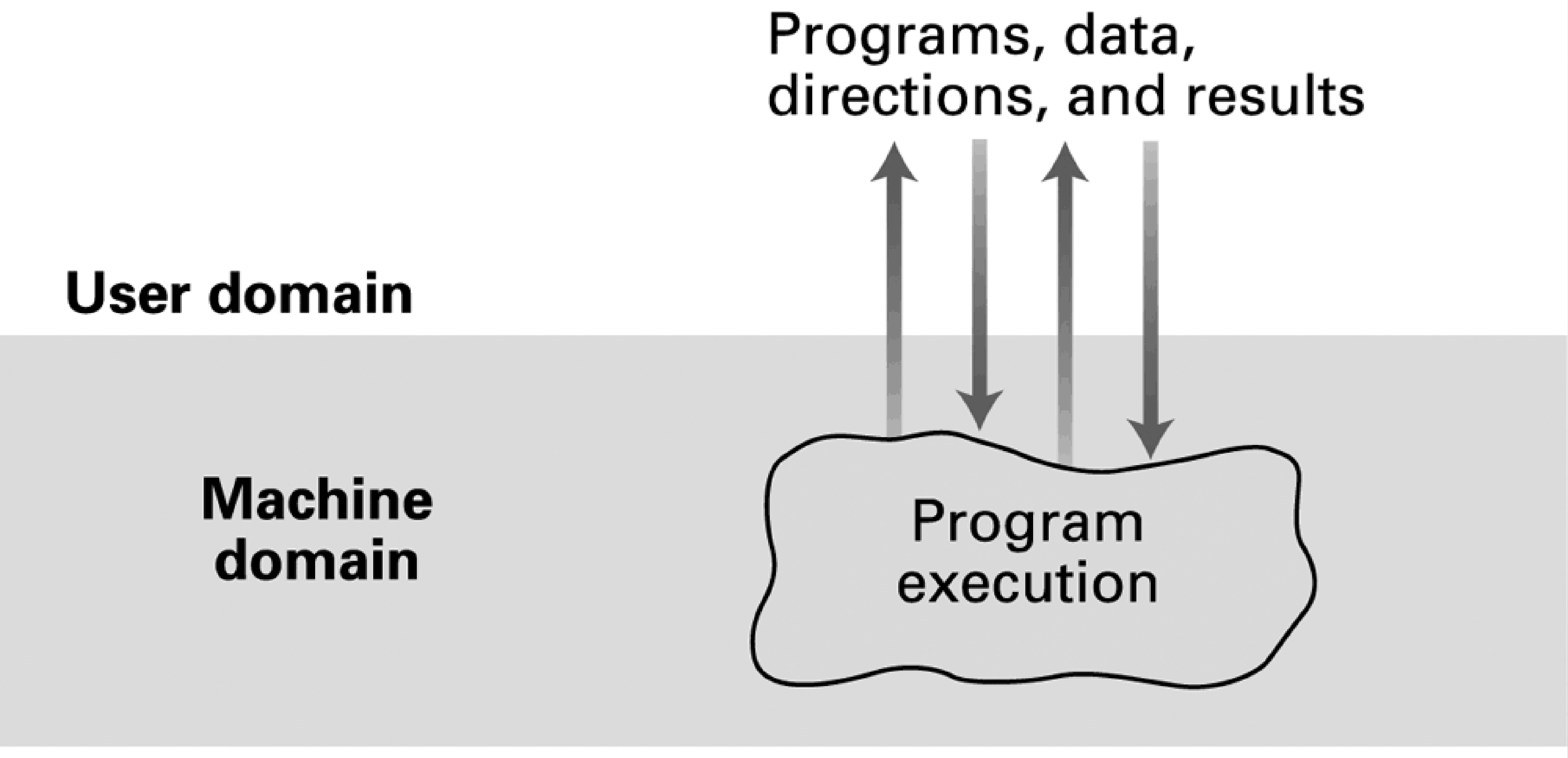
Multiprocessing OS

- Provide time sharing/multi-tasking capabilities by assigning different tasks to different processors as well as sharing the time of one single processor

- Problems to solve:
  - Load balancing – dynamically allocating tasks to the various processors so that all of them are used efficiently
  - Scaling – breaking tasks into sub-tasks compatible with the number of processors available

- Trend to develop a network wide operating system rather than networks of individual operating systems

Interactive Processing/ Real-Time OS



Embedded OS

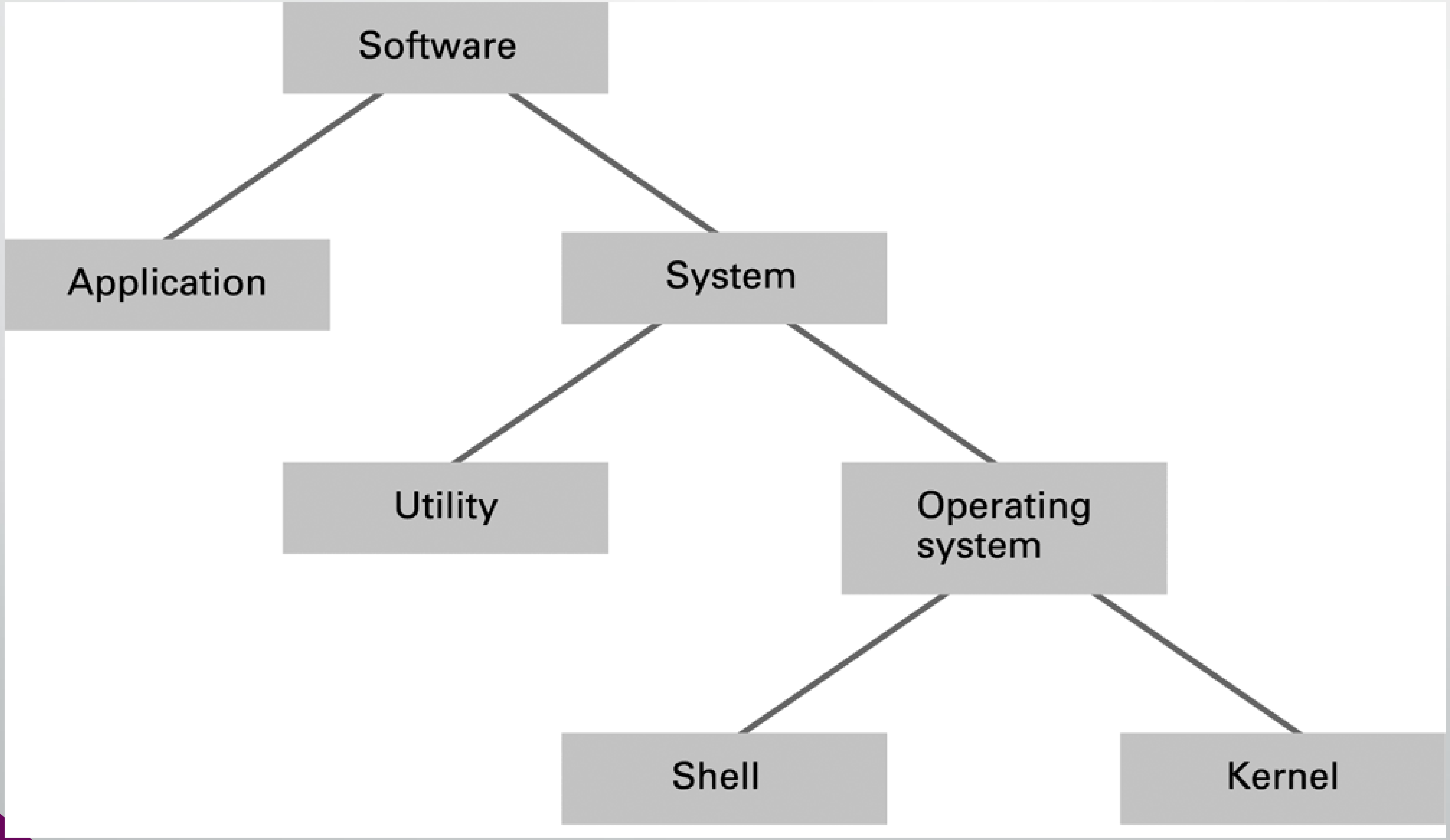
- Embedded OSs can be found in mobile phones, cars, large laser printers, some home appliances etc.
  - Other examples: Embedded Linux - Of which Android is a subset
- Limited data storage and power conservation are the big challenges
- Embedded operating system does not load and execute applications. Therefore, the system is only able to run a single application
- The applications are built into the OS or part of the OS, so they are loaded immediately when the OS starts

Types of Software

OÉ Gaillimh
NUI Galway

- Application software
  - Performs specific tasks for users: spreadsheets, database systems, desktop publishing, program development, games, etc…
- System software
  - Provides infrastructure/platform for application software to run
  - Consists of **operating system** and **utility software**

Software Classification

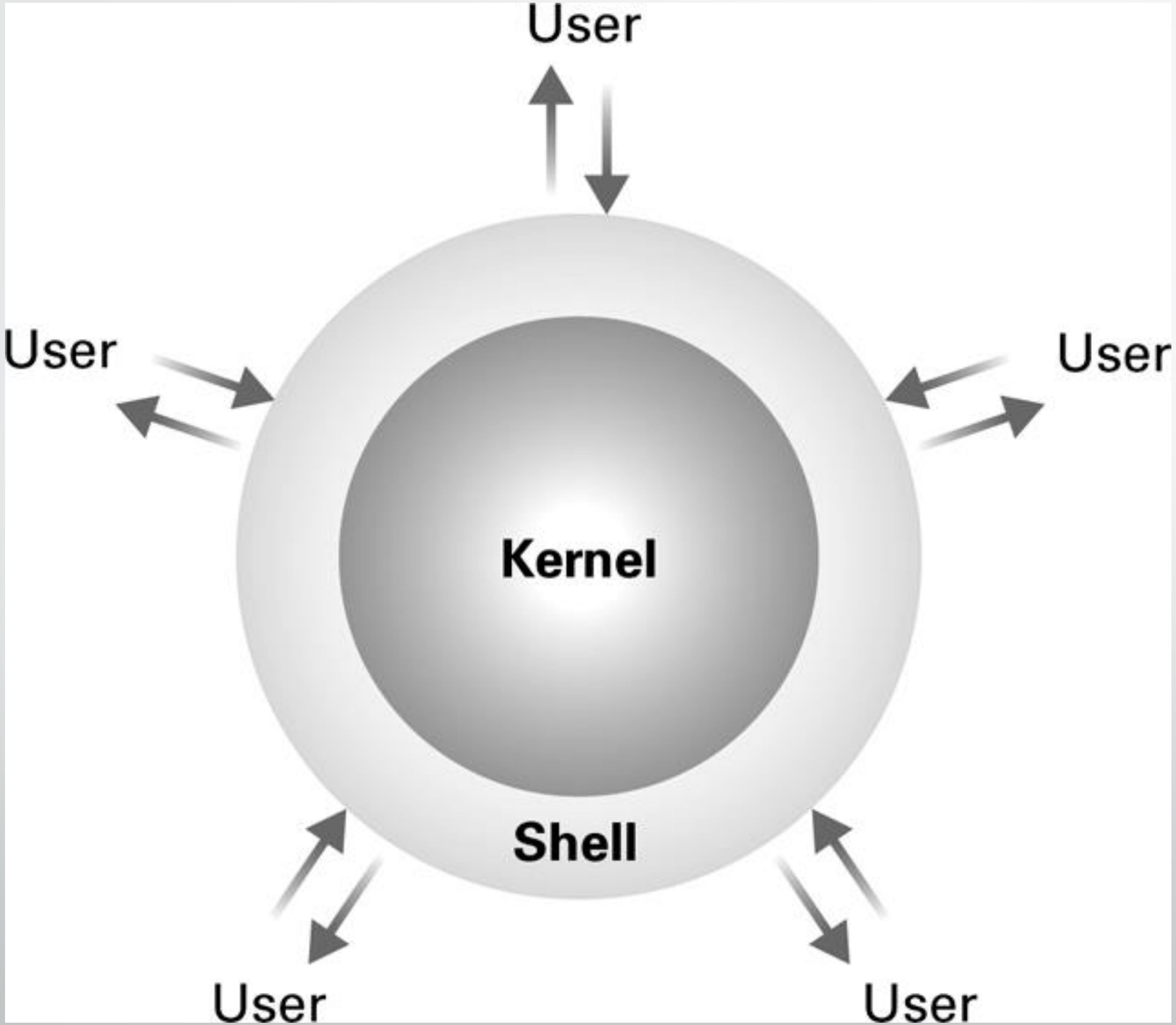


Operating System Components

- **Shell:** Communicates with users
  - Text based
  - Graphical user interface (GUI)

- **Kernel:** Performs basic required functions
  - Storage / File manager
  - Device drivers
  - Memory manager
  - Process manager (Scheduler, dispatcher, etc..)

The shell as an interface between users and the operating system



Storage/Hard Disk Management with the help of File Management

OÉ Gaillimh NUI Galway

- Role – coordinate the use of machine's mass storage facilities

- Hierarchical organization
  - **Directory** (or **Folder**): A user-created bundle of files and other directories (subdirectories)
  - **Directory Path:** A sequence of directories within directories

- Access/operations to files is provided by file manager via a **file descriptor**

I/O Device Management

OÉ Gaillimh
NUI Galway

- Part of OS presented as a collection of device drivers – specialized software that communicate with the controllers to carry out operations on peripheral devices connected to the computer

- Each driver is specifically designed for its type of device (e.g. printer, monitor, etc..) and translates generic requests into device specific sequence of operations

Memory Management /Main Memory/RAM

OÉ Gaillimh NUI Galway

- Has the task of coordinating the use of main memory – allocates/deallocates space in main memory

- When the total required memory space exceeds the physical available space.
  - May create the illusion that the machine has more memory than it actually does (**virtual memory**) by playing a “shell game” in which blocks of data (**pages**) are shifted back and forth between main memory and mass storage

Processes

- **Process:** The activity of executing a program
  - Program – static set of directions (instructions)
  - Process – dynamic entity whose properties change as time progresses. It is an instance in execution of a program.

- **Process State:** Current status of the activity
  - Program counter
  - General purpose registers
  - Related portion of main memory

Process Management

- **Scheduler** – the part of kernel in charge with the strategy for allocation/de-allocation of the CPU to each competing process
  - Maintains a record of all processes in the OS (via a **process table**), introduces new processes to this pool and removes the ones that completed

- **Dispatcher** is the component of the kernel that overseas the execution of the scheduled processes
  - Achieved by multiprogramming

Scheduler

- **Scheduler:** Adds new processes to the process table/memory and removes completed processes from the process table

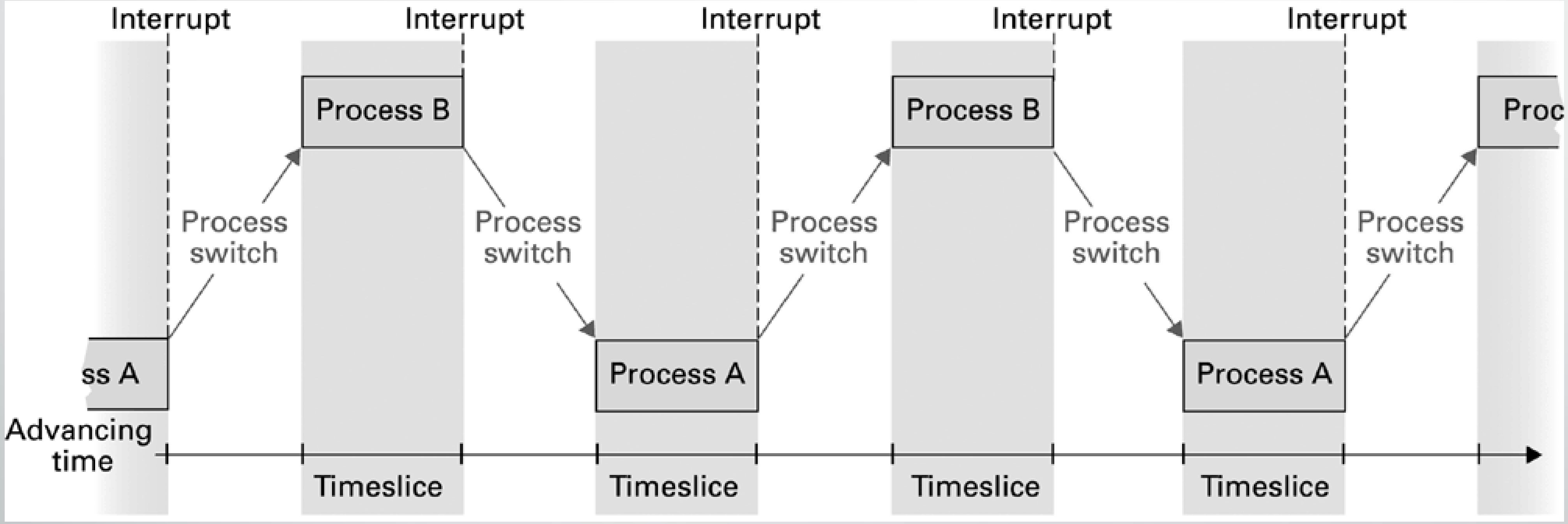
- Process table contains
  - Memory area assigned to the process
  - Priority of the process
  - State of the process (ready or waiting)

Dispatcher

- **Dispatcher:** Controls the allocation of CPU (of time slices) to the processes in the process table/memory
  - The end of a time slice is signaled by an interrupt.
  - Each process is allowed to execute for one time slice

- It performs “**process switch**” – procedure to change from one process to another
  - ProcessA→ Dispatcher→ ProcessB

Time-sharing between process A and process B



Security

- One of the role of OS is to provide security
- Attacks from outside
  - Problems
    - Insecure passwords
    - Sniffing software
  - Counter measures
    - Auditing software
  - Example:
    - SW that would impersonate the Operating System's user login screen
- Attacks from inside (Security at the process level: No process can interfere with the other one)
  - Securing the CPU to ensure that only one process can run at the same time
  - In case of Multiprocessing, securing all the processes in all the CPUs

Security (continued)

- Attacks from inside
  - Problem: Unruly processes
  - Counter measures: Control process activities via privileged modes and privileged instructions
  - Examples on attacker SW:
    - Alters the timer of OS – extend its own time slice and dominate the machine
    - Access to peripheral devices directly – access to files that otherwise access would have been denied
    - Access memory cells outside its allowed area, it can read and alter data from other processes

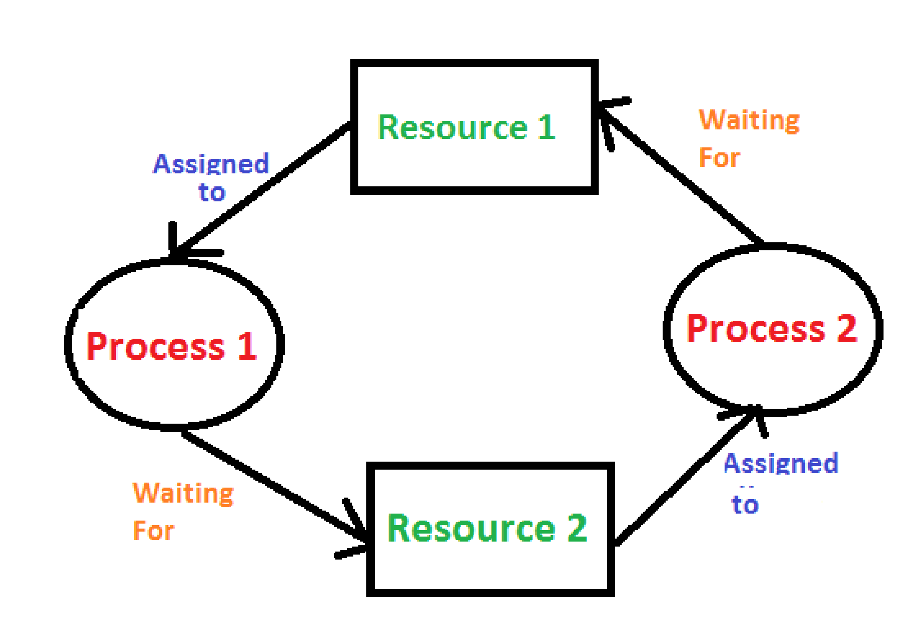
Handling Competition for Resources

- Important task of OS is to allocate *resources* to the processes
- **Semaphore:** A “control flag”
- **Critical Region:** A group of instructions that should be executed by only one process at a time
- **Mutual exclusion:** Requirement for proper implementation of a critical region so that only one process at a time will execute the sequence of instructions part of a critical region

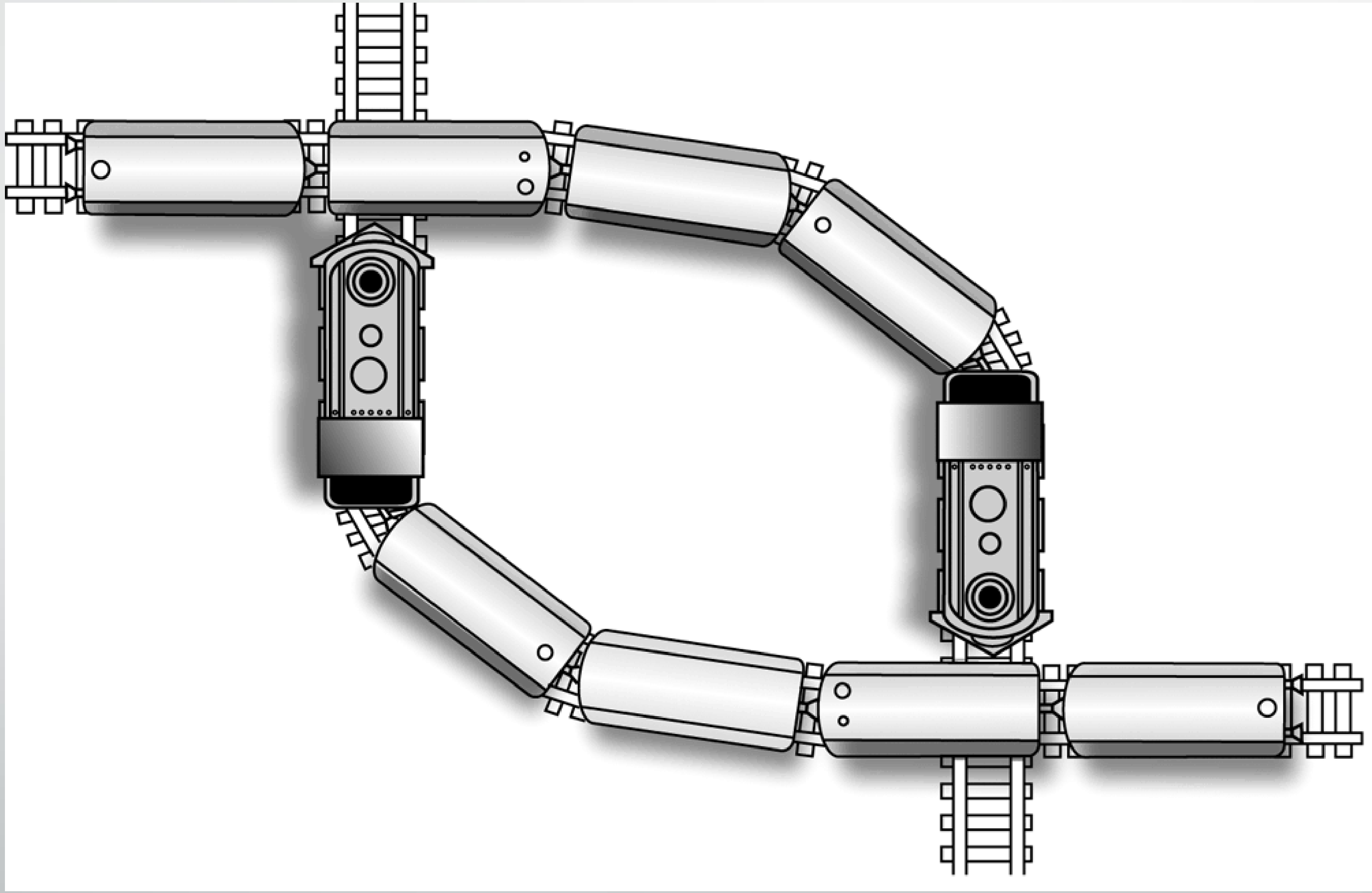
Deadlock

- Another problem of resource allocation - Processes block each other from continuing

- Conditions required for deadlock
  1. Competition for non-sharable resources
  2. Resources requested on a partial basis
  3. An allocated resource can not be forcibly retrieved



A deadlock resulting from competition for non-shareable railroad intersections

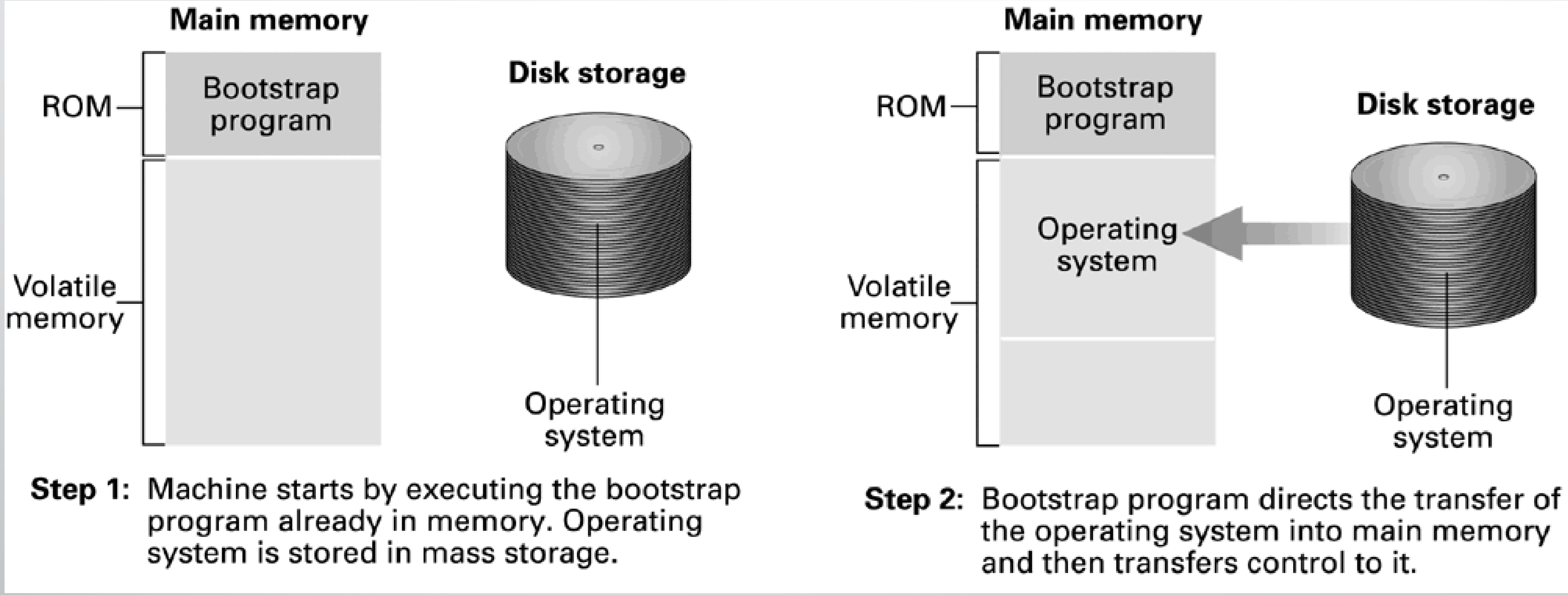


Getting OS Started (Bootstrapping)

- **Booting:** Procedure that transfers the OS from mass storage (permanent) into the main memory (volatile-thus empty when machine is turned on)

- **Bootstrap:** Program in ROM (example of firmware)
  - Run by the CPU when power is turned on (PC starts at pre-defined address when power is applied)
  - Transfers operating system from mass storage to main memory
  - Executes jump to operating system

The Booting Process



Reference

- J Glenn Brookshear “Computer Science – An Overview”, ISBN: 0-321-54428-5

OÉ Gaillimh
NUI Galway

- CT101 -
Computing Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Data Representation

Computing Systems Data

OÉ Gaillimh
NUI Galway

- Usually computing systems are complex devices, dealing with a vast array of information categories
- Computing systems store, present, and help us to modify:
  - Text
  - Audio
  - Images and graphics
  - Video

Digital vs. Analog (1)

- Computing systems are finite machines. They store a limited amount of information, even if the limit is very big.
  - The goal, is to represent enough of the world to satisfy our computational needs and our senses of sight and sound.

- The information can be represented in one or two ways:
  - **analog** or **digital**.

Digital vs Analog (2)

- **Analog data** is a continuous representation, analogous to the actual information it represents.
    - In example, a mercury thermometer is an analog device. The mercury rises in a continuous flow in the tube in direct proportion to the temperature.

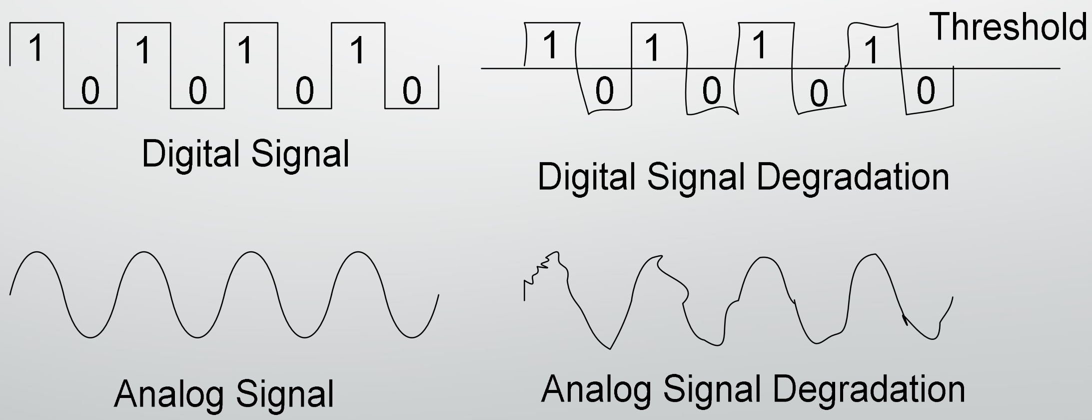
- **Digital data** is a discrete representation, breaking the information up into separate (discrete) elements.
    - Computers can't work with analog information, so they need to digitize the analog information.
    - This is done by breaking the analog information into pieces and representing those pieces using binary digits

Digital vs. Analog (3)

- Why digital signal?
  - Both electronic signals (analog and digital) degrade as they move down a line. The voltage of the signal fluctuates due to environmental effects.
  - As soon as an analog signal degrades, information is lost. Since any voltage level within the range is valid, it is impossible to know that the original signal was even changed
  - Digital signals jump sharply between two extremes (high and low state). A digital signal can degrade quite a bit until the information is lost, because any value over a certain threshold is considered high value and below the threshold is considered low value
- Answer: Signal Integrity can be maintained!

Digital vs. Analog (4)

- You can still retrieve the information from a reasonably degraded digital signal
- Periodically a digital signal is reclocked to regain its original shape. As long as it is reclocked before too much degradation, no information is lost.



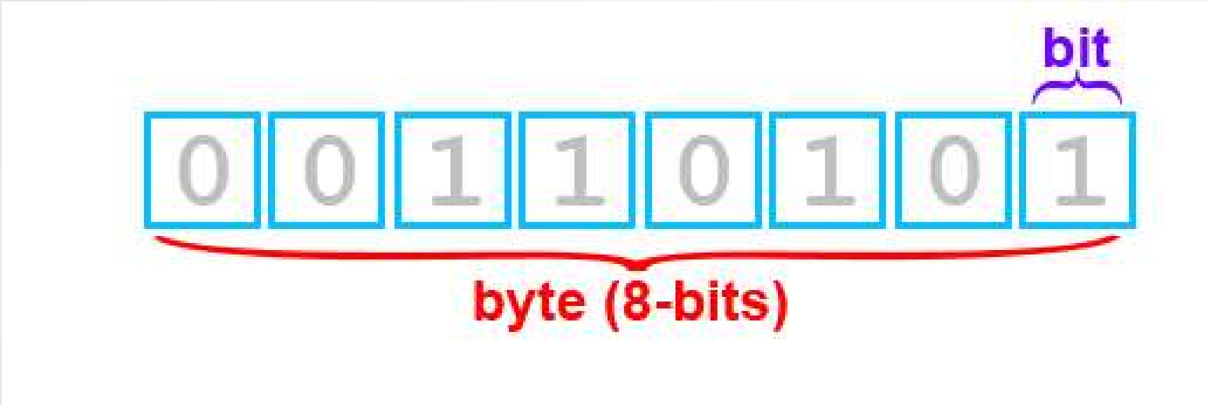
Binary Representation (1)

- One bit can be either 0 or 1. Therefore, one bit can represent only two things.
- To represent more than two things, we need multiple bits. Two bits can represent four things because there are four combinations of 0 and 1 that can be made from two bits: 00, 01, 10,11.
- In general, n bits can represent 2^n things because there are 2^n combinations of 0 and 1 that can be made from n bits. Note that every time we increase the number of bits by 1, we **double** the number of things we can represent.

Binary Representation (2)

- Why binary representation (as opposed to decimal or octal, etc..)?
- Because the devices that store and manage the digital data are far *less expensive and complex* for binary representation.
- They are also far *more reliable* when they have to represent one out of two possible values.
- Because the electronic signals are *easier to maintain* if they carry only binary data.

Binary Representation



- A **byte** is made up of 8 bits
- A byte can represent 256 different pieces of information
  - 2 to the power of 8

Review Question 1

- Why is a digital signal better than an analogue signal in computing systems
  A. Signal integrity can be maintained relatively easy
  B. Information is never lost
  C. Digital signal is more precise
  D. Digital signal can hold more information

Review Question 2

How many things can a bit represent ?

A. One
B. Two
C. Ten
D. Eight

Review Question 3

OÉ Gaillimh
NUI Galway

How many things can one byte represent ?

A. 2
B. 8
C. 256
D. 128

Data Formats - How to Interpret Data

- Meaning of internal representation must be appropriate for the type of processing to take place:
  - Images & sound: have to be digitized
    - Images – need detailed description of the data, how color is represented at each data point
    - Sound – need sampling rate
- Proprietary formats
  - Unique to a product or company
  - E.g., Microsoft *Word*, *Excel*, *PowerPoint*
- Standards
  - Evolve two ways:
    - Proprietary formats become standards (e.g., Adobe *PostScript*, Apple *Quick Time*)
    - Committee is struck to solve a problem (Motion Pictures Experts Group, *MPEG*)

Why Standards? (1)

- **Convenient** – sometimes the time to market is very important whenever trying to finish a product, therefore existing standards may be used to save time elaborating own protocols and interfaces
- **Efficient** – most of the standards are put together by committees with wide experience in the specific area
- **Flexible** – usually the standards allow for manufacturer or OEM specific extensions
- **Appropriate** – address a specific problem in a specific domain

Why Standards? (2)

- Allow communication and sharing of information
- Allow computing systems and software to interoperate (at both hardware and software levels)
- Sometimes standards are arbitrary and have some "*blast from the past*" (due to historical evolution)

Standards Organizations

- ISO – International Standards Organization
- IEEE – Institute for Electrical and Electronics Engineers
- CSA – Canadian Standards Association
- ANSI – American National Standards Institute
- NSAI – National Standards Authority of Ireland

Examples of Standards

Type of Data	Standards
Alphanumeric	ASCII, Unicode
Image	JPEG, GIF, PCX, TIFF, BMP, etc
Motion picture	MPEG-2, MPEG-4, etc
Sound	WAV, AU, MP3, etc..
Outline graphics/fonts	PostScript, TrueType, PDF

- CT101 -
Computing Systems

Dr. Fatemeh Ahmadi Zeleti

Fatemeh.ahmadizeleti@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Combinational Logic Design

Contents

- Overview
- Basic Gates and Boolean algebra
- Boolean functions manipulation and implementation
- Complex combinatorial circuit elements (multiplexers, decoders, encoders, comparators, adders)
- CLC Design & Implementation

Overview

- Gates, latches, memories and other logic components are used to design computer systems and their subsystems

- Two types of digital logic:
  - **Combinatorial Logic**: output is a function of inputs
  - **Sequential logic**: output is a complex function of current inputs, previous inputs or state and previous outputs

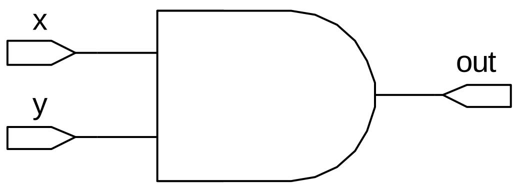
- Neither combinatorial logic nor sequential logic is better than the other. In practice, both are used as appropriate in circuit design.

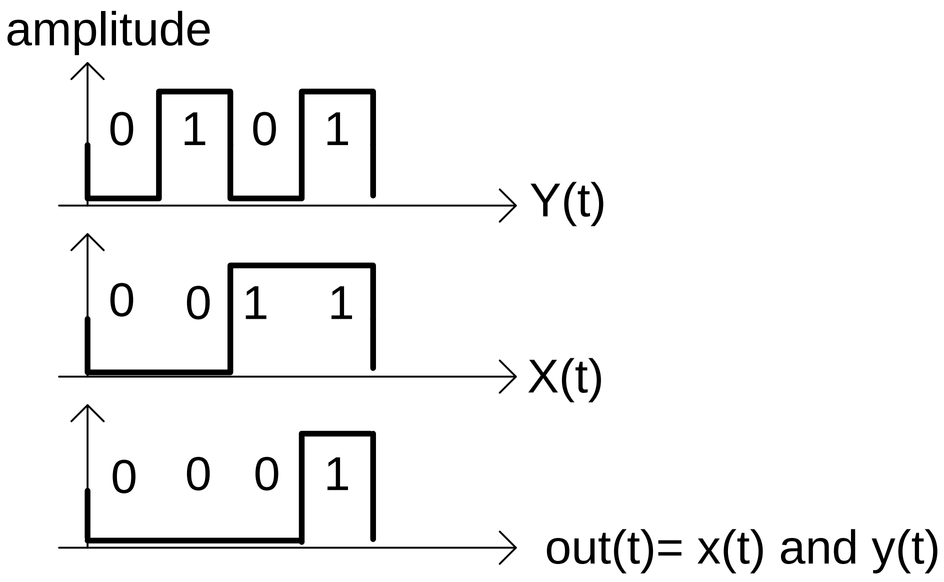
Boolean Algebra

- Review Boolean algebra, basic functions and methods used to **combine**, **manipulate** and **transform** Boolean functions & application to the implementation of combinatorial logic circuitry
- A *Boolean* algebra value can be either true or false.
- Digital logic uses 1 to represent true and 0 to represent false.
- Main operations of Boolean algebra are:
  - The conjunction **and** denoted as **∧** or **.** (multiplication)
  - the disjunction **or** denoted as **∨** or **+** (sum/addition) and
  - the negation **not** denoted as **¬**

AND (multiplication/dot notation)

x	y	out = x·y
0	0	0
0	1	0
1	0	0
1	1	1

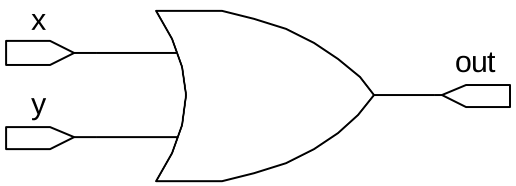


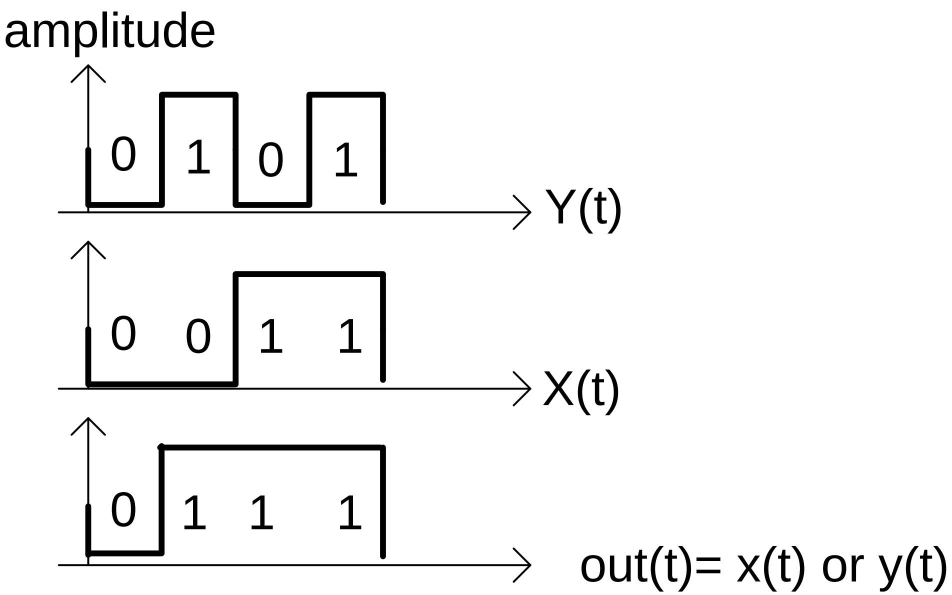


- Output is one if every input has value of 1
- More than two values can be "and-ed" together
- For example xyz = 1 only if x=1, y=1 and z=1

OR (addition/plus notation)

x	y	out = x+y
0	0	0
0	1	1
1	0	1
1	1	1

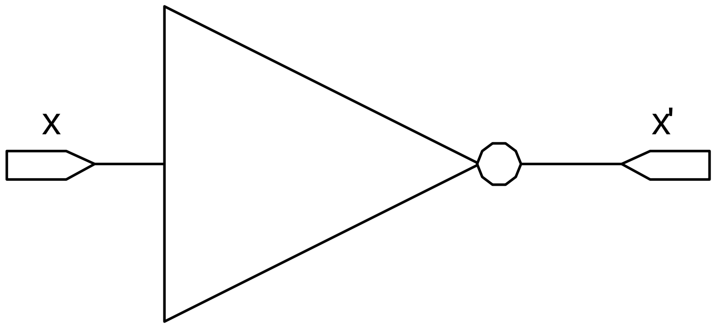


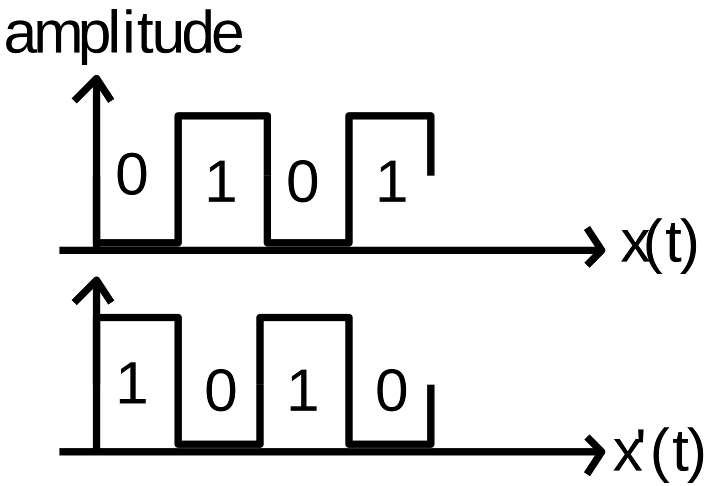


- Output is 1 if at least one input is 1.
- More than two values can be "or-ed" together.
- For example x+y+z = 1 if at least one of the three values is 1.

NOT (negation/logical complement)

x	x'
0	1
1	0



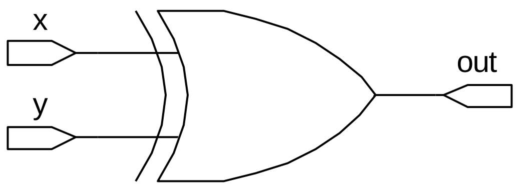


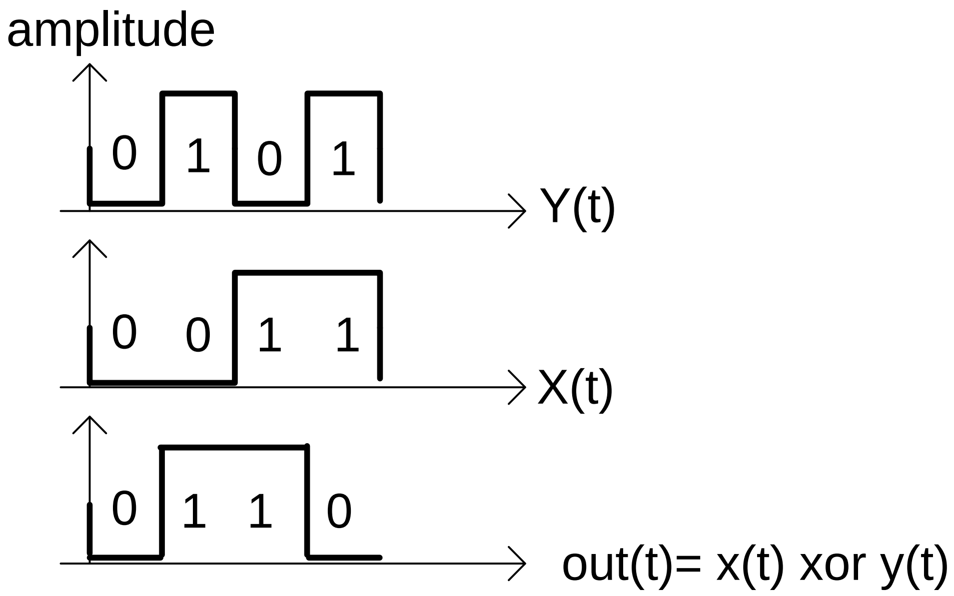
- This function operates on a single Boolean value.
- Its output is the complement of its input.
- An input of 1 produces an output of 0 and an input of 0 produces an output of 1

XOR (Exclusive OR)

XOR is a logical operation that outputs true or **1** only when inputs **differ**

x	y	out = $x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0



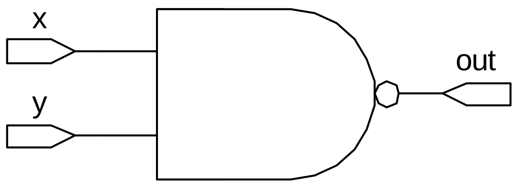


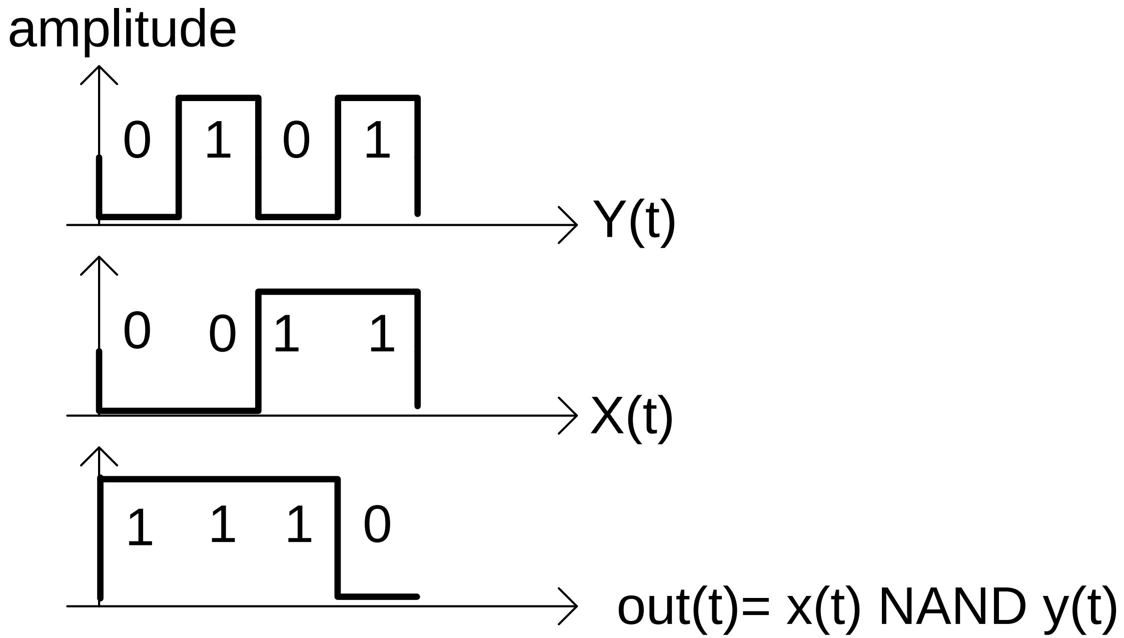
- The number of inputs that are 1 matter.
- More than two values can be “xor-ed” together.
- General rule: the output is equal to 1 if an odd number of input values are 1 and 0 otherwise

NAND (negative AND/multiplication)

NAND is a logic gate which produces false or **0** <u>only</u> if **all its inputs are true or 1**

x	y	out = x NAND y
0	0	1
0	1	1
1	0	1
1	1	0



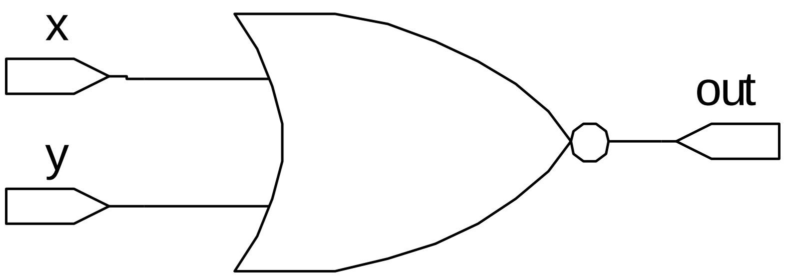


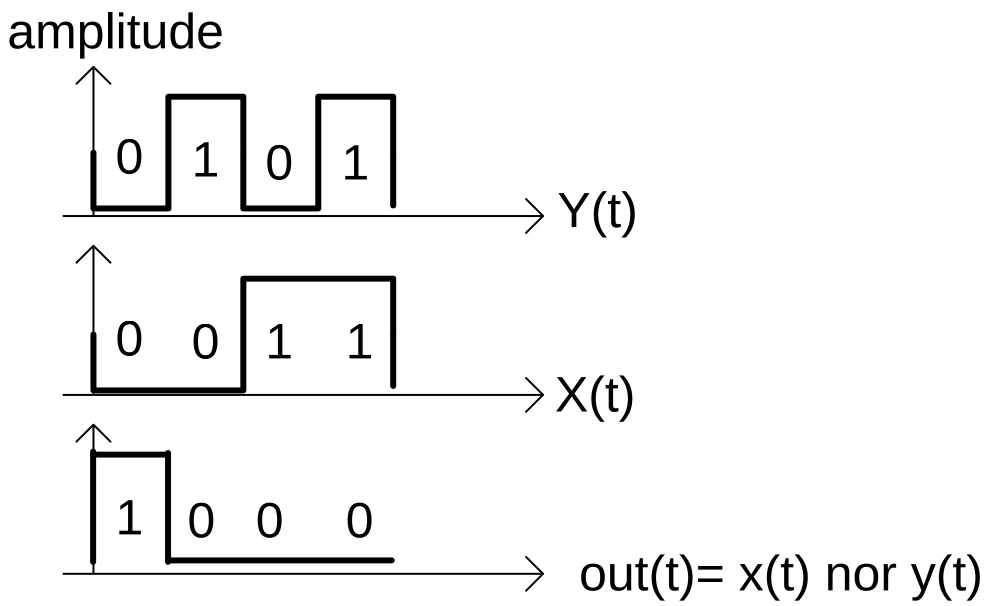
- Output value is the complemented output from an "AND" function.

NOR (negative OR/addition)

NOR is a logic gate which produces true or **1** if **both the inputs to the gate are 0**

x	y	out = x NOR y
0	0	1
0	1	0
1	0	0
1	1	0

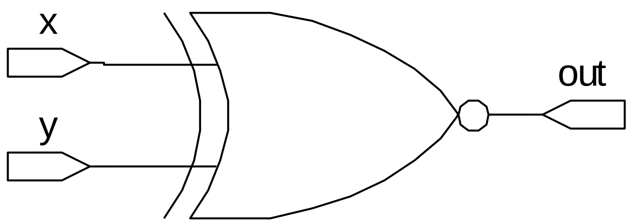


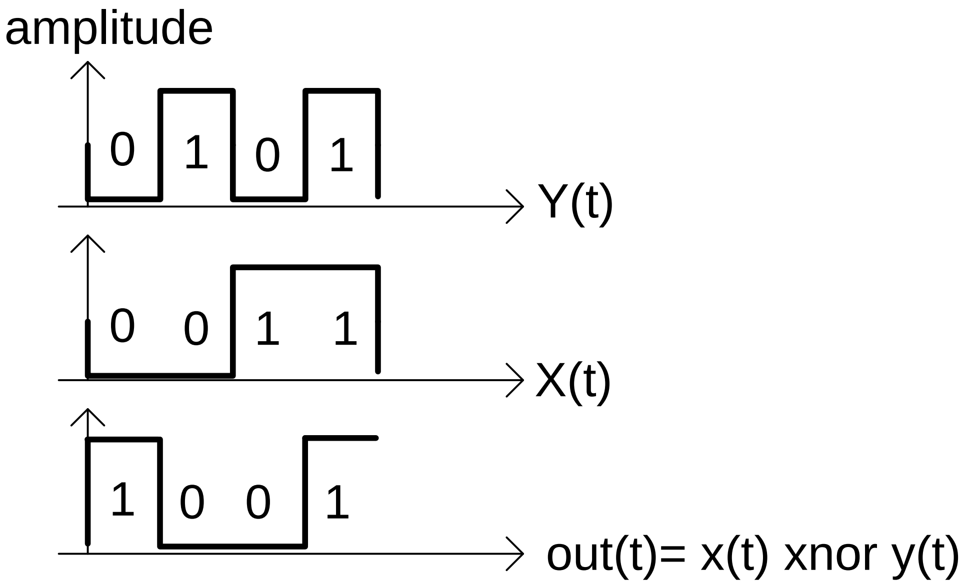


- Output value is the complemented output from an “OR” function.

XNOR (exclusive negative OR/addition)

x	y	out =x xnor y
0	0	1
0	1	0
1	0	0
1	1	1





- Output value is the complemented output from an “XOR” function.

Manipulating Boolean Functions

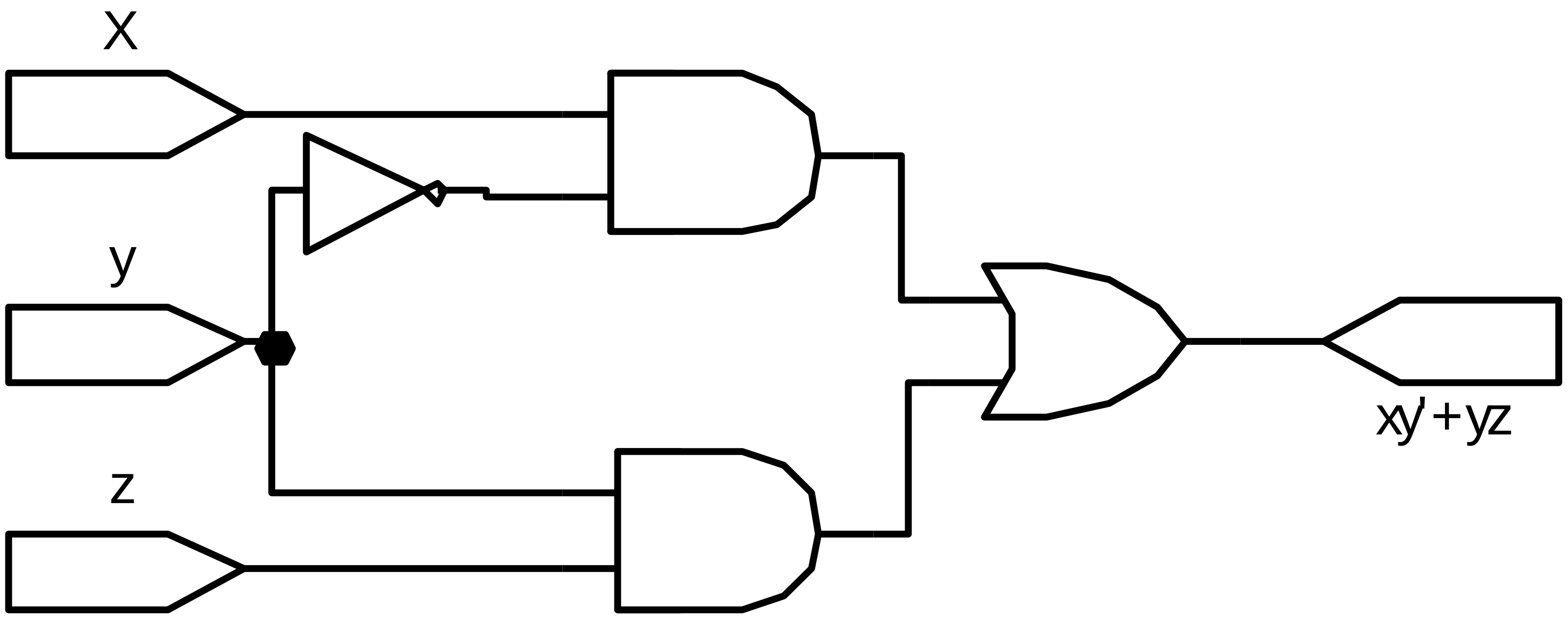
- Consider a function that must be 1 if either x = 1 and y = 0 or y = 1 and z = 1

- We express it as: **f(x,y,z) = xy'+ yz**

x	y	z	xy'	yz	xy'+yz
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	1	0	1
1	0	1	1	0	1
1	1	0	0	0	0
1	1	1	0	1	1

Combinatorial Logic Circuit

- Combinatorial Logic Circuit that implements the function **f(x,y,z)=xy’+yz**



DeMorgan's Laws

(ab)'=a'+b' => AND to OR

(a+b)'=a'b' => OR to AND

- Property for generating equivalent functions
  - Allows conversion of AND function to an equivalent OR function and vice-versa
- Could allow the simplification of complex functions, that will allow a simpler design
- It is useful in generating the complement of a function

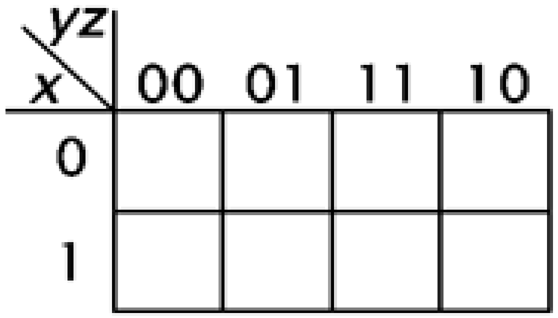
Using DeMorgan's law

- Generate complement of **f(x,y,z)=xy'+yz**
- (xy' + yz)' = (xy')'(yz)' = (x' + y)(y' + z') = x'y' + x'z' + yy' + yz' (because yy'=0) => **x'y' + x'z' + yz'**

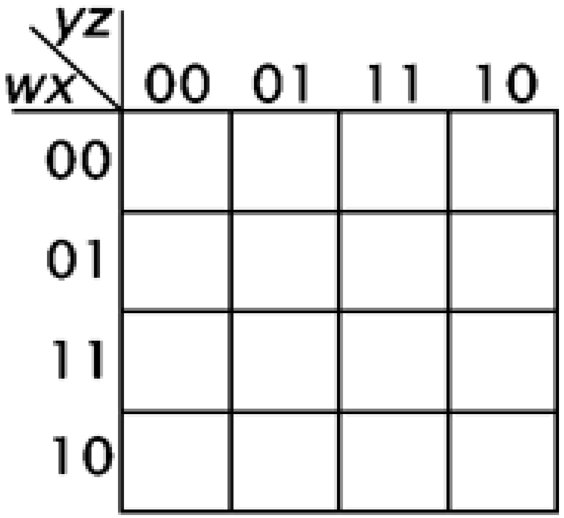
x	y	z	x'y'	x'z'	yz'	x'y' + y'z' + yz'
0	0	0	1	1	0	1
0	0	1	1	0	0	1
0	1	0	0	1	1	1
0	1	1	0	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	0	0	1	1
1	1	1	0	0	0	0

Karnaugh Map (K map)

- Pictorial Method for minimizing logic
- The rows and columns of the K-map correspond to the possible values of the function's input
- Each cell in the K-map represents a **minterm (**i.e. a three variables function has: x’y’z’, x’y’z, x’yz’, x’yz, xy’z’, xy’z, xyz’ and xyz)



(a)



(b)

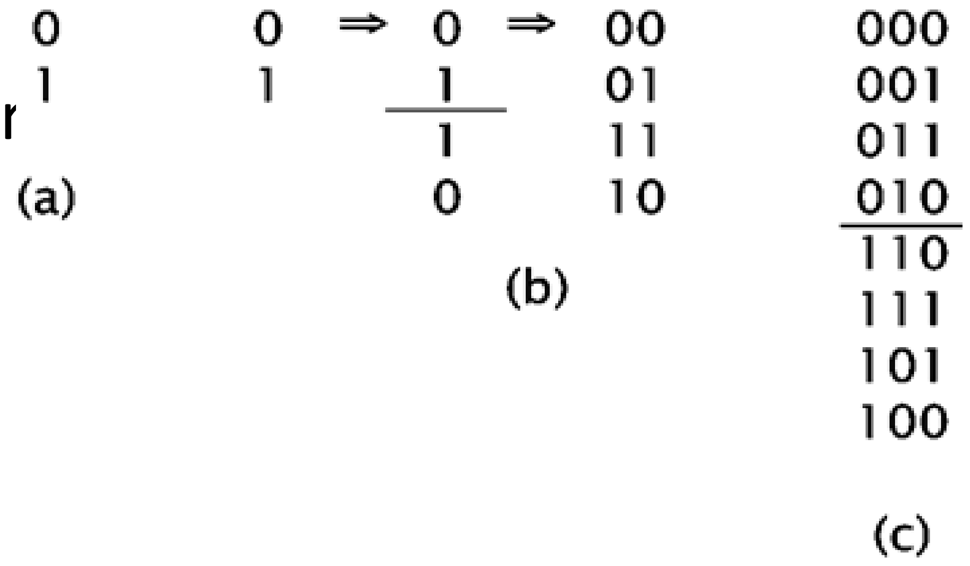
Gray Code order: input values do not follow the linear progression

Gray Code

- The 1-bit Gray code serves as basis for the 2-bit Gray code, the 2-bit Gray code is the basis for 3-bit Gray code, etc…

- Gray code sequences are cycles: 000 -> 001 -> 011 -> 010 -> 110 -> 111 -> 101 -> 100 -> 000 ….

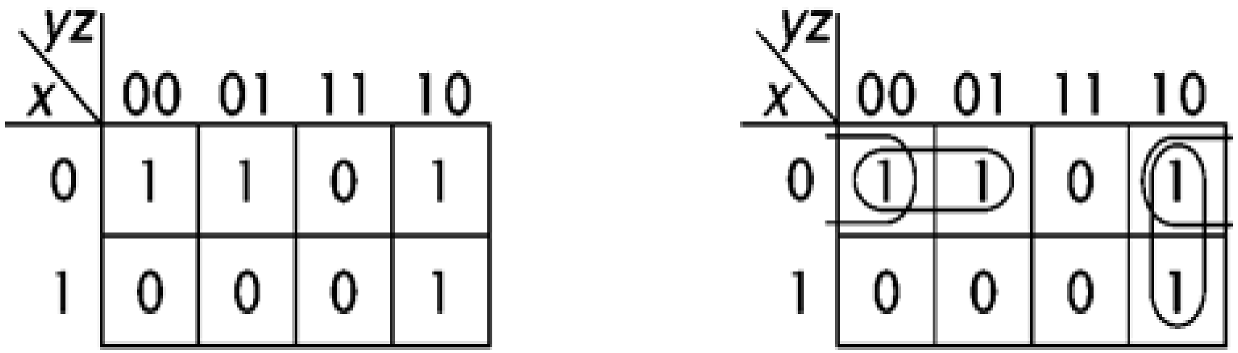
- Adjacent values differ



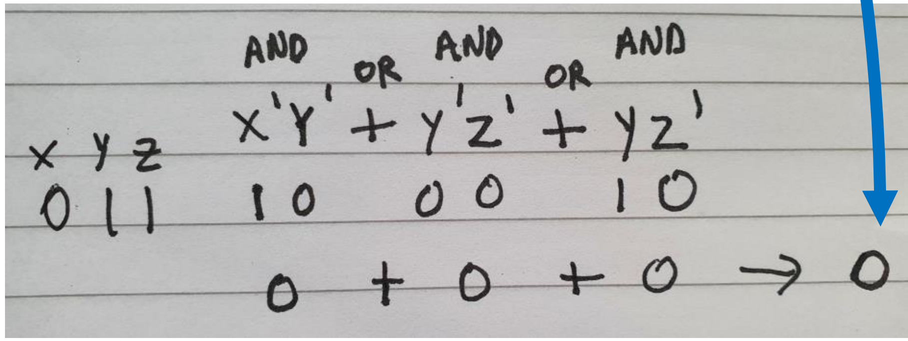
K-map Example

- Consider $(xy'+yz)' = x'y' + x'z' + yz'$
- Group together the 1s in the map:
  - g1: $x'y'z'+x'y'z=x'y'(z'+z)=x'y'$
  - g2: $x'yz'+xyz' = yz'(x'+x)=yz'$
  - g3: $x'yz'+x'y'z'=x'z'(y+y')=x'z'$
- Must select the fewest groups that cover **all** active minterms (1s): $(xy' + yz)'= x'y' + yz'$

x	y	z	x'y'+y'z'+yz'
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

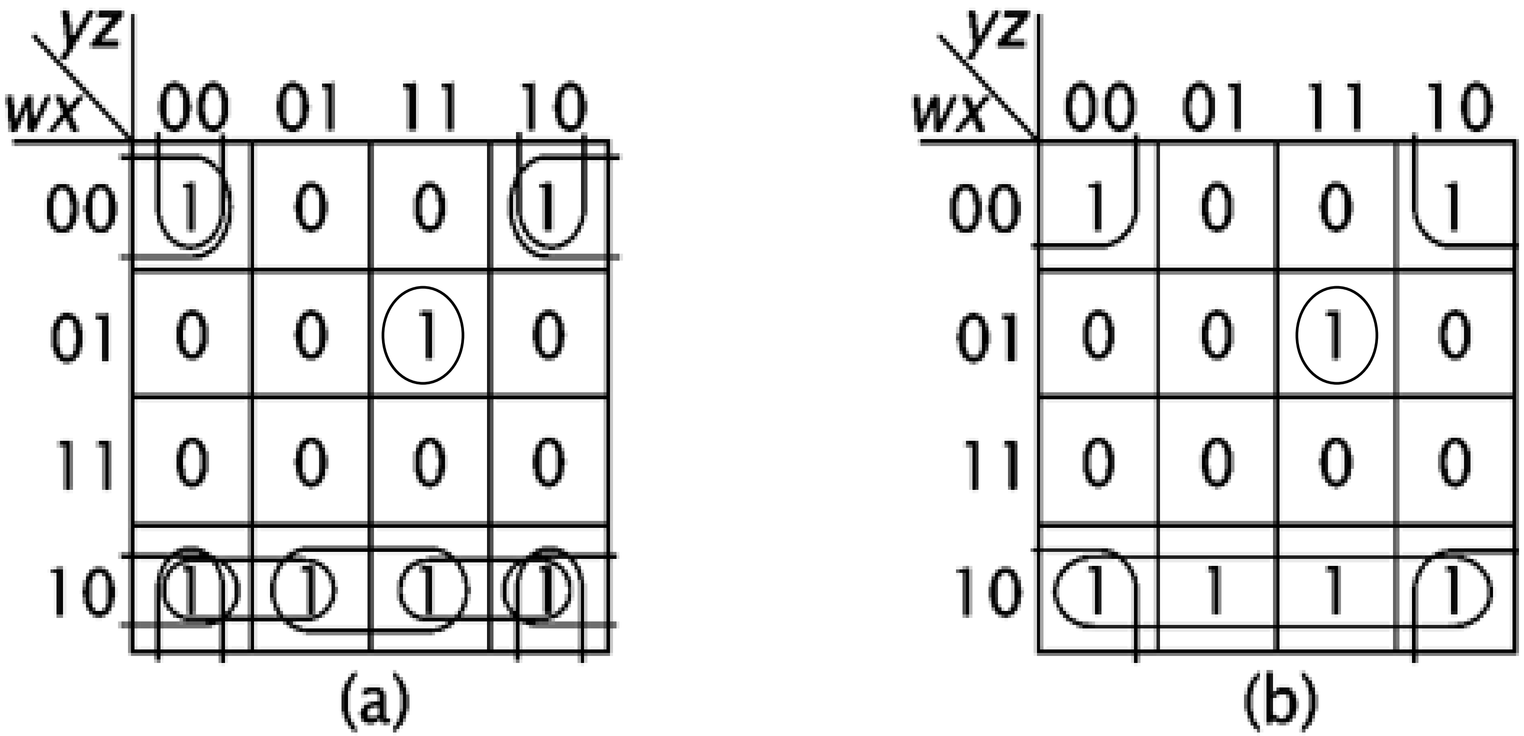


(a) (b)



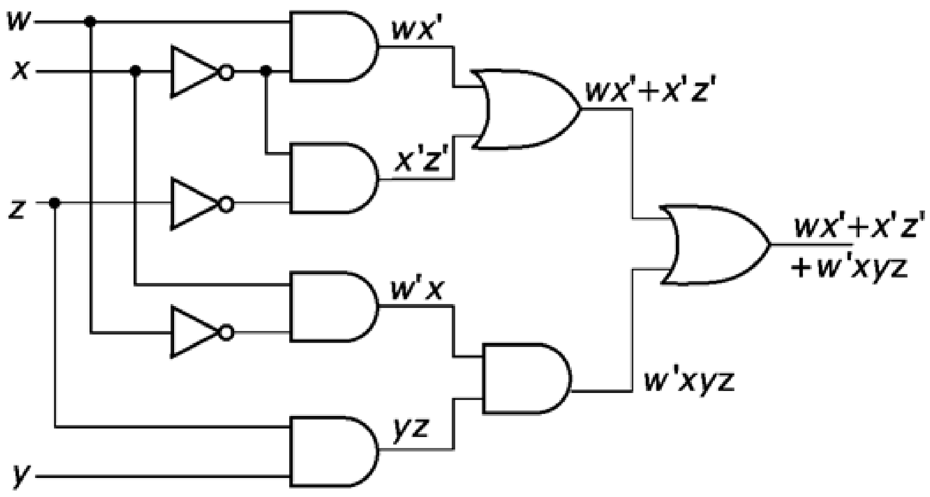
K-map for more complex function

w’x’y’z’ + w’x’yz’ + wx’y’z’ + wx’y’z + wx’yz + wx’yz’ + w’xyz

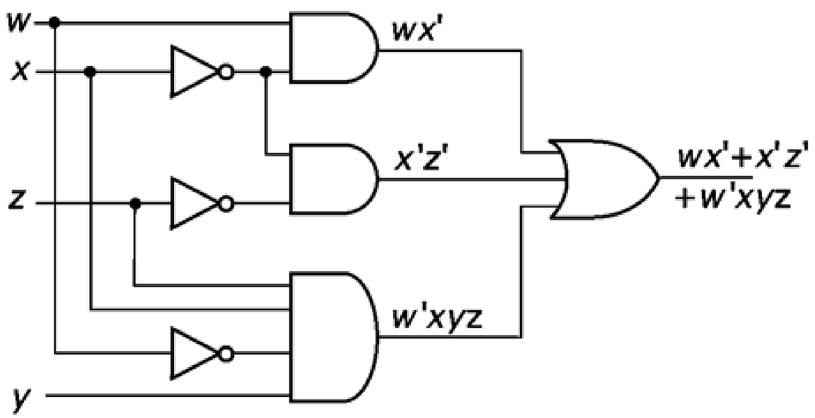


The final minimized function is: x’z’ + wx’ + w’xyz

Possible Implementations



(a)



(b)

Past Exam Question

- Consider the following four input variable function:
- **f(X3,X2,X1,X0) = X3'X2'X1'X0 + X3'X2'X1X0 + X3'X2X1'X0' + X3X2X1'X0' + X3X2'X1'X0 + X3X2'X1X0**
- Determine the minimum form of the function using a Karnaugh map.

- CT101 -
Computing Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Alphanumeric Data

- Standards for representing letters (alpha) and numbers
  - ASCII – ***A***merican ***S***tandard ***C***ode for ***I***nformation ***I***nterchange
  - Unicode

Codes and Characters

- The problem:
  - Representing text strings, such as "`Hello, world`", in a computer
- Each character is coded as a byte (= 8 bits)
- Most common coding system is ASCII
- ASCII = American National Standard Code for Information Interchange
- Defined in ANSI document X3.4-1977

ASCII Features

OÉ Gaillimh
NUI Galway

- 7-bit code
- 8^{th} bit is unused (or used for a parity bit)
- 2^7 = **128** codes
- Two general types of codes:
  - 95 are “Graphic” codes (displayable on a console)
  - 33 are “Control” codes (control features of the console or communications channel)

OÉ Gaillimh
NUI Galway

Most significant bit

	000	001	010	011	100	101	110	111
0000	NULL	DLE		0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EDT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Least significant bit

	000	001	010	011	100	101	110	111
0000	NULL	DLE		0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EDT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

i.e. ‘a’ $= 1100001_2 = 97_{10} = 61_{16}$

	000	001	010	011	100	101	110	111
0000	NULL	DLE		0	@	P	`` ` ``	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EDT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

95 Graphic codes

OÉ Gaillimh
NUI Galway

	000	001	010	011	100	101	110	111
0000	NULL	DLE		0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EDT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

33 Control codes

	000	001	010	011	100	101	110	111
0000	NULL	DLE		0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EDT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Alphabetic codes

“Hello, world” Example

		Binary		Hexadecimal		Decimal
H	=	01001000	=	48	=	72
e	=	01100101	=	65	=	101
l	=	01101100	=	6C	=	108
l	=	01101100	=	6C	=	108
o	=	01101111	=	6F	=	111
,	=	00101100	=	2C	=	44
	=	00100000	=	20	=	32
w	=	01110111	=	77	=	119
o	=	01101111	=	6F	=	111
r	=	01110010	=	72	=	114
l	=	01101100	=	6C	=	108
d	=	01100100	=	64	=	100

Note: 12 characters – requires 12 bytes

Each character requires 1 byte

	000	001	010	011	100	101	110	111
0000	NULL	DLE		0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EDT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Numeric codes

“4+15” Example

		Binary		Hexadecimal		Decimal
4	=	00110100	=	34	=	52
+	=	00101011	=	2B	=	43
1	=	00110001	=	31	=	49
5	=	00110101	=	35	=	53

“4+15” is represented as
“00110100 00101011 00110001 00110101”

or “$34_{16}2B_{16}31_{16}35_{16}$”

OÉ Gaillimh
NUI Galway

	000	001	010	011	100	101	110	111
0000	NULL	DLE		0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EDT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Punctuation, etc.

Common Control Codes

OÉ Gaillimh
NUI Galway

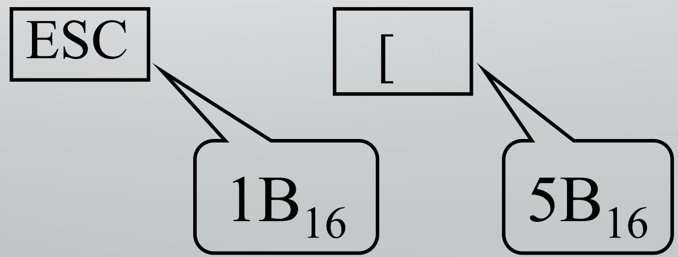
CR	0D	carriage return
LF	0A	line feed
HT	09	horizontal tab
DEL	7F	delete
NULL	00	null

OÉ Gaillimh
NUI Galway

	000	001	010	011	100	101	110	111
0000	NULL	DLE		0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EDT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Escape Sequences

- Extend the capability of the ASCII code set
- For controlling terminals and formatting output
- Defined by ANSI in documents X3.41-1974 and X3.64-1977
- The escape code is ESC = $1B_{16}$
- An escape sequence begins with two codes:
- Example:
  - Erase display: ESC [2 J
  - Erase line: ESC [K



Unicode (1)

- The extended version of the ASCII character set is not enough for international use.
- The Unicode character set uses 16 bits per character. Therefore, the Unicode character set can represent 2^{16}, or over 65 thousand, characters.
- Unicode was designed to be a superset of ASCII. That is, the first 256 characters in the Unicode character set correspond exactly to the extended ASCII character set.

Unicode (2)

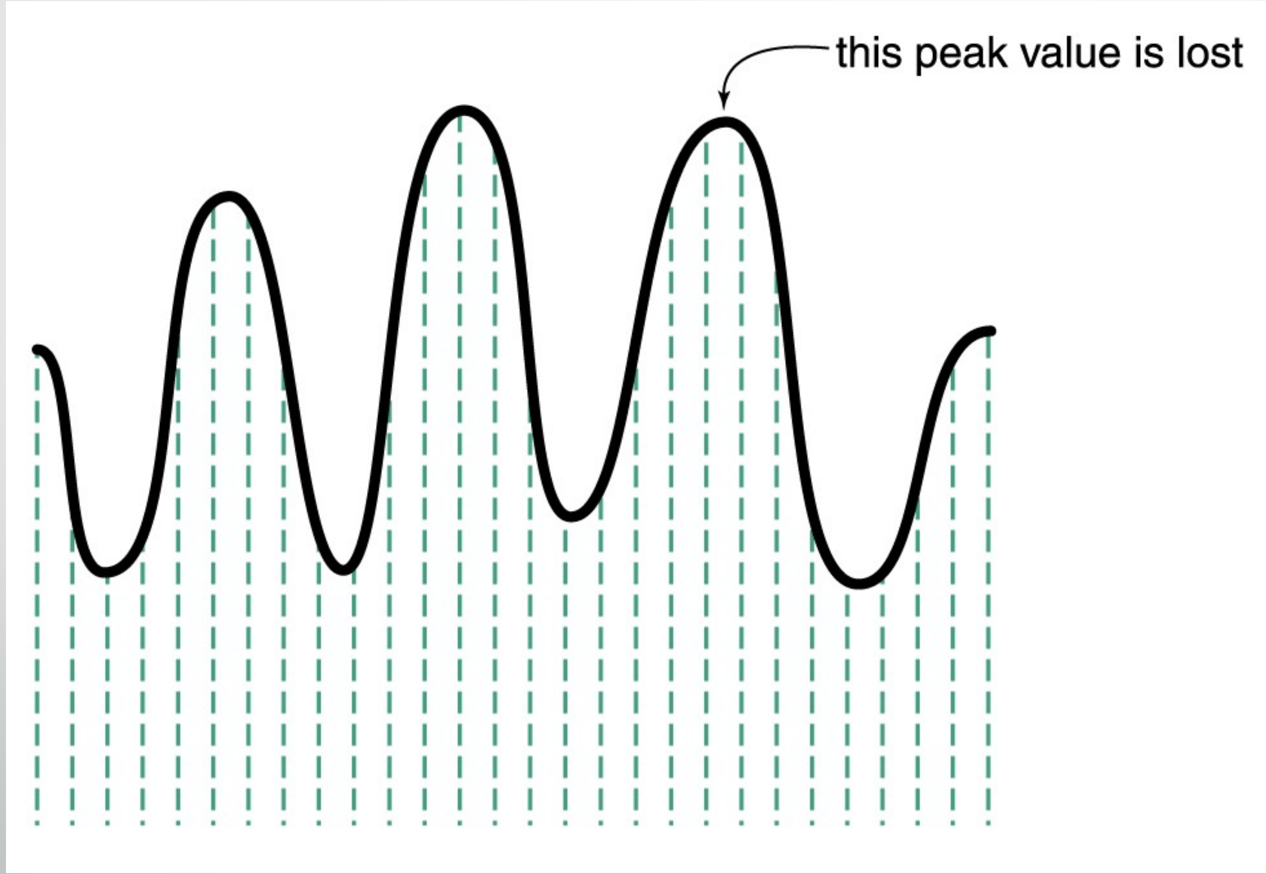
- Current version: Unicode 13 (March 2020)
- Added characters brings total to 143,859. These additions include various new scripts and new emoji characters.
- The new scripts and characters add support for lesser-used languages worldwide.

http://www.unicode.org

Audio Information Representation (1)

- Sound is perceived when a series of air compressions vibrate a membrane in our ear, which sends signals to our brain
- A stereo sends an electrical signal to a speaker to produce sound. This signal is an analog representation of the sound wave. The voltage in the signal varies in direct proportion to the sound wave
- To digitize the signal we periodically measure the voltage of the signal and record the appropriate numeric value. The process is called *sampling*
- In general, a sampling rate of around 40,000 times per second is enough to create a very good high-quality sound reproduction

Audio Information Representation (2)



Sampling an audio signal

Audio Formats

- Several popular formats are: WAV, AU, AIFF, VQF, and MP3. Currently, the dominant format for compressing audio data is MP3.
- MP3 is short for MPEG-2, audio layer 3 file.
- Compressed formats usually employ both ***lossy*** and ***lossless*** compression.
  - Analyzes the frequency spread and compares it to mathematical models of human psychoacoustics (the study of the interrelation between the ear and the brain) and it discards information that can't be heard by humans.
  - Then the bit stream is compressed using a form of Huffman encoding to achieve additional compression.

- CT101 -
Computing Systems

Dr. Fatemeh Ahmadi Zeleti

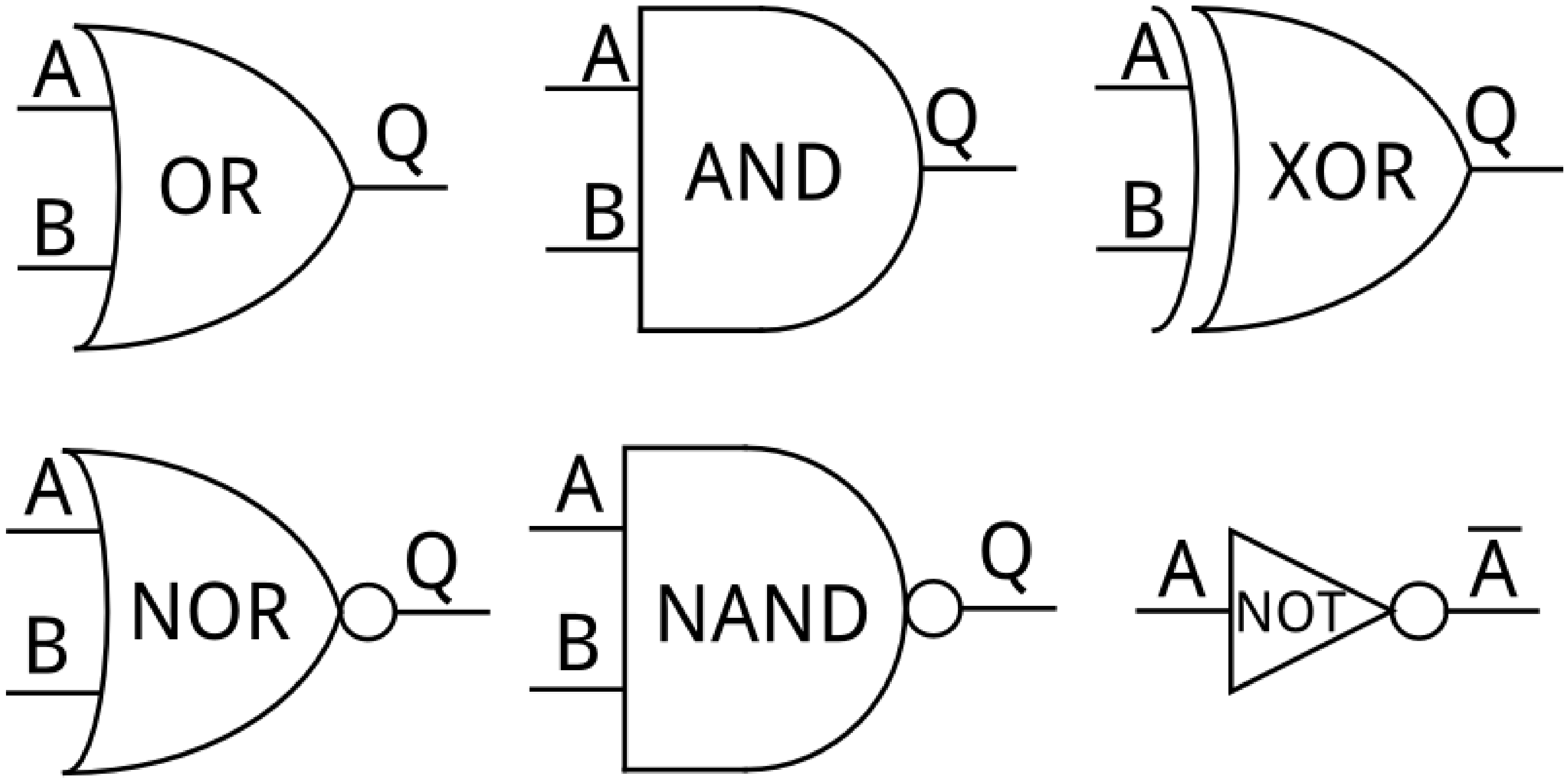
Fatemeh.ahmadizeleti@NUIGalway.ie

School of Computer Science,

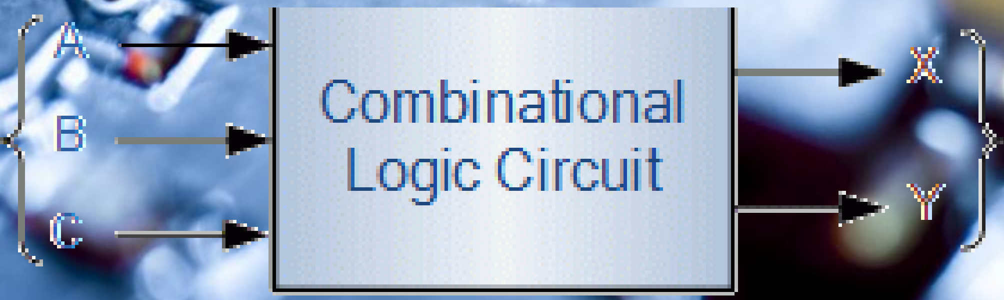
National University of Ireland, Galway

Combinational Logic Design

Digital Logic (Covered)



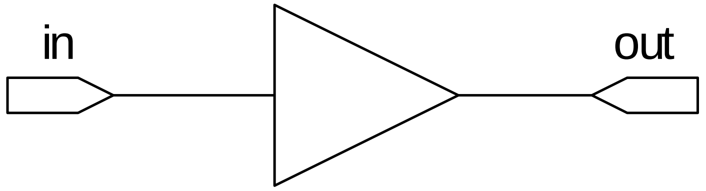
Combinational Circuits



Buffers

- A digital buffer (or a voltage buffer) is **an electronic circuit element that is used to <u>boost power without changing voltage waveform</u>**

- **Used to isolate the input from the output**, providing either no voltage or a voltage that is same as the input voltage

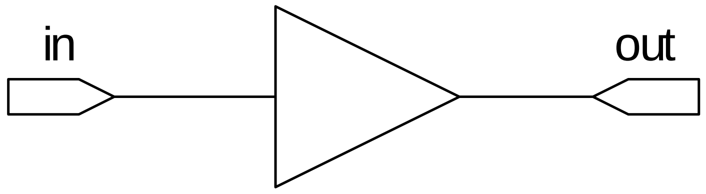


Buffers

- **Regular buffer** - always passes the input to the output
  - Its purpose being to *boost the signal* of the input to a higher level (maintain 0 or 1 values to ensure that the system performs properly)

- It will introduce a delay (as any other gate), known as **propagation time** through buffers.
  - If they are not used wisely, they can be a dangerous source of hazard in digital logic circuits

in	out
0	0
1	1



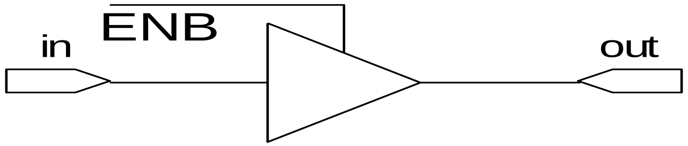
Buffers

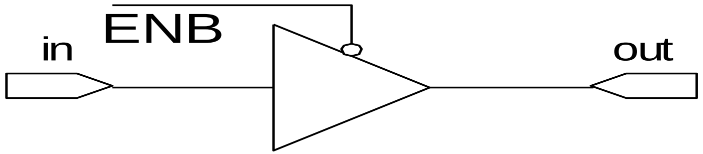
- **The tri-state buffer**: it has a data input, just like regular buffers, but also has an ENABLE/CONTROLLER input.
  - If ENB=1 then the buffer is enabled (input is passed to output)
  - if ENB=0, the buffer is disabled (regardless of the input, output will be in a high impedance state Z)

- **High Impedance/Resistance State**
  - I = V/R (Ohm Law) if R (impedance) -> very big then the I (current) goes nearly to zero (I-> 0)
  - They can be disabled to essentially break connections.

in	ENB	out
x	0	Z
0	1	0
1	1	1

in	ENB	out
x	1	Z
0	0	0
1	0	1

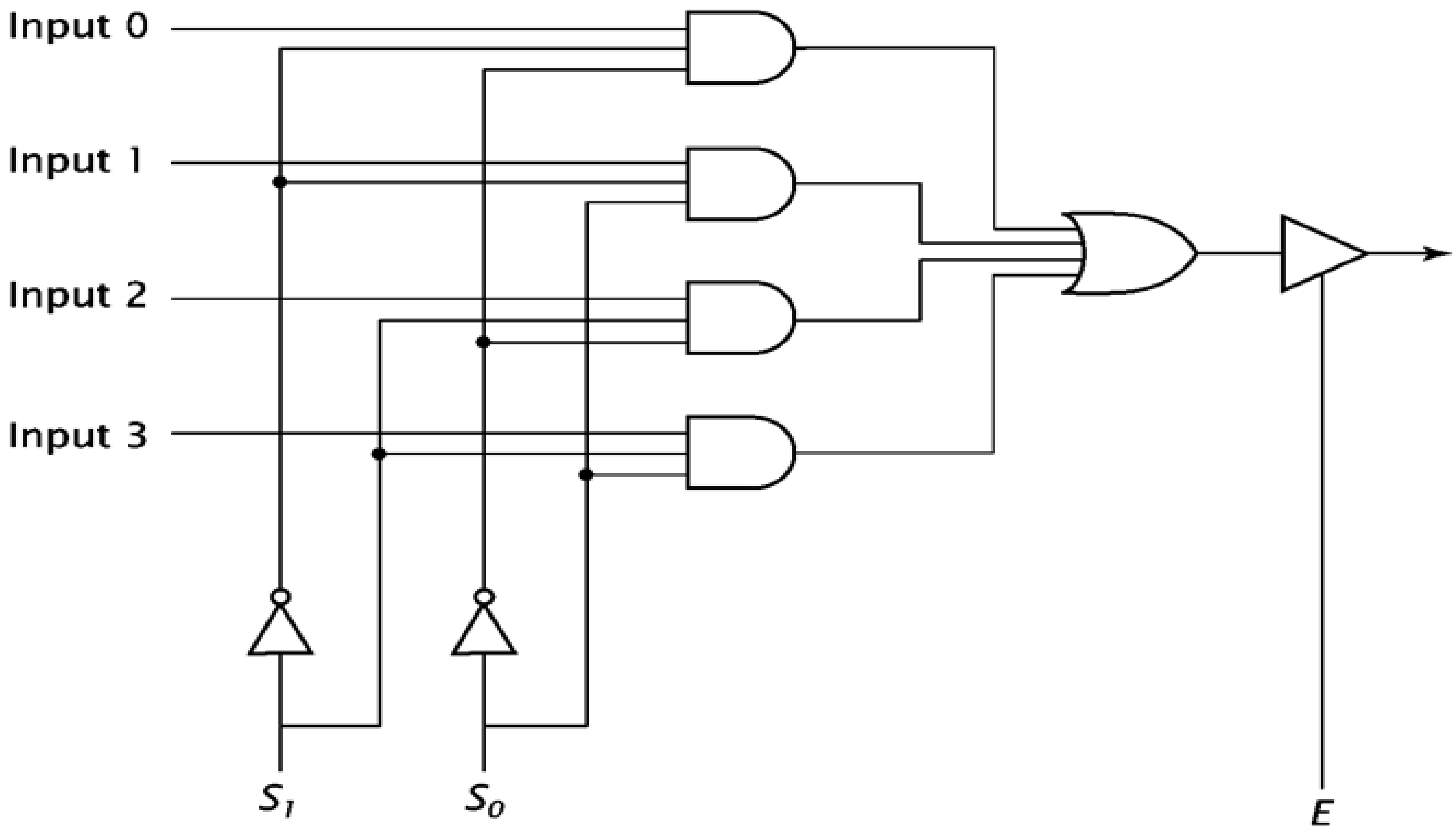




Multiplexers

- It is a combinational circuit that selects binary information from one of the input lines and directs it to the output line
- It is a selector.
  - Chooses one of its data inputs and passes it to the output according to some other selection inputs
- Consider four binary data inputs as inputs of a multiplexer.
  - Two select signals will determine which of the four inputs will be passed to the output.
- Figure (a) presents the internal structure of a four inputs multiplexer, b and c present the multiplexer schematic representation with active high enable signal (b) and active low enable signal (c)

Multiplexer



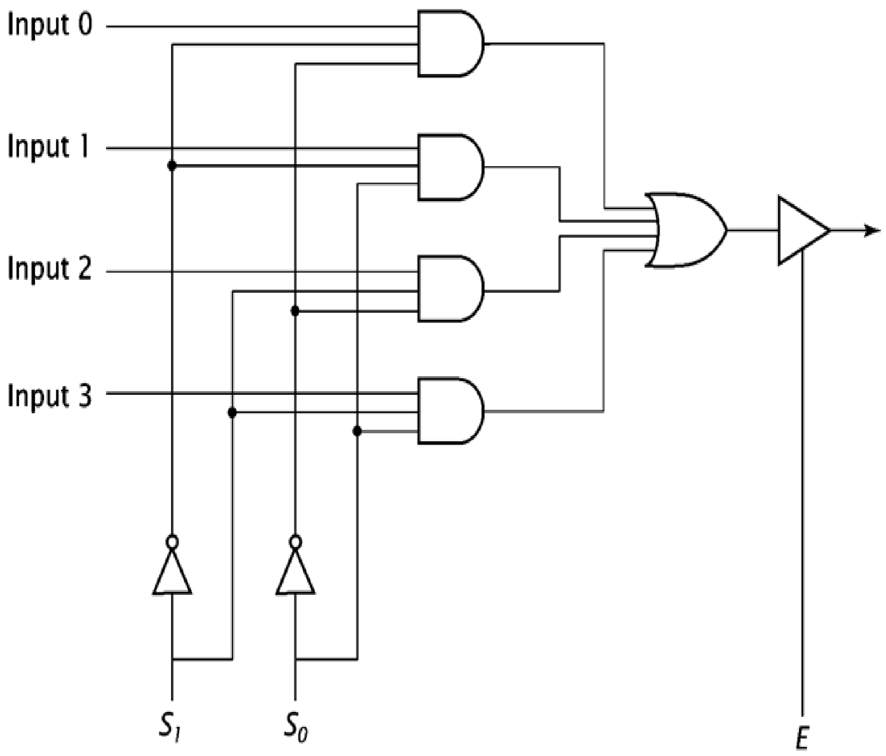
(a)

Multiplexer internal structure

Multiplexer

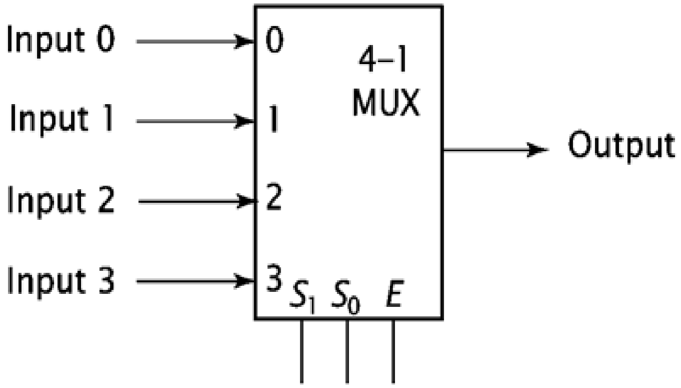
- The four AND gates include the following pair of inputs, besides the data inputs: S1’S0’, S1’S0, S1S0’ and S1S0.
- If S1=0 and S0=0 then the inputs at the top AND gate are input0, S1’(1) and S0’(1). The output of this AND gate is the value of the Input0. The other three AND gate inputs are either S1 or S0 or both … that means that the output of those gates is zero. The inputs of the OR gate are Input0 and three zeros => the output is Input0
- Setting S1 and S0 to 01, 10 or 11 produces outputs of the value of Input1, Input2, respectively Input3
- Finally the values are passed to a tri-state buffer. If the buffer is Enabled (E=1) than the value is passed to the output of the multiplexer. Otherwise, the output is high impedance value Z.

- To summarise, the values for S1 and S0 will decide which input is chosen.



(a)

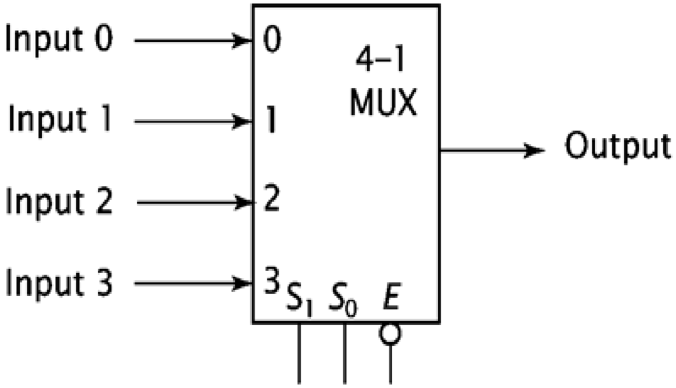
Multiplexers



(b)

S_1	S_0	E	Output
X	X	0	Z
0	0	1	Input 0
0	1	1	Input 1
1	0	1	Input 2
1	1	1	Input 3

Multiplexer schematic representation with <u>**active high** enable signal</u>



(c)

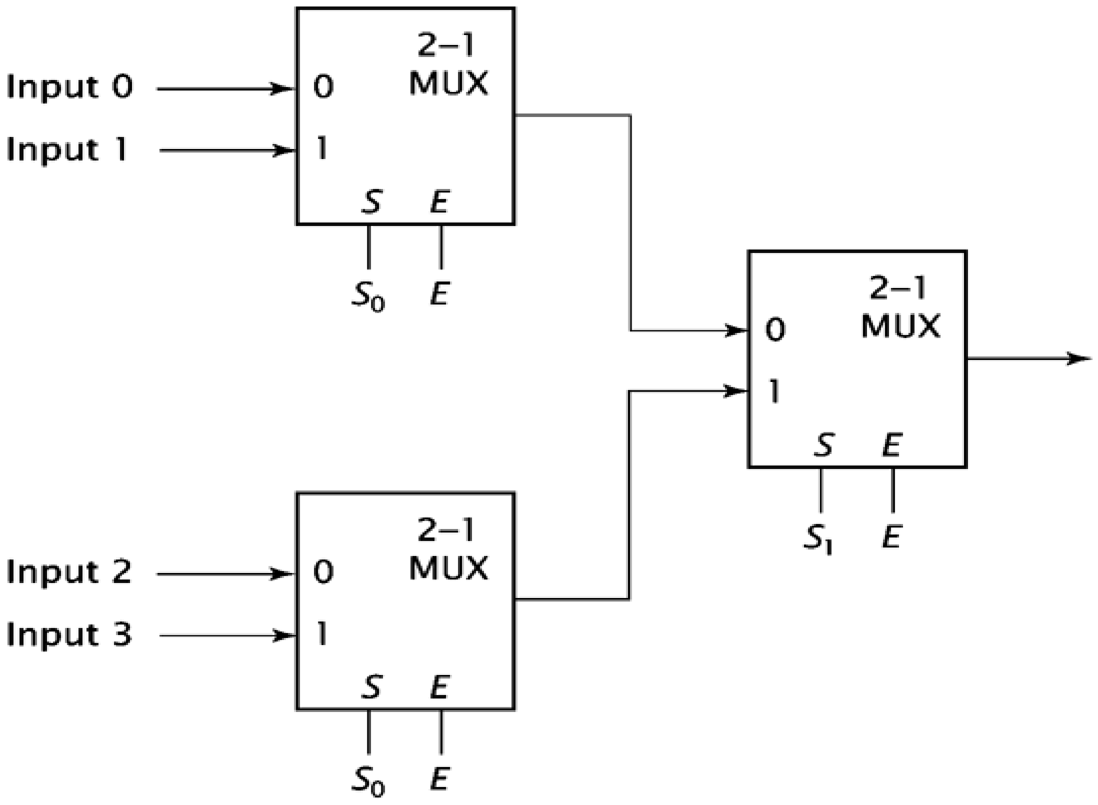
S_1	S_0	E	Output
X	X	1	Z
0	0	0	Input 0
0	1	0	Input 1
1	0	0	Input 2
1	1	0	Input 3

Multiplexer schematic representation with <u>**active low** enable signal</u>

Multiplexer

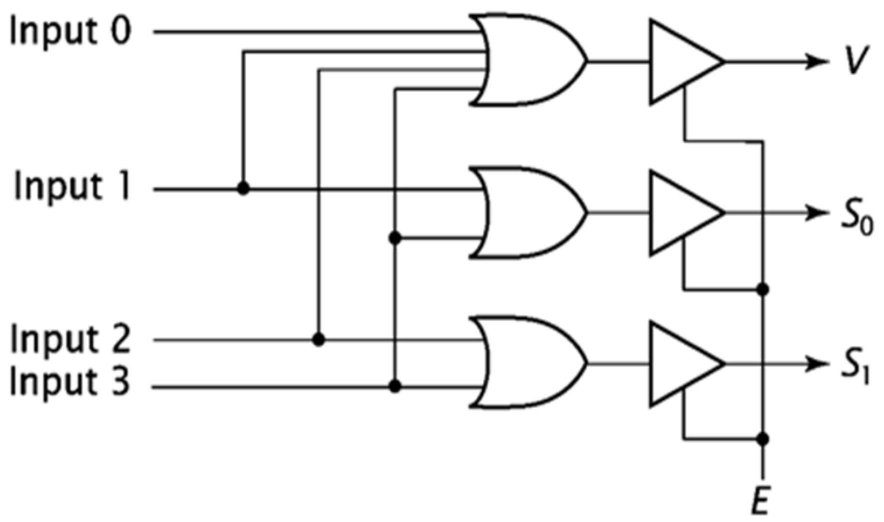
• Multiplexers can be cascaded to select from a large number of inputs

• 4-to-1 multiplexer made of 2-to-1 multiplexers



Encoders

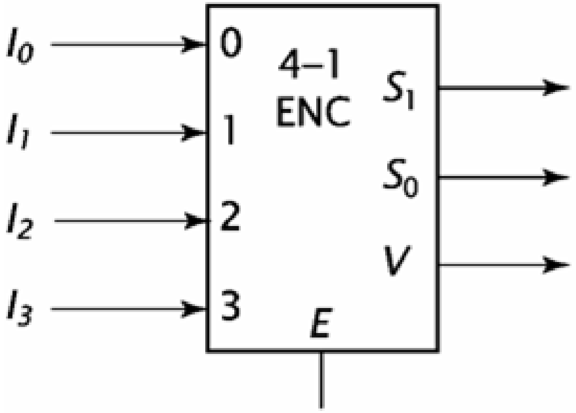
- An encoder is a circuit that changes set of signals into codes
- Encoder receives 2^n inputs and outputs a n bit value corresponding to the one input that has a value of 1
- A 4-to-2 encoder and its schematic representations are presented in (a), (b) and (c) .



(a)

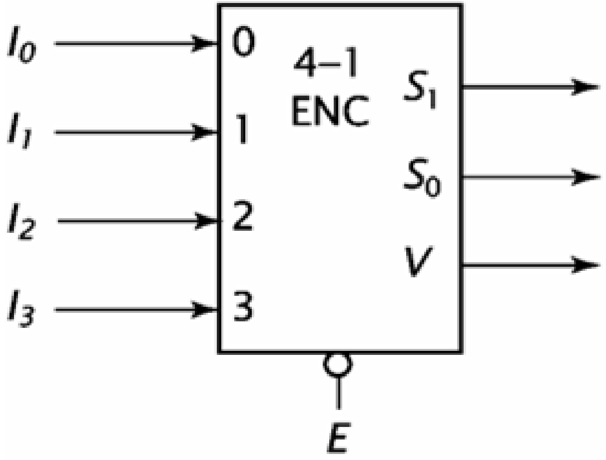
- Exactly zero or one input is active
- It will fail if more than one input is high
- The encoder will output S1 S0 = 00 if either input 0 is active or no input is active.
- The **V signal** distinguishes between these two cases

Encoders



I_0	I_1	I_2	I_3	E	S_1	S_0	V
X	X	X	X	0	Z	Z	Z
0	0	0	0	1	0	0	0
1	0	0	0	1	0	0	1
0	1	0	0	1	0	1	1
0	0	1	0	1	1	0	1
0	0	0	1	1	1	1	1

(b)



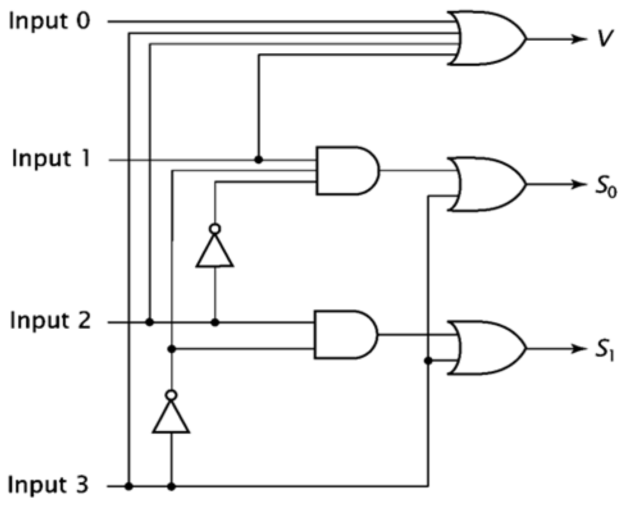
I_0	I_1	I_2	I_3	E	S_1	S_0	V
X	X	X	X	1	Z	Z	Z
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
0	1	0	0	0	0	1	1
0	0	1	0	0	1	0	1
0	0	0	1	0	1	1	1

(c)

Priority Encoders

- A priority encoder works just a regular encoder, with **one exception**: **<u>whenever one or more input is active, the output is set to correspond to the highest active input</u>**

- For example, in a 4-to-2 encoder, if Inputs 0, 1, and 3 are hight, then the S1 S0 = 11 output is set, corresponding to the input 3.



(a)

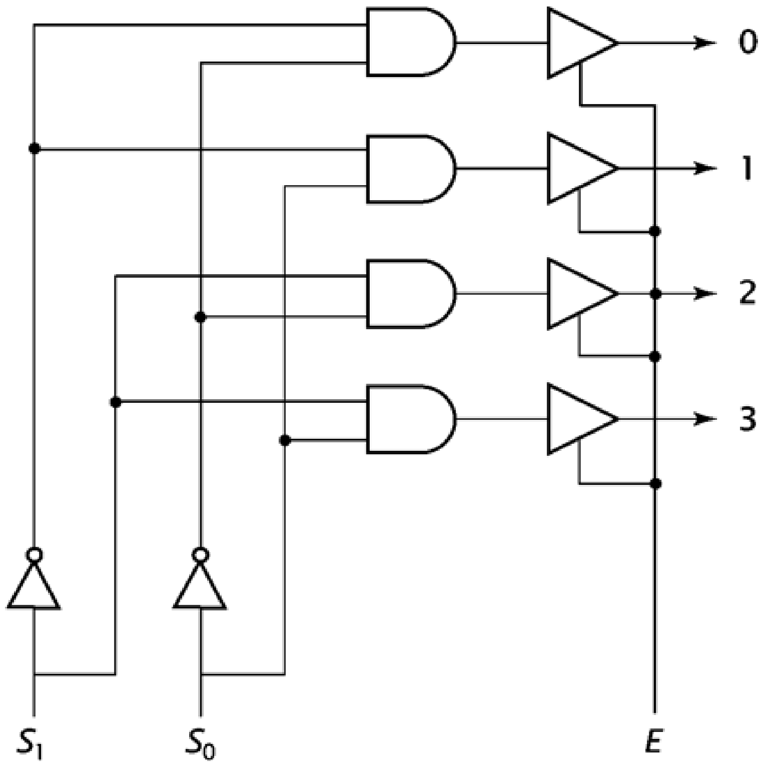
- This circuitry disables a given input if a higher numbered input is active

- This guarantees that not more than one active signal is passed to the rest of the circuitry, which can be the same as the regular encoder

Decoders

- The decoder is the exact opposite of the encoder.
- A decoder is a circuit that changes a code into a set of signals
- Accepts a binary value as input and decodes it.
  - It has n inputs and 2^n outputs, numbered from 0 to 2^n -1.
  - Each output represents one **minterm** of the inputs
- The output corresponding to the value of the n inputs is activated
  - For example, a decoder with three inputs and eight outputs will activate output 6 whenever the input values are 110.
- Figure (a) shows a two to four decoder internal structure, (b) and (c) show its schematic representation with active high enable signal and active low enable signal

Decoders



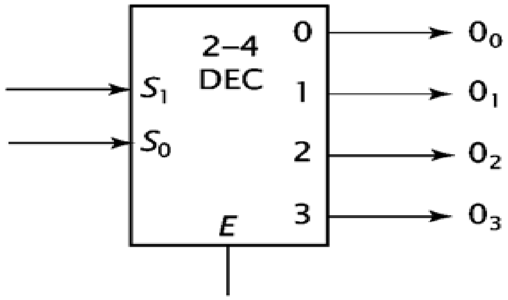
(a)

- For inputs S1 S0 = 00, 01, 10 and 11 the outputs 0, 1, 2 and respectively 3 are active

- As with the multiplexer, the output can tri-state all outputs

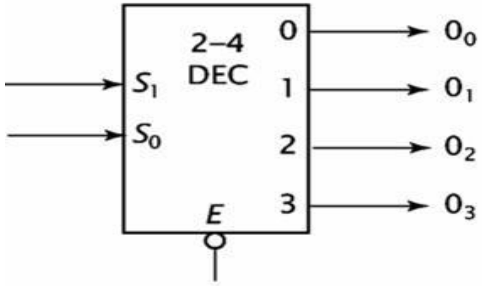
Decoders

- Can have **active high** or **active low** enable signals.
- Other variants:
  - Have active low outputs (the selected output has a value 0 and all the other outputs have a value 1)
  - Output all 0 when not enabled instead of state Z (the ones in the figure).



S_1	S_0	E	O_0	O_1	O_2	O_3
X	X	0	0	0	0	0
0	0	1	1	0	0	0
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	1

(b)

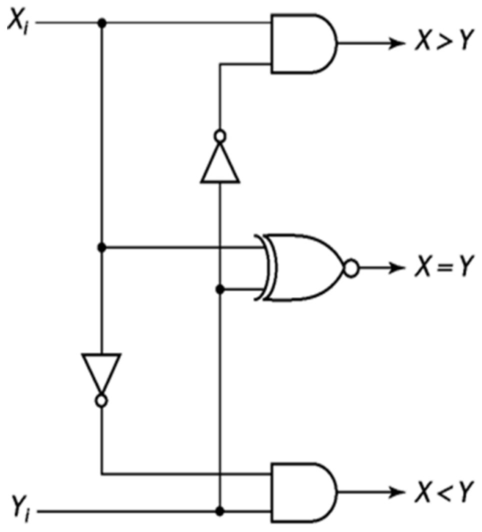


S_1	S_0	E	O_0	O_1	O_2	O_3
X	X	1	0	0	0	0
0	0	0	1	0	0	0
0	1	0	0	1	0	0
1	0	0	0	0	1	0
1	1	0	0	0	0	1

(c)

Comparators

- Compares two input values or voltages (connects analog to digital world)
- A comparator compares two n-bit binary values to determine which is greater or if they are equal (*Reference voltage/input vs. detected voltage/input*)
    - Consider the simple 1-bit comparator to illustrate the design
    - It is possible to extend the design for multi-bit numbers



(a)

X_i	Y_i	$X>Y$	$X=Y$	$X<Y$
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

(b)

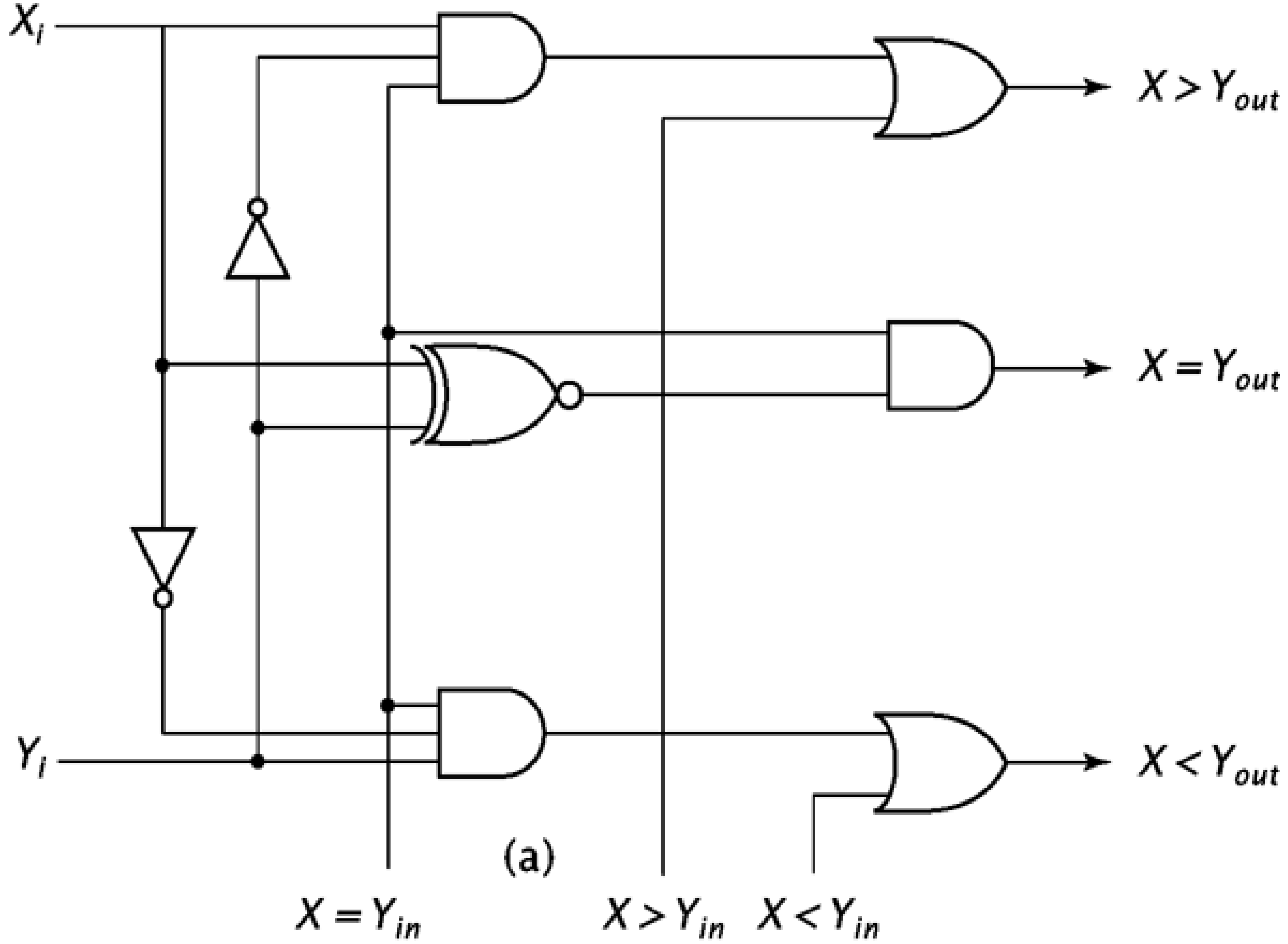
1 bit comparator

$X>Y$ only if Xi=1, Yi=0

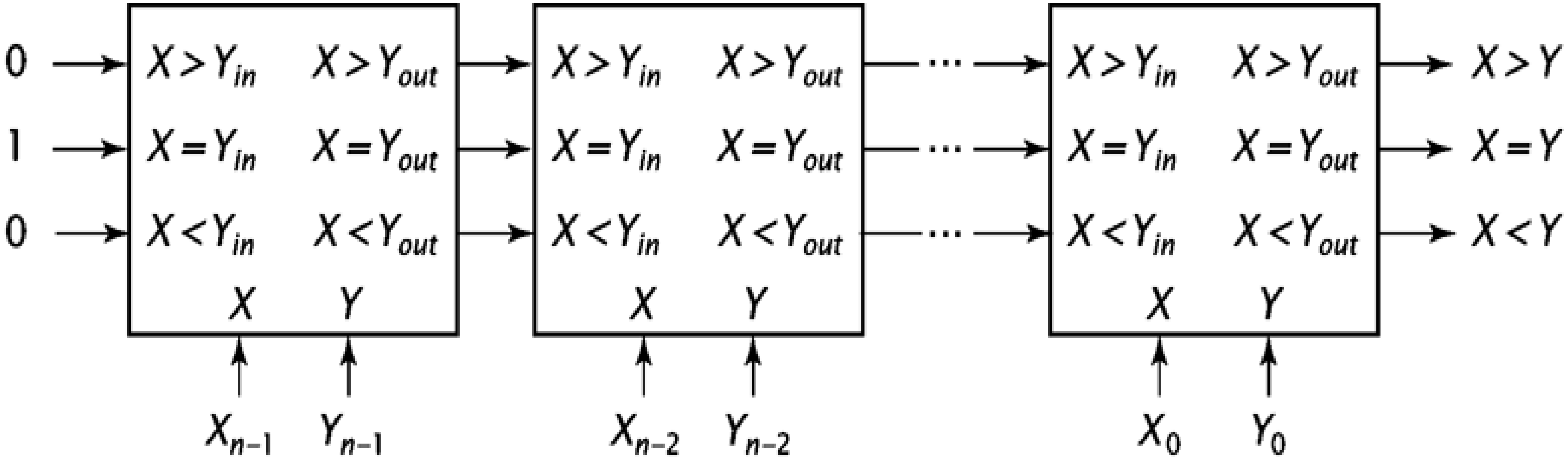
$X<Y$ only if Xi=0, Yi=1

$X=Y$ only if Xi=Yi=0 or Xi=Yi=1

1-bit comparator with propagated inputs



N bit comparator



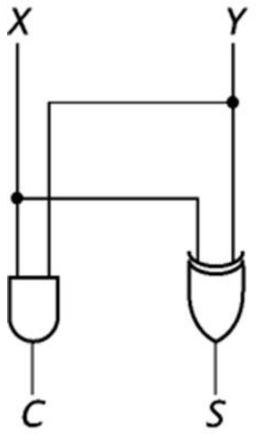
- If: X = Yin is active then the numbers are equal so far

- If X>Yin or X<Yin is active, that value is simply passed through; This corresponds to the case where we have checked the high-order bits and already know which value is larger.

Adders/Half Adder

- Used not only to perform addition but also to perform subtraction, multiplication and division

- The most basic of the adders is the half adder
  - Inputs two 1-bit value, x and y, and outputs their 2-bit sum as bits C and S
  - Bit **C** is the **carry** and bit **S** is the **sum**



(a)

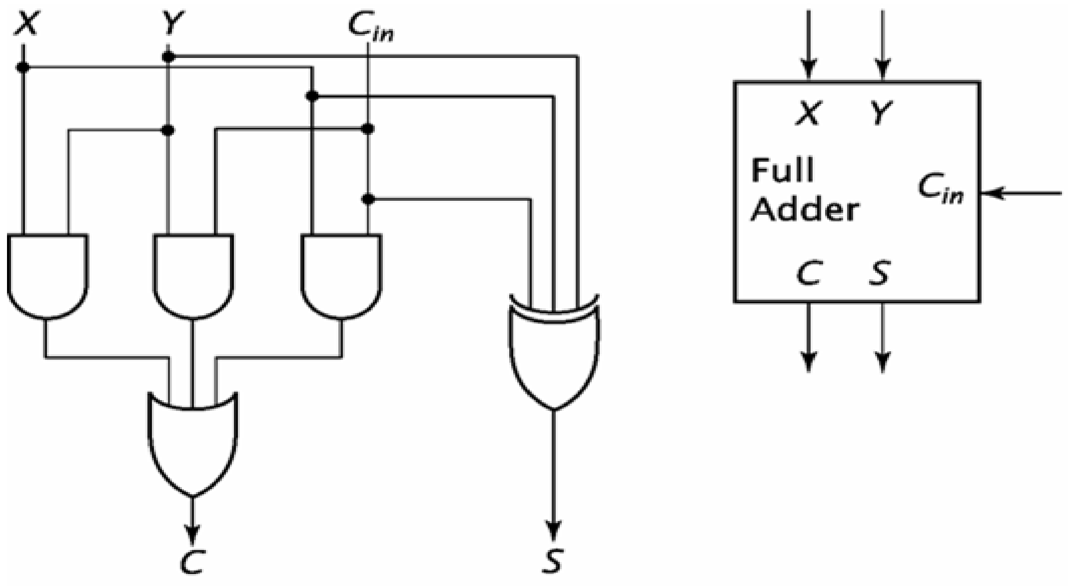
(b)

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(c)

- In real world, circuits that perform addition are more than 1 bit wide
- A wider than 1-bit adder can’t use this circuit, because there is no way to input carry information from the previous bits

Full Adder



(a) (b)

X	Y	C_{in}	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

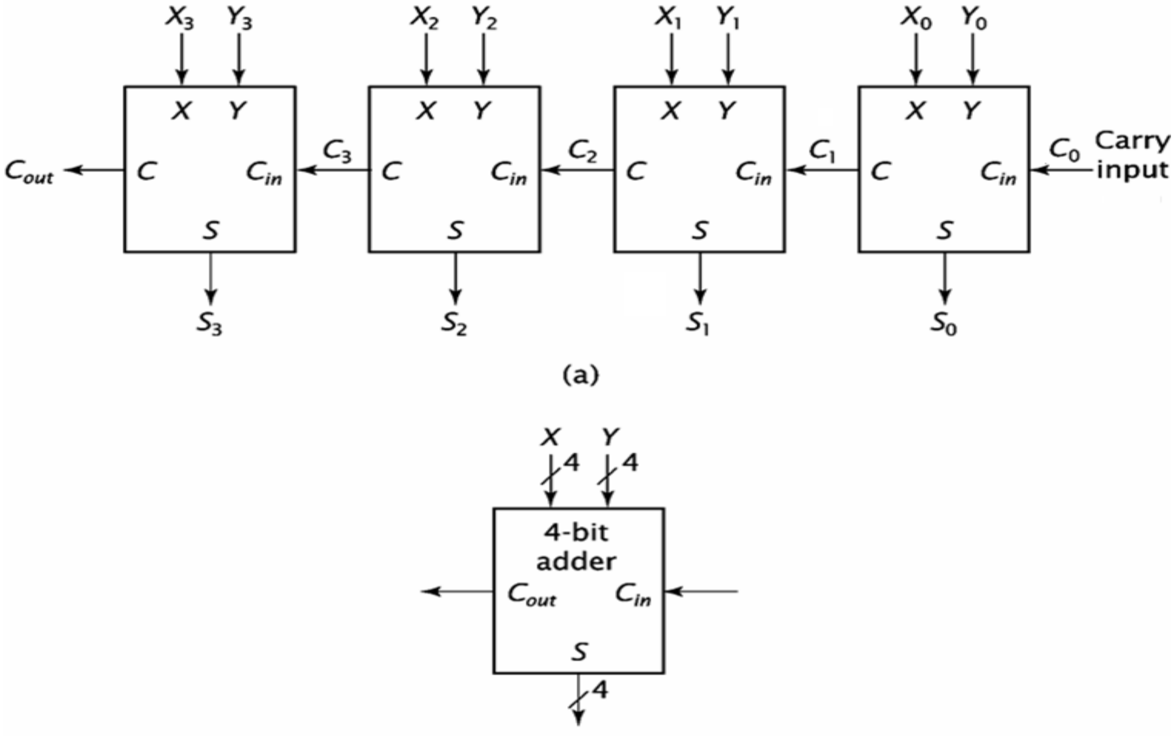
(c)

- Three inputs:
  - Two data inputs
  - One carry input
- Functions

$$S = Xin \oplus Yin \oplus Cin$$

$$C = XinYin + XinCin + YinCin$$

N-bit adders



(a)

(b)

- With the carry input, full adders can be cascaded to produce an n bit adder by connecting output C from one adder to input Cin of the next adder
- Such an adder is called **Ripple adder** (because the bits ripple through the adder). Consider the worst-case scenario (X=1111 and Y=0001) and follow the carry through the circuit
- A four-bit ripple adder is presented

Memory Circuits

- Group of circuits used to store data
  - It is not strictly combinatorial in design, but it can be used as combinatorial component in circuit design; for that reason we will include a brief presentation of the memory circuitry in this presentation

- Has some number of memory locations, each of which stores a binary value of some fixed length

- The number of locations and the size of locations is variable from memory chip to memory chip, but it is the same **within the same chip**

- The size is denoted as the number of locations times the number of bits in each location

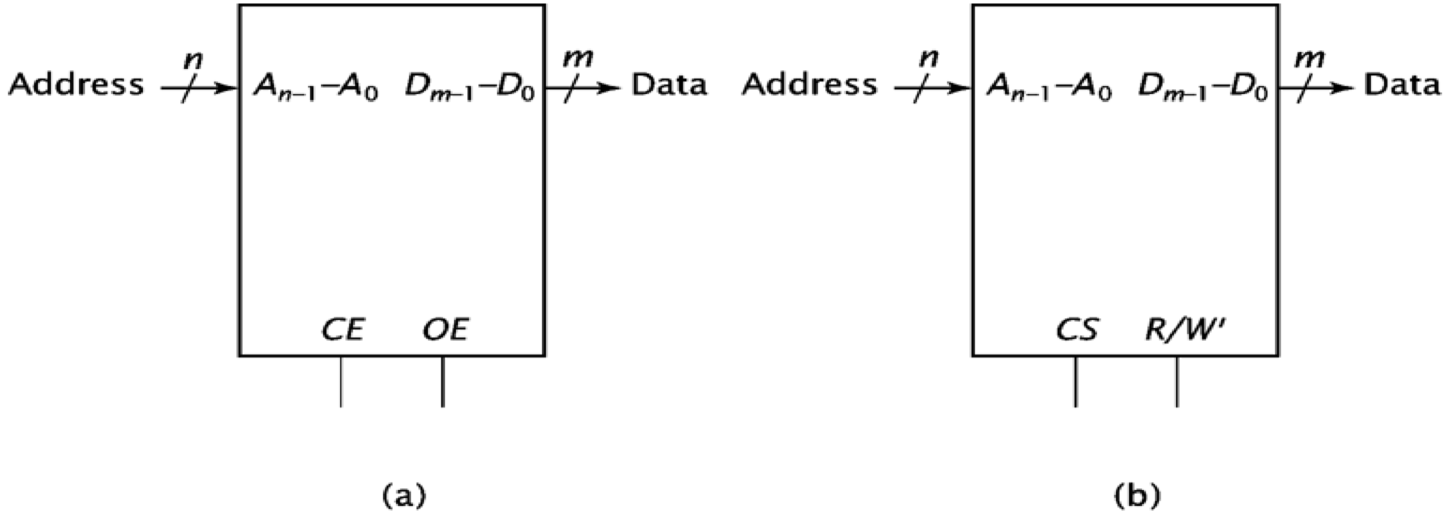
Memory

- The **address** input of a memory chip choose one of its locations.
  - A memory chip with 2^n locations requires n address inputs, usually labeled $A_{n-1}A_{n-2}$ … A_0 (512 X 8 memory has address lines $A_8A_7A_6$ … A_0)

- The **data** pins on a memory chip are used to access the data. There is one pin per bit in each location.
  - For chips with m bits per location, these pins are $D_{m-1}D_{m-2}$ … D_0 (512 X 8 memory has address lines $D_7D_6D_5$ … D_0)

- Other pins:
  - Chip enable (CE) enables or disables the chip. When disabled, the data pins output the high impedance Z; CE may be active *high* or *low*
  - Some other type of pins, dependent upon the class of the memory

Memory



- Two main memory classes:
  - ROM (**R**ead **O**nly **M**emory) (a)
  - RAM (**R**andom **A**ccess **M**emory) (b)

ROM (Read Only Memory)

- Data is programmed into the chip using an external ROM programmer
  - The programmed chip is used as a component in the circuit
  - The circuit doesn't change the content of the ROM
- Can be used as lookup tables to implement various Boolean functions – can be used implement CLCs
- Used by PCs to store the instructions that form their ***B***asic ***I***nput/***O***utput ***S***ystem (BIOS)
- **When power is removed from a ROM chip, the information is not lost, so it is a non-volatile type of memory**
- It has an OE (Output Enable) specific control pin. Both OE and CE must be enabled in order for the ROM to output data; otherwise its data output is tri-stated.

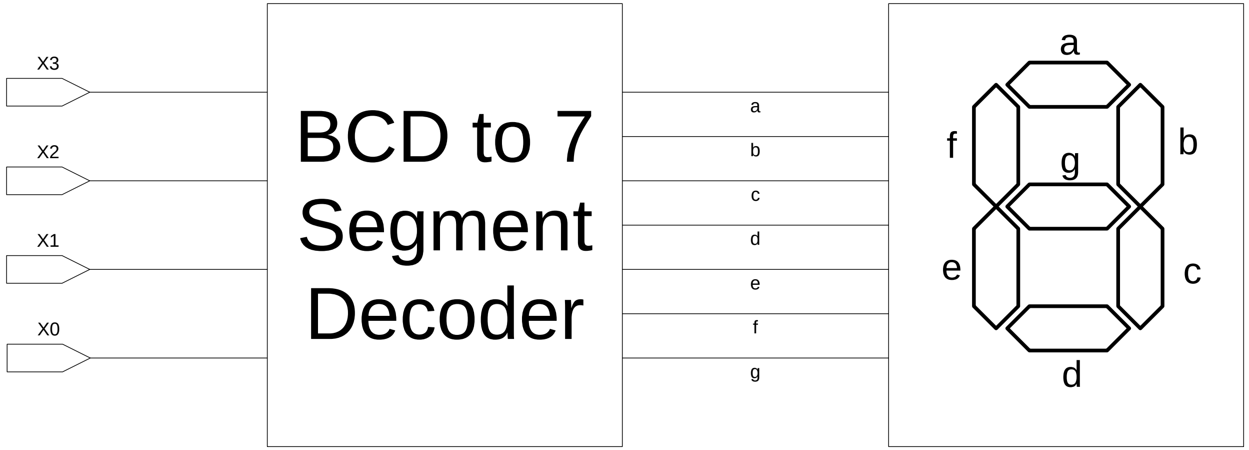
RAM (Random Access Memory)

- Read/write memory, that initially doesn't contain any data
- The computing system that it is used in usually stores data at various locations to retrieve it later from these locations
- Its data pins are bidirectional (data can flow into or out of the chip via these pins), in contrast to those of ROM that are output only
- **It loses its data once the power is removed, so it is a volatile memory**
- It has a directional select signal R/W'; When R/W'=1, the chip outputs data to the rest of the circuit; when R/W' = 0 it inputs data from the rest of the circuit

Application of the Combinatorial Circuit Design (LED Display)

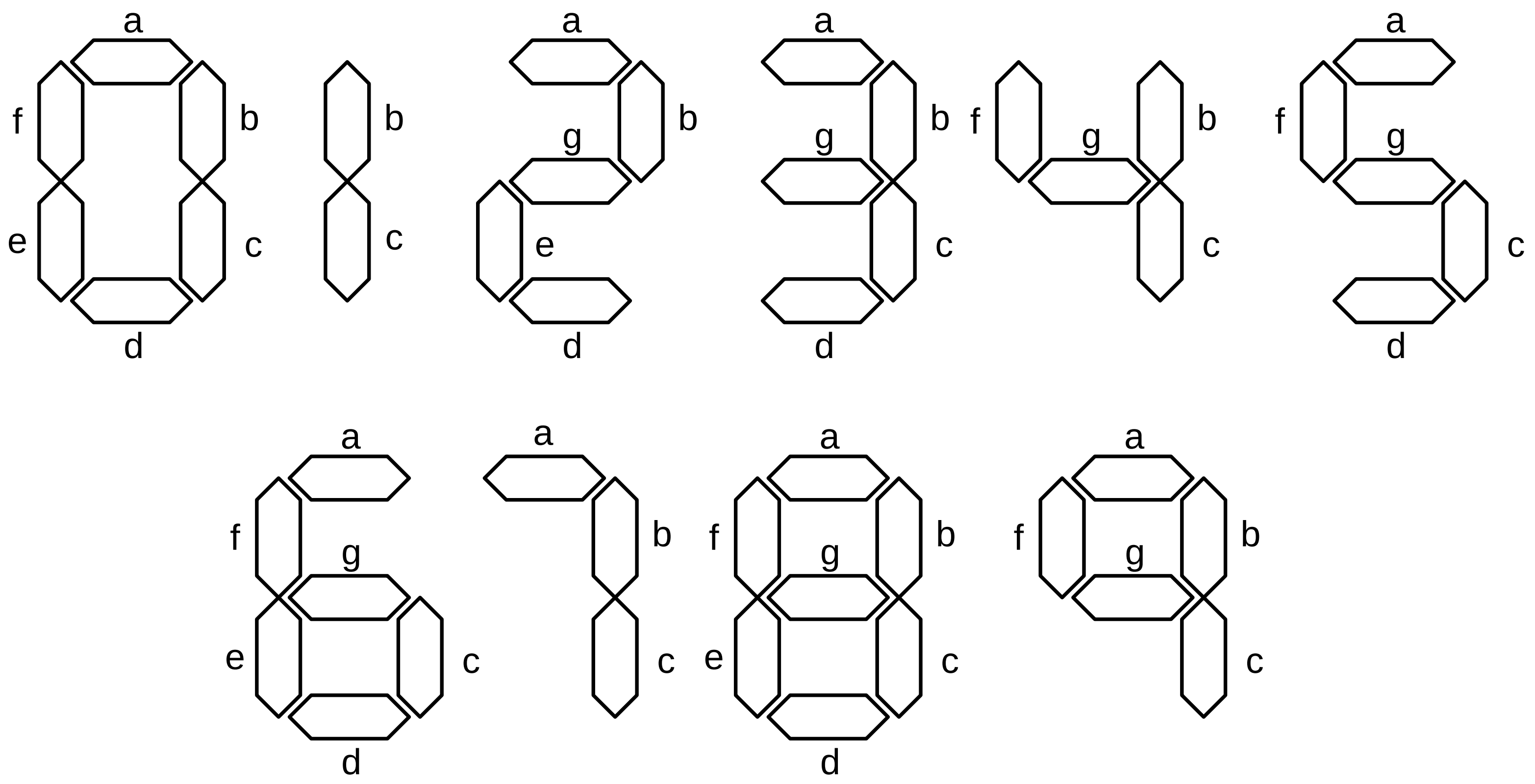
- Some useful components can be designed using the gates and the components described so far during the course
- This part describes the design of a binary coded decimal (BCD) to 7 segment decoder, which is used in **digital displays**
- This design will use only combinatorial logic gates, making use of the minimization logic techniques we have described
- Alternative design can be done using lookup tables for each logical function stored in ROM

Design Requirements for
7 segments display decoder



Design the logic circuitry that will drive a seven segment **LED display** and will be able it to represent numbers from 0 to 9

Possible numbers and their representation on 7 segment display



Truth Table (Encoder and Decoder for the LED Display)

0001 binary to decimal = 1

	X3	X2	X1	X0	a	b	c	d	e	f	g
0000 – Displays 0	0	0	0	0	1	1	1	1	1	1	0
0001 – Displays 1	0	0	0	1	0	1	1	0	0	0	0
	0	0	1	0	1	1	0	1	1	0	1
	0	0	1	1	1	1	1	1	0	0	1
	0	1	0	0	0	1	1	0	0	1	1
	0	1	0	1	1	0	1	1	0	1	1
	0	1	1	0	1	0	1	1	1	1	1
	0	1	1	1	1	1	1	0	0	0	0
	1	0	0	0	1	1	1	1	1	1	1
1001 – Displays 9	1	0	0	1	1	1	1	0	0	1	1
	1	0	1	0	x	x	x	x	x	x	x
	1	0	1	1	x	x	x	x	x	x	x
	1	1	0	0	x	x	x	x	x	x	x
	1	1	0	1	x	x	x	x	x	x	x
	1	1	1	0	x	x	x	x	x	x	x
	1	1	1	1	x	x	x	x	x	x	x

Signal a K-map implementation

To make the K-map, look the truth table in the previous slide and see where a is 1

X3X2 \ X1X0	**00**	**01**	**11**	**10**
00	1	0	1	1
01	0	1	1	1
11	X	X	X	X
10	1	1	X	X

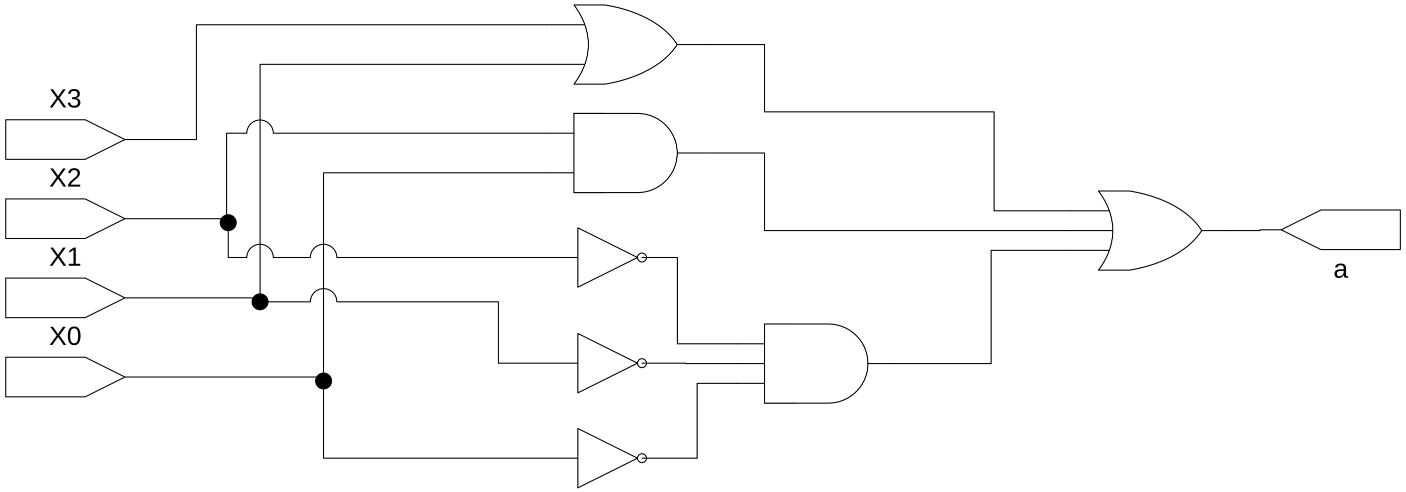
a = f(X3, X2, X1, X0) =

X3

+ X1

+ X2X0

+ X2'X1'X0'



Signal b implementation

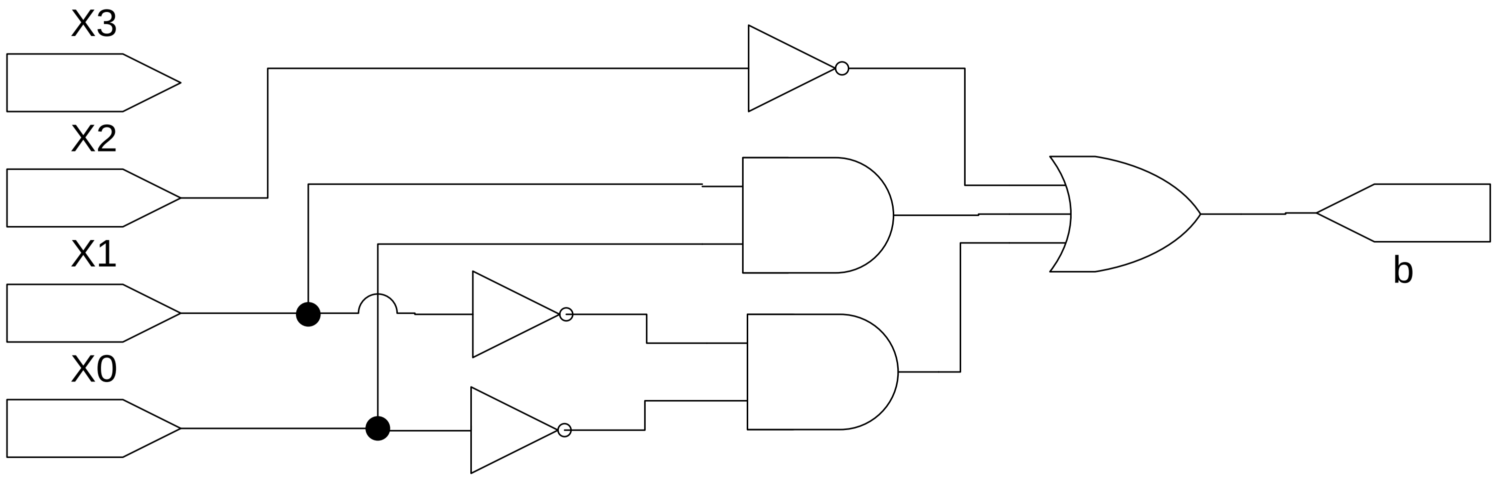
X3X2 \ X1X0	00	01	11	10
00	1	1	1	1
01	1	0	1	0
11	X	X	X	X
10	1	1	X	X

$$b = f(X3, X2, X1, X0) =$$

$$X1'X0'$$

$$+\ X1X0$$

$$+\ X2'$$



Signal c implementation

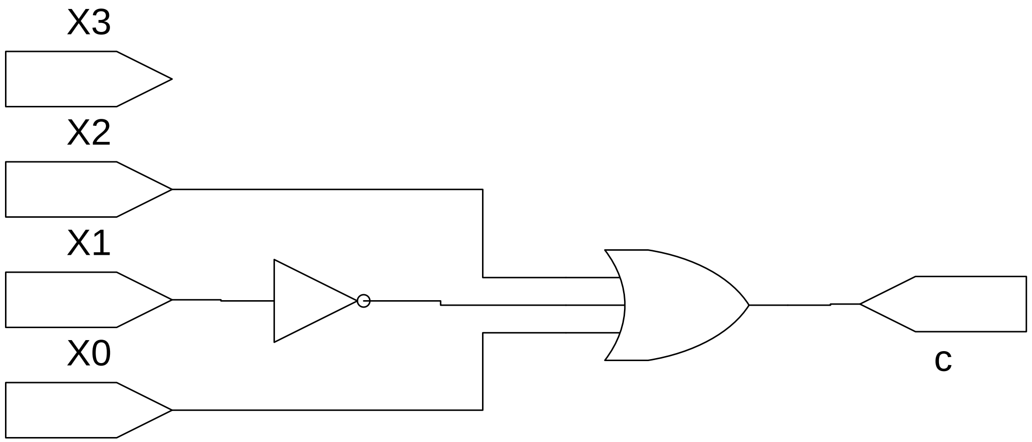
X3X2 \ X1X0	00	01	11	10
00	1	1	1	0
01	1	1	1	1
11	X	X	X	X
10	1	1	X	X

c = f(X3, X2, X1, X0) =

X1’+

+ X0

+ X2

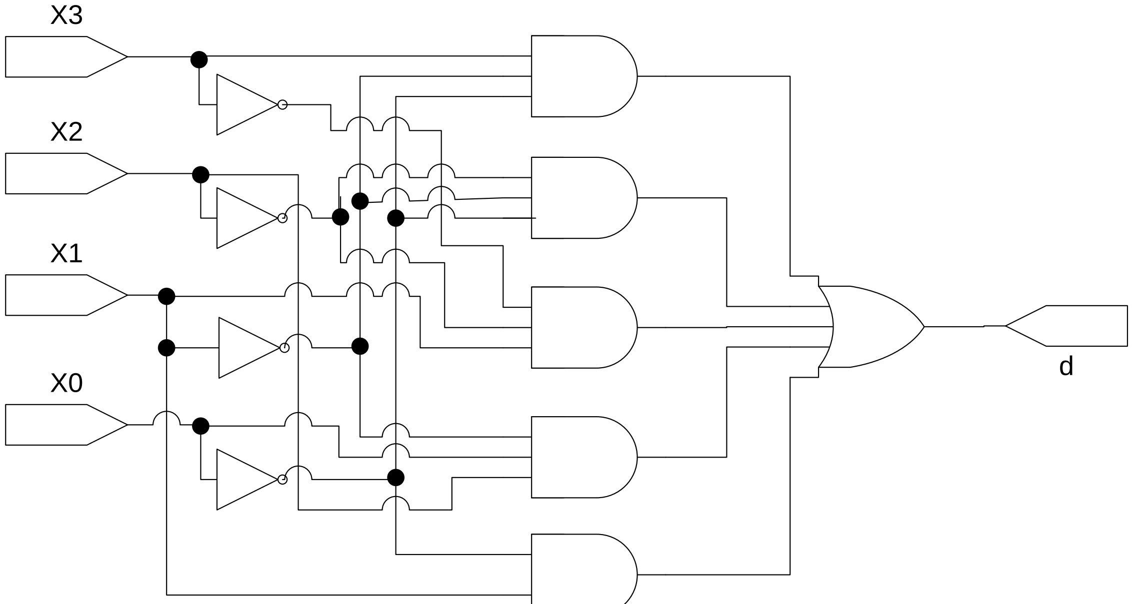


Signal d implementation

X3X2 \ X1X0	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	X	X	X	X
10	1	0	X	X

$$d = f(X3, X2, X1, X0) =$$

$$X3X1'X0' + \\ + X2'X1'X0' \\ + X3'X2'X1 \\ + X2X1'X0 \\ + X1X0'$$



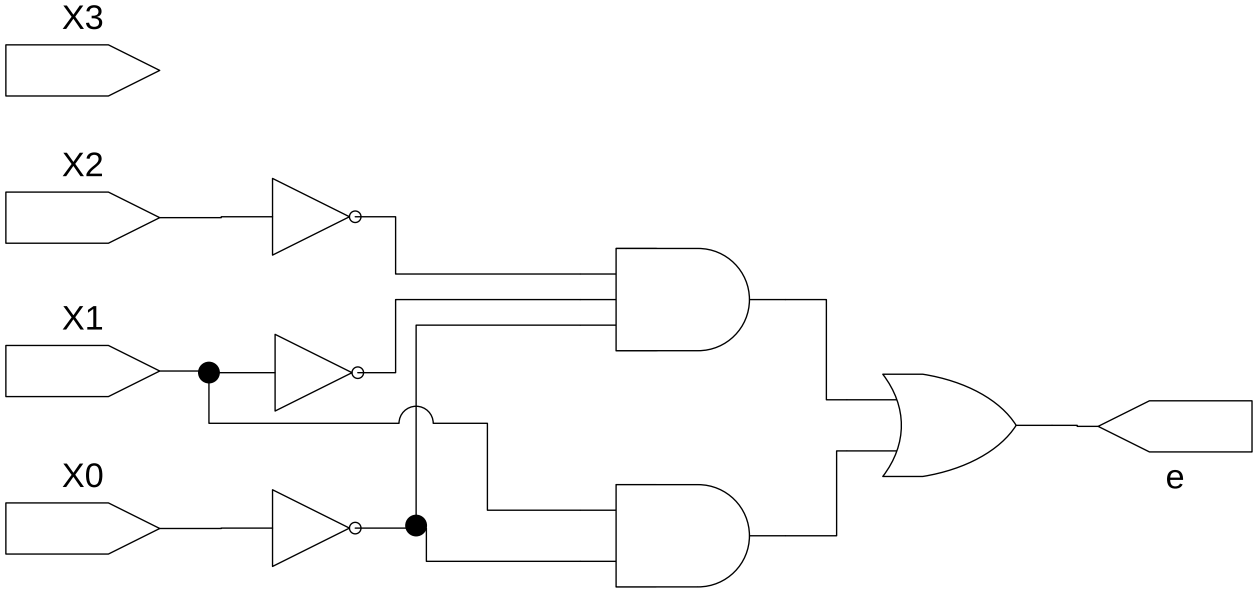
Signal e implementation

X3X2 \ X1X0	00	01	11	10
00	1	0	0	1
01	0	0	0	1
11	X	X	X	X
10	1	0	X	X

e = f(X3, X2, X1, X0) =

X1X0'

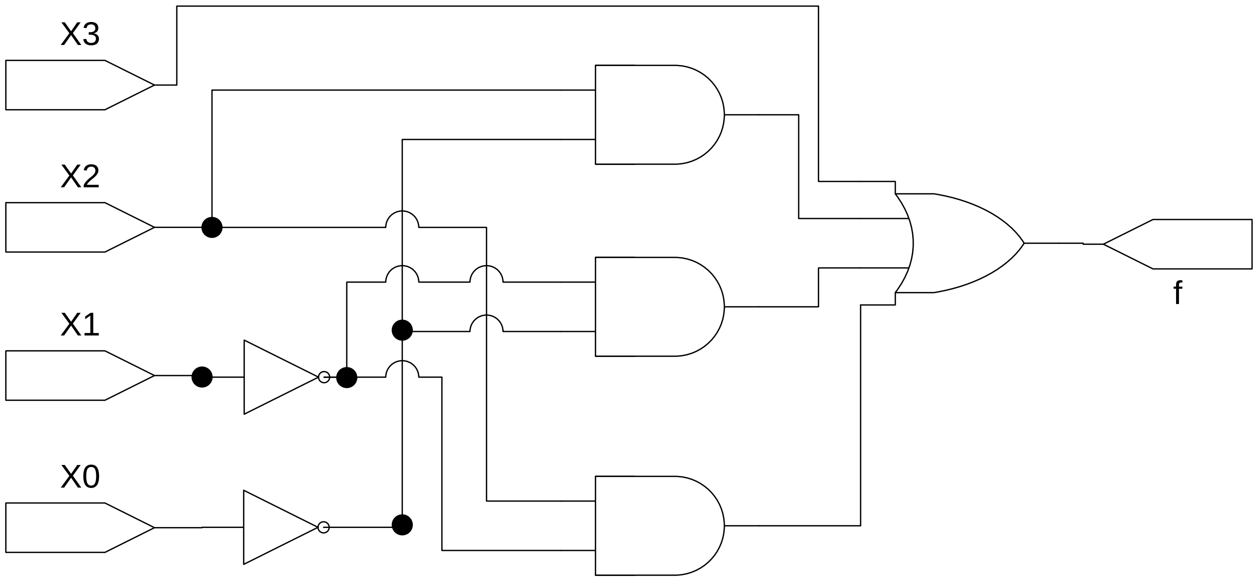
+ X2'X1'X0'



Signal f implementation

X3X2 \ X1X0	00	01	11	10
00	1	0	0	0
01	1	1	0	1
11	X	X	X	X
10	1	1	X	X

$$f = f(X3, X2, X1, X0) = X3 + X2X0' + X1'X0' + X2X1'$$



Signal g implementation

X3X2 \ X1X0	00	01	11	10
00	0	0	1	1
01	1	1	0	1
11	X	X	X	X
10	1	1	X	X

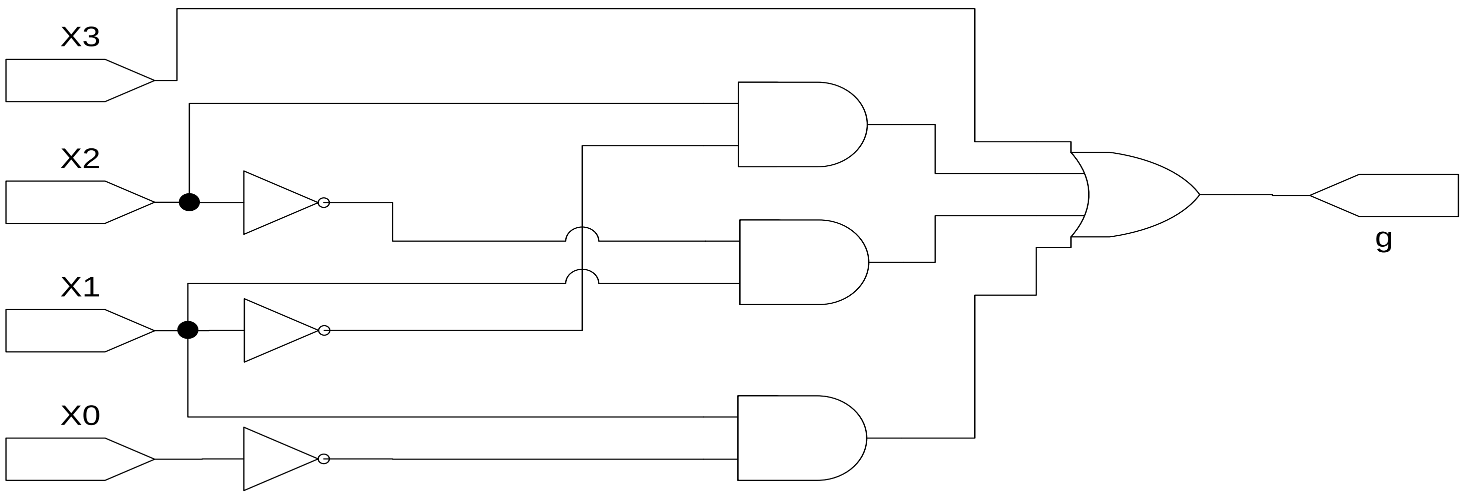
g = f(X3, X2, X1, X0) =

X3

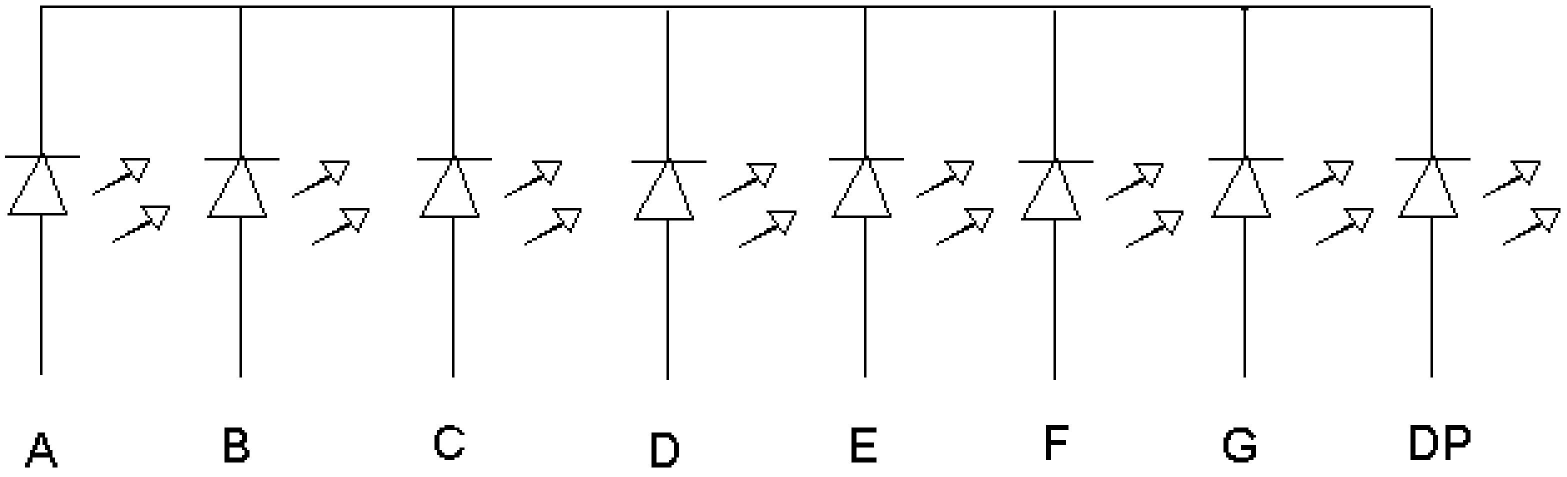
+ X1X0’

+ X2X1’

+ X2’X1



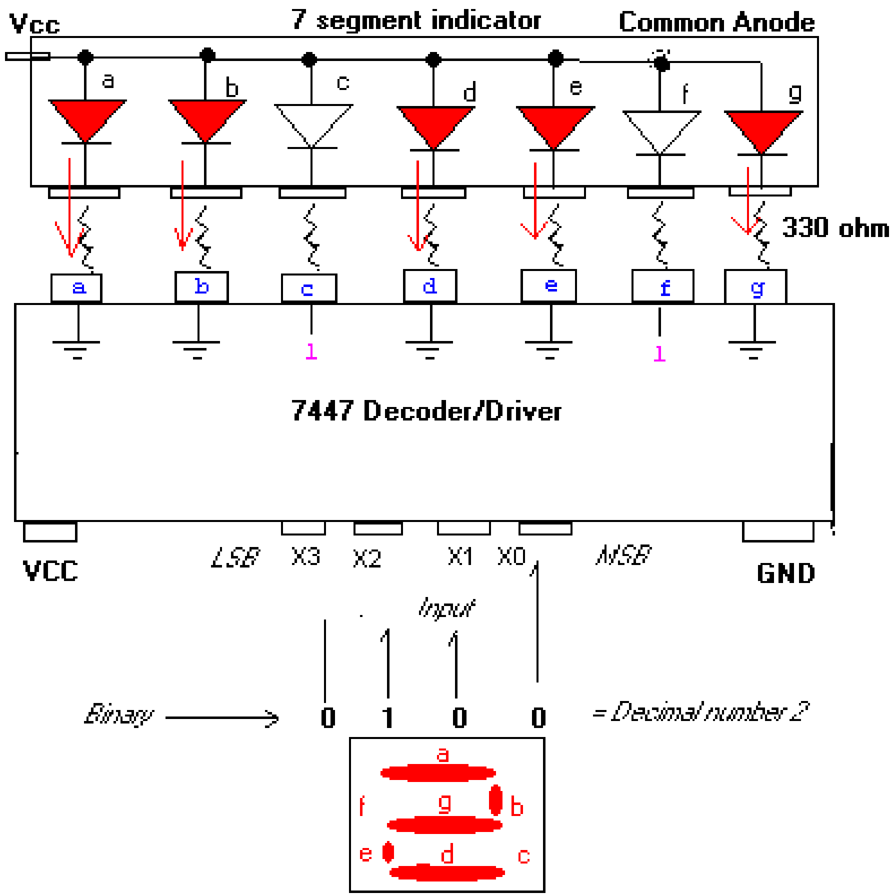
7 segment display



Common Cathode

- All the cathode of the LED are connected together
- The common connection must be grounded, and power must be applied to appropriate segment in order to illuminate that segment
- The current to light the active LED is generated by the logic component, which generates the logic 1

7447 TTL IC



- Real world example of BCD to 7 segment decoder
- Outputs of the decoder are *active low* and a common anode 7 segment display is used

References

- "Computer Systems Organization & Architecture", John D. Carpinelli, ISBN: 0-201-61253-4

OÉ Gaillimh
NUI Galway

- CT101 -
Computing Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

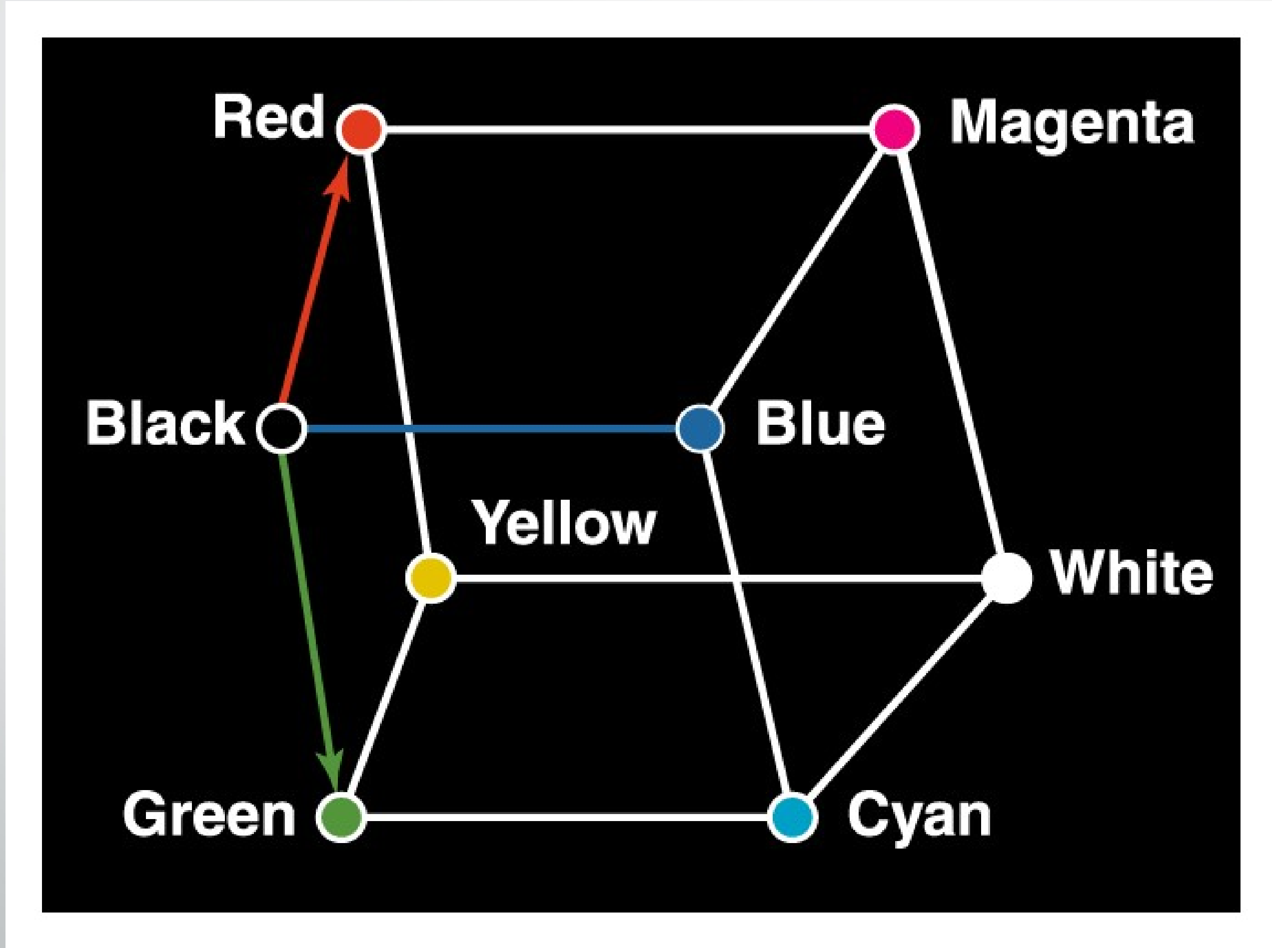
National University of Ireland, Galway

Representing Images and Graphics (1)

- Color is our perception of the various frequencies of light that reach the retinas of our eyes
- Our retinas have three types of color photoreceptor cone cells that respond to different sets of frequencies.
  - These photoreceptor categories correspond to the colors of Red, Green, and Blue
- Color is often expressed in a computer as an RGB (red-green-blue) value, which is actually three numbers that indicate the relative contribution of each of these three primary colors
- For example, an RGB value of (255, 255, 0) maximizes the contribution of red and green, and minimizes the contribution of blue, which results in a bright yellow

Representing Images and Graphics (2)

OÉ Gaillimh
NUI Galway



Three-dimensional color space

Representing Images and Graphics (3)

- The amount of data that is used to represent a color is called the **color depth**.
- **HiColor** is a term that indicates a 16-bit color depth.
  - Five bits are used for representing the R and B components.
  - Six bits are used for representing the G component, because the human eye is more sensitive to G;
- **TrueColor** indicates a 24-bit color depth. Therefore, each number in an RGB value is represented using eight bits.

Representing Images and Graphics (4)

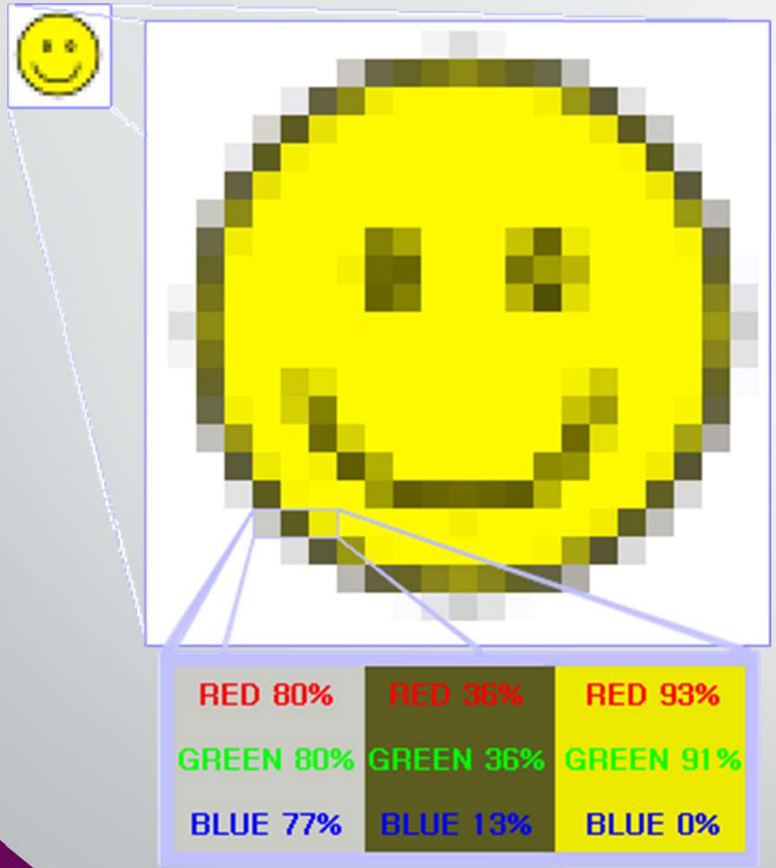
OÉ Gaillimh
NUI Galway

RGB Value			Color
Red	**Green**	**Blue**	
0	0	0	black
255	255	255	white
255	255	0	yellow
255	130	255	Pink
146	81	0	brown
157	95	82	purple
140	0	0	maroon

Digitized Images and Graphics

- Digitizing a picture is the act of representing it as a collection of individual dots called **pixels**.
- The number of pixels used to represent a picture is called the **resolution**.
- The storage of image information on a pixel-by-pixel basis is called a **raster-graphics format**.
  - Several popular raster file formats including bitmap (BMP), GIF, and JPEG.

BMP Raster Image Example



- The smiley face in the top left corner is a bitmap image.
- When enlarged, individual pixels appear as squares.
- Each pixel is described by a value for red, green and blue.

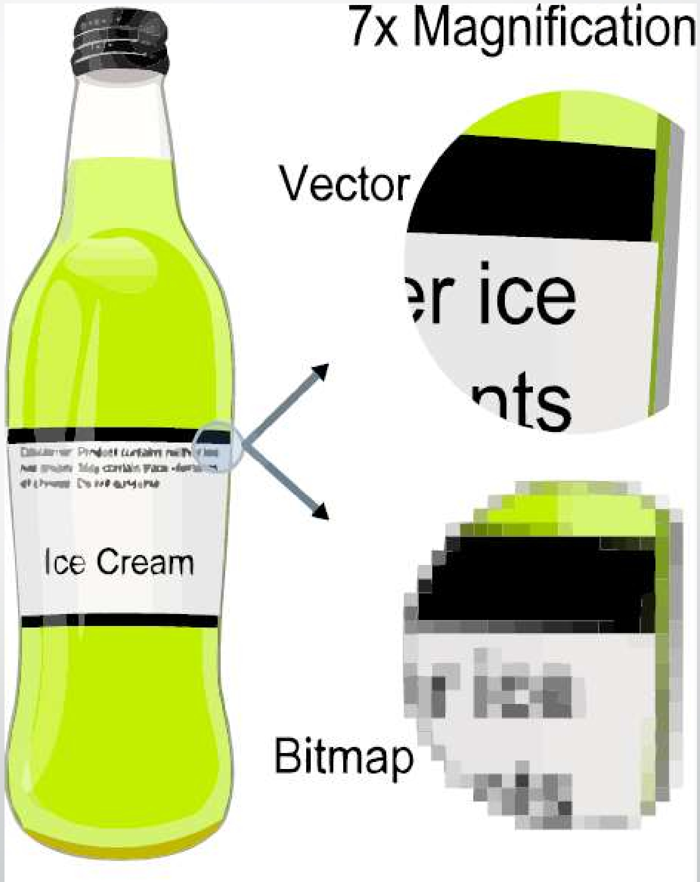
Vector Graphics

OÉ Gaillimh
NUI Galway

- Instead of assigning colors to pixels as we do in raster graphics, a vector-graphics format describe an image in terms of lines and geometric shapes.
  - A vector graphic is a series of commands that describe a line's direction, thickness, and color. The file size for these formats tend to be small because every pixel does not have to be accounted for.
- Vector graphics can be resized mathematically, and these changes can be calculated dynamically as needed.
- However, vector graphics is not good for representing real-world images.

Example of Vector Image

- Effect of vector graphics versus raster graphics.
- Magnification of 7x as a vector image vs same magnification as a bitmap image.
- Examples of vector image formats: SVG (Scalable Vector Graphics), EPS (Encapsulated Post Script), etc..



Video

- **What is video?**
  - The technology of electronically capturing, recording, processing, storing, transmitting and reconstruction a sequence of still images representing scenes in motion
  - It is a collection of still images
- **How does video camera work?**
  - The lens of the camera focuses an image onto a sensor, and the sensor converts the image into an electronic signal that is stored on disc, hard-drive, or memory card (in a compressed or raw format).
- **What about sound?**
  - Video cameras usually record sound along with images. Almost all video cameras have microphones, but even though images and sound are usually recorded to the same disc, or card they are two different types of information - so sometimes it helps to think of them separately.
  - You might record a beautiful visual scene with terrible noise, but you know that you won't use the sound. Or you might record some beautiful sound with your video camera while the lens cap is on because you just want the sound.

Representing Video

- **Frame rate:** the number of still images (or frames) recorded every second.
  - Usually frame rate is expressed in frames per second (fps)
- **Resolution:** how many pixels the image has.
  - Resolution is usually expressed by numbers for horizontal and vertical: 640 by 480 means 640 pixels wide, by 480 pixels tall.
  - Multiply the numbers and you get the total number of pixels. In this case 640x480 = 307,200.
- **Aspect Ratio:** what defines the width and height of your images.
  - *The most common aspect ratios are 3:2, 4:3, and 16:9.*
- **Compression and Format:** to save space the movie gets compressed to make it smaller.
  - The way a camera compresses the image data and records it is the recording *format.*

Representing Video

OÉ Gaillimh
NUI Galway

- A video codec Compressor/De-compressor refers to the methods used to shrink the size of a movie
  - Almost all video codecs use ***lossy*** compression to minimize the huge amounts of data associated with video.
- Two types of compression: temporal and spatial.
- **Temporal compression** looks for differences between consecutive frames. If most of an image in two frames has not changed, why should we waste space to duplicate all of the similar information?
- **Spatial compression** removes redundant information within a frame.
  - For instance, a line compression algorithm, instead of representing a white line as a series of dots with individual color info, it can represent it as how many dots of white color (saving storage space)
  - This problem is essentially the same as that faced when compressing still images.

Video Formats

- There are different layers of video transmission and storage, each with its own set of formats to choose from.
- Video gets transported via a physical connector and signal protocol ("video connection standard“)
- A given physical link can carry certain "display standards" which specify a particular refresh rate, display resolution and colour space (digital and analogue television and computer display standards).
- There are a number of standards for storage:
  - Analogue and digital tape formats
  - Digital video files can also be stored on a computer file system (with its own standards/formats) on different media (optical – DVD, Blue-ray or magnetic - HDD).
- In addition to the physical format used by the storage or transmission medium, the stream of ones and zeros that is sent must be in a particular digital video "encoding“ format (MPEG-2, MPEG-4, etc..)

Data Compression

- It is important that data be represented efficiently for two reasons: storage and transmission
- For now, we will study some common text compression techniques:
  - Keyword encoding
  - Run-length encoding
  - Huffman encoding

Keyword Encoding

OÉ Gaillimh
NUI Galway

- Frequently used words are replaced with a single character. For example:

Word	Symbol
as	^
the	~
and	+
that	$
must	&
well	%
those	#

Keyword Encoding

OÉ Gaillimh
NUI Galway

- The following paragraph:
  - The human body is composed of many independent systems, such as the circulatory system, the respiratory system, and the reproductive system. Not only must all systems work independently, they must interact and cooperate as well. Overall health is a function of the well-being of separate systems, as well as how these separate systems work in concert.

Keyword Encoding

- The encoded paragraph is:
  - The human body is composed of many independent systems, such ^ ~ circulatory system, ~ respiratory system, + ~ reproductive system. Not only & each system work independently, they & interact + cooperate ^ %. Overall health is a function of ~ %- being of separate systems, ^ % ^ how # separate systems work in concert.

Keyword Encoding

- There are a total of 349 characters in the original paragraph including spaces and punctuation. The encoded paragraph contains 314 characters, resulting in a savings of 35 characters. The compression ratio for this example is 349/314 or approximately 1.11:1.

  The space saving is 0.1003 (approx. 10%)

- **Compression Ratio** =
  - Uncompressed size / Compressed size

- **Space Saving** =
  - 1 – (Compressed size / Uncompressed size)

Run-Length Encoding (1)

- A single character may be repeated over and over again in a long sequence. This type of repetition does not generally take place in English text, but often occurs in large data streams.

- In run-length encoding, a sequence of repeated characters is replaced by a *flag character*, followed by the repeated character, followed by a single digit that indicates how many times the character is repeated.

Run-Length Encoding (2)

- AAAAAAA would be encoded as: *A7
- *n5*x9ccc*h6 some other text *k8eee would be decoded into the following original text:

  nnnnnxxxxxxxxxccchhhhhh some other text kkkkkkkkeee
- The original text contains 51 characters, and the encoded string contains 35 characters. What is the compression rate and space saving for this example?
- Since we are using one character for the repetition count, it seems that we can't encode repetition lengths greater than nine. Instead of interpreting the count character as an ASCII digit, we could interpret it as a binary number.

Huffman Encoding (1)

- Why should the character “X”, which is seldom used in text, take up the same number of bits as the blank, which is used very frequently?
  - Huffman codes using variable-length bit strings to represent each character.
- A few characters may be represented by five bits, and another few by six bits, and yet another few by seven bits, and so forth.
- If we only use a few bits to represent characters that appear often and reserve longer bit strings for characters that don’t appear often, the overall size of the document being represented is small

Huffman Encoding (2)

- Consider the following Huffman codes:

Huffman code	Character
00	A
01	E
100	L
110	O
111	R
1010	B
1011	D

Huffman Encoding (3)

- DOORBELL would be encoded in binary as:
  - 1011 110 110 111 1010 01 100 100.
  - If we used a fixed-size bit string to represent each character (say, 8 bits), then the binary form of the original string would be 64 bits.
  - The Huffman encoding for that string is 25 bits long, giving a compression ratio of 64/25, or approximately 2.56:1.
- An important characteristic of any Huffman encoding is that no bit string, used to represent a character, is the prefix of any other bit string used to represent a character.

Huffman Encoding (4)

OÉ Gaillimh
NUI Galway

- Decode the following message using the code table:
  - 101011101001011

Huffman code	Character
00	A
01	E
100	L
110	O
111	R
1010	B
1011	D

References

- “The Architecture of Computer Hardware and Systems Software”, Irv Englander, ISBN: 0-471-36209-3
- “Computer Science Illuminated”, Nell Dale, John Lewis, ISBN: 0-7637-1760-6

- CT101 -
Computing Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Number Systems

Overview

- Know the different types of numbers
- Describe positional notation
- Convert numbers in other bases to base 10
- Convert base 10 numbers into numbers of other bases
- Describe the relationship between bases 2, 8, and 16
- Fractions
- Negative Numbers Representation
- Floating Point Numbers Representation

Number Systems

OÉ Gaillimh
NUI Galway

- Number categories
  - Many categories: natural, negative, rational, irrational and many others important to mathematics but irrelevant to the understanding of computing

- Number – unit belonging to an abstract mathematical system and subject to specified laws of succession, addition and multiplication
  - **Natural number** is the number 0 or any other number obtained by repeatedly adding 1 to this number.
  - A **negative number** is less than 0 and it is opposite in sign to a positive number.
  - An **integer** is any of the positive or negative natural numbers
  - A **rational number** is an integer or the quotient of any two integer numbers
    - ….is a value that can be expressed as a fraction

Number Systems

OÉ Gaillimh
NUI Galway

- The **base** of number system represents the number of digits that are used in the system. The digits always begin with 0 and continue through to one less than the base

- Examples:
  - There are two digits in base two (0 and 1)
  - There are eight digits in base 8 (0 through 7)
  - There are 10 digits in base 10 (0 through 9)

- The base also determines what the position of the digits mean

Positional Notation

- It is a system of expressing numbers in which the digits are arranged in succession and, the position of each digit has a place value and the number is equal to the sum of the products of each digit by its place value

- Example:
  - Consider the number 954:
    - $9 * 10^2 + 5 * 10^1 + 4 * 10^0 = 954$
  - Polynomial representation - formal way of representing numbers, where X is the base of the number:
    - $9 * X^2 + 5 * X^1 + 4 * X^0$
- Formal representation – consider that the base of representation is B and the number has n digits, where d_i represents the digit in the ith position.
  - $d_n * B^{n-1} + d_{n-1} * B^{n-2} + \ldots + d_2 B + d_1$
  - 642 is:
    $6_3 * 10^2 + 4_2 * 10 + 2_1 * 10^0$

Other bases

- What if 642 has the base of 13?

$$
\begin{aligned}
&+ 6 \times 13^2 &= \;& 6 \times 169 &= \;& 1014 \\
&+ 4 \times 13^1 &= \;& 4 \times 13 &= \;& 52 \\
&+ 2 \times 13^0 &= \;& 2 \times 1 &= \;& 2 \\
& & & &= \;& \mathbf{1068 \text{ in base } 10}
\end{aligned}
$$

- 642 in base 13 is equivalent to 1068 in base 10

Binary, Octal and Hexadecimal

- Decimal base has 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
- Binary is base 2 and has two digits (0 and 1)
- Octal is base 8 and has 8 digits (0, 1, 2, 3, 4, 5, 6, 7)
- Hexadecimal is base 16 and has 16 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)

Converting Octal to Decimal

- What is the decimal equivalent of octal number 642?

$$
\begin{aligned}
& 6 \times 8^2 = 6 \times 64 = 384 \\
+ & 4 \times 8^1 = 4 \times 8 = 32 \\
+ & 2 \times 8^0 = 2 \times 1 = 2 \\
& = 418 \text{ in base } 10
\end{aligned}
$$

- Remember that octal base has only 8 digits

  (0, 1, 2, 3, 4, 5, 6, 7)

Converting Hexadecimal do Decimal

- What is the decimal equivalent of the hexadecimal number DEF?

$$
\begin{aligned}
&\mathbf{D \times 16^2 = 13 \times 256 = 3328} \\
&\mathbf{+\ E \times 16^1 = 14 \times 16 = 224} \\
&\mathbf{+\ F \times 16^0 = 15 \times 1 = 15} \\
&\mathbf{= 3567 \text{ in base } 10}
\end{aligned}
$$

- Remember that hexadecimal base has 16 digits

  (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)

Converting Binary to Decimal

- What is the equivalent decimal of the binary 10110 number?

$$
\begin{aligned}
& 1 \times 2^4 = 1 \times 16 = 16 \\
+ & 0 \times 2^3 = 0 \times 8 = 0 \\
+ & 1 \times 2^2 = 1 \times 4 = 4 \\
+ & 1 \times 2^1 = 1 \times 2 = 2 \\
+ & 0 \times 2^0 = 0 \times 1 = 0 \\
& = 22 \text{ in base } 10
\end{aligned}
$$

- Remember that binary base has only 2 digits (0, 1)

Arithmetic in Binary

- The rules of arithmetic are analogous in other bases as in decimal base

Should read 1+1=0 with a **carry** of 1 similar to base 10 where 9 + 1 = 0 with a carry of 1 = 10

Addition			
1	1	0	0
+	+	+	+
1	0	1	0
10	1	1	0

-1 can be stated as 1 with a **borrow** of 1. Leading 1 we consider to be the sign, so 11 means -1

Subtraction			
1	1	0	0
-	-	-	-
1	0	1	0
0	1	11	0

Addition in Binary

Case	A	+	B	Sum	Carry
1	0	+	0	0	0
2	0	+	1	1	0
3	1	+	0	1	0
4	1	+	1	0	1

In the fourth case, a binary addition is creating a sum of (1 + 1 = 10). That is, the 0 is written in the given column with a carry of 1 over to the next column to the left.

Addition in Binary

OÉ Gaillimh
NUI Galway

- Base 2: 1+1 operation - the rightmost digit reverts to 0 and there is a carry into the next position to the left

- We can check if the answer is correct by converting the both operands in base 10, adding them and comparing the result

$$
\begin{array}{r}
\color{red}{1\ 1\ 1}\ \ \ \\
0\ 1\ 0\ 1 \\
+\ 1\ 0\ 1\ 1 \\
\hline
1\ 0\ 0\ 0\ 0
\end{array}
$$

Carry Values

Subtraction in Binary

OÉ Gaillimh
NUI Galway

Case	A - B	Subtract	Borrow
1	0 - 0	0	0
2	1 - 0	1	0
3	1 - 1	0	0
4	0 - 1	0	1

- In the fourth case, when we are subtracting 1 from 0 we need to "borrow" 1.

Subtracting in Binary

- The rules of the decimal base applies to binary as well. To be able to calculate 0-1, we have to “borrow one” from the next left digit.
- More precisely, we have to borrow **one power of the base** (2)

```
   1 2
 0 2 0 2
 1 0 1 0
-0 1 1 1
--------
 0 0 1 1
```

- You can check if the result is correct by converting the operands in decimal and making the calculus.

Review Question 4

- Add 4 bit number 0100 with 0111. The answer is:

A. 1001

B. 1011

C. 1110

D. 1111

Review Question 5

- Subtract 4-bit number 0100 from 1111. The answer is:

A. 1001
B. 1011
C. 1110
D. 0100

- CT101 -
Computing Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Power of Two Number Systems

- Binary and octal numbers have a very special relationship between them: given a binary number, can be read in octal and given an octal number can be read in binary (i.e. have 753 in octal, in binary you have 111 101 011 by replacing each digit by its binary representation)
- Table represents counting in binary with octal and decimal representation

Binary	Octal	Decimal
000	0	0
001	1	1
010	2	2
011	3	3
100	4	4
101	5	5
110	6	6
111	7	7
1000	10	8
1001	11	9
1010	12	10

Converting Binary to Octal

- Start at the rightmost binary digit and mark the digits in groups of three
- Convert each group individually

10101011	**<u>10</u>**	**<u>101</u>**	**<u>011</u>**
	2	**5**	**3**

10101011 is 253 in base 8

- The reason that binary can be immediately converted in octal and vice-versa is because 8 is power of 2
- There is a similar relationship between binary and hexadecimal

Converting Binary to Hexadecimal

OÉ Gaillimh
NUI Galway

- Start at the rightmost binary digit and mark the digits in groups of four

- Convert each group individually

10101011   **1010** **1011**

A   B

10101011 is AB in base 16

Converting Decimal to Other Bases

OÉ Gaillimh
NUI Galway

- Involves dividing by the **base** into which you convert the number
- Algorithm:
  - Dividing the number by the base you get a quotient and a remainder
  - While the quotient is ***not*** zero:
    - Divide the decimal number by the <u>new base</u>
    - Make the remainder the next digit to the left in the answer
    - Replace the original dividend with the quotient
- The base 10 number 680 is what number in base 16?

Converting Decimal to Hexadecimal

- 680 in decimal
- 16 ^ 2 = 256, 16^1 = 16, 16^0 = 1
- 680 divided by 256 = 2 (**remainder** is 680 – 512 = 168)
  - First digit of hexadecimal number is 2
- Divide remainder by 16 = 10 (**new remainder** is 168-160=8)
  - Second digit of hexadecimal is A
- Third digit of hexadecimal is the new remainder: 8
- Therefore the hexadecimal number is: 2A8
- Divide initial number by 256, remainder by 16 and the final remainder is the final hexadecimal digit

Converting Decimal to Hexadecimal

256	16	1
680 / 256	168 / 16	8 / 1
Quotient: 2	Quotient: 10	Quotient: 8
Remainder: 168	Remainder: 8	Remainder: 0
2	A	8

Review of Binary Values in Computing Systems

- Modern computers are *binary* machines
- A digit in binary system is either 0 or 1
- The binary values in a computer are encoded using voltage levels:
  - 0 is represented by a 0V signal (or low voltage)
  - 1 is represented by a 5V signal (i.e. in TTL logic), or by a high voltage signal.
- Bit – is a short expression for binary digit
- Byte – eight binary digits
- Word – a group of one or more bytes; the number of bits in a word is the word length in a computer

Fractions

- Representation and conversion of fractional numbers is more difficult because there is not necessarily an exact relationship between fractional numbers in different number bases.
- Fractional numbers that can be represented exactly in one number base, may be impossible to represent exactly in another
- Example:
  - The decimal fraction 1/3 is not representable as a decimal value in base 10: 0.3333333_{10}…; this can be represented exactly in base 3 as 0.1_3
  - The decimal fraction 1/10 (or 0.1_{10}) cannot be represented exactly in binary form. The binary equivalent begins: 0.000110011001_2…

Fractions

- The strength of each digit is B times the strength of its right neighbour (where B is the base for a given number).
- If we move the number point to the right, the value of the number will be multiplied by the base:
  - 1390_{10} is 10 times as large as 139.0_{10}
  - Then 100_2 is twice as big as 10_2
- The opposite is also true – if we move the number point to the left one place, then the value is divided by the base
- A given number $.D_1D_2D_3 \ldots D_n$ will be represented as:
  - $D_1 * B^{-1} + D_2 * B^{-2} + D_3 * B^{-3} + .. + D_nB^{-n}$
  - 0.2589 = 2 * (1/10) + 5 * (1/100) + 8 * (1/1000) + 9 * (1/10000)
  - 0.101011_2 = (½) + (1/8) + (1/32) + (1/64)

Fractional Conversion Methods

- The intuitive method:
  - Determine the appropriate weights for each digit, multiply each digit by its weight and then add the values
    - Example:
    - Convert 0.12201_3 to base 10 = (1/3) + 2 * (1/9) + 2 * (1/27) + (1/243) = 0.63374
  - Convert the number into a natural number (and record what was the multiplier) and then divide the result by the multiplier
    - Example:
    - convert 0.110011_2 to base 10 – shifting the binary point six places to the right and converting, we have: 32 + 16 +2 +1 = 51; shifting the point back is the equivalent of 2^6 or 64, so we can obtain the final number by dividing 51 by 64 = 0.796875
- Variation of the division method shown earlier: we multiply the fraction by the base value, repeatedly, and record, then drop the values that move to the left of the point.
  - This is repeated until the level of accuracy is obtained or until the value being multiplied is zero

Fractions Base Conversion

The 1 is saved as result then dropped and the process repeated

```
0.828125 *
       2
--------------
1.656250 *
       2
--------------
1.312500 *
       2
--------------
0.625000 *
       2
--------------
1.250000 *
       2
--------------
0.500000 *
       2
--------------
1.000000
```

0.110101

Fraction Conversions between Bases of Power of Two

- The conversion between bases where one base is an integer power of the other can be performed for fractions by grouping the digits in the smaller base as before
- For fractions, the grouping must be done from the **left to right**; the method is otherwise identical
- Example:
  - Convert 0.10111_2 to base 8: **0.101_110 = $\mathbf{0.56_8}$**
  - Convert 0.1110101 to base 16: **0.1110_1010 = $\mathbf{0.EA_{16}}$**

- CT101 -
Computing Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Representing Negative Numbers

- Are negative numbers just numbers with a minus sign in the front? This is probably true…but there are issues to represent negative numbers in computing systems

- Common schemas:
  - Sign-magnitude
  - Complementary representations:
    - 1’s complement
    - ***2’s complement – most common & important***

Sign Magnitude

- Left most bit used to represent sign
  - 0 = positive value
  - 1 = negative value
  - behaves like a “flag”
- It is important to decide how many bits we will use to represent the number
- Example: Representing +5 and -5 on 8 bits:
  - +5: 00000101
  - -5: 10000101
- So the very first step we have to decide on the **number of bits to represent the number**

Difficulties with Sign Magnitude

- Two representations of zero
  - Using 8-bit sign-magnitude…
    - `0: 00000000`
    - `0: 10000000`
- Arithmetic is awkward!
  - 8-bit sign-magnitude:
    - 00000001 + 00000010 = 00000011
    - 00000010 + 10000001 = 00000001
    - It requires a different algorithm, can't just add and carry, meaning more complexity in hardware in order to implement an ALU

Complementary Representations

OÉ Gaillimh
NUI Galway

- 9’s (Decimal) complement
- 1’s (Binary) complement

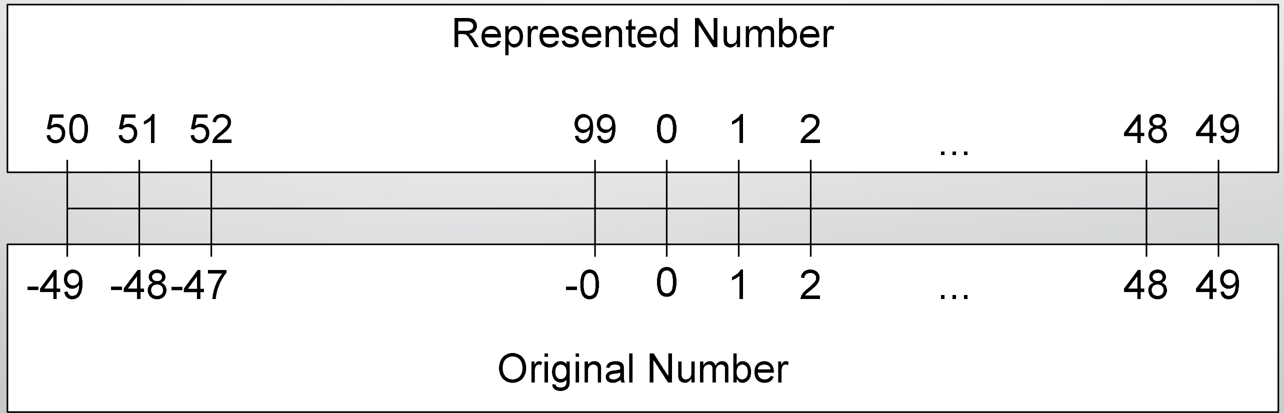
- 10’s (Decimal) complement
- 2’s (Binary) complement

9^{th} Decimal Complement

- Decide on the number of digits (word length) to represent numbers
- Then represent the negative numbers by the largest number minus the absolute value of the negative number.
- Example:
  - 2-digit 9's complement of –12
    - 99 – 12 = 87
      - To get back the **abs value**, invert again; i.e. 99 – 87 = 12
- Most negative number:
  - representation 50 ……… 99 | 0 ……… 49
  - original number -49………...-0 | 0 ……… 49

9th Decimal Complement Problems

- Two representations of zero
  - Using 2-digit 9's complement…
    - `0: 00`
    - `0: 99`
- Arithmetic is still a little awkward, meaning that complex hardware will be required to build arithmetic units for this logic.



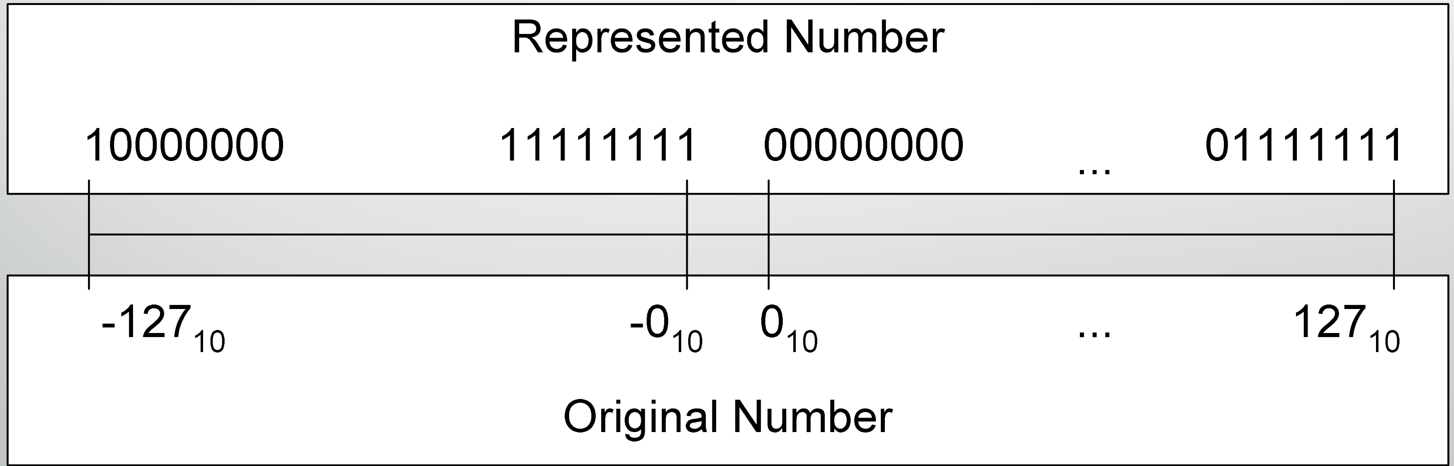
9's Decimal Complement for two digit numbers

1's Binary Complement

- Decide on the **number of bits** (word length) to represent numbers
- Then represent the negative numbers by the largest number minus the absolute value of the negative number.
- Example:
  - 8-digit 1's complement of –101
    - 11111111 – 00000101 = 11111010 (= $(2^8 - 1) - 5$ in base 10)
    - Notice: very easy to flip or "invert" the 1's and 0's to compute 1's complement of a number
    - To get back the **abs value**, invert again
- Most negative:
  - representation 10000000 …11111111 | 0… 01111111
  - original number -01111111 …-00000000 | 0… 01111111

Difficulties with 1's Complement

- Two representations of zero
  - Using 6-digit 1's complement…
    - `0: 000000`
    - `0: 111111`
- Arithmetic is still a little awkward!



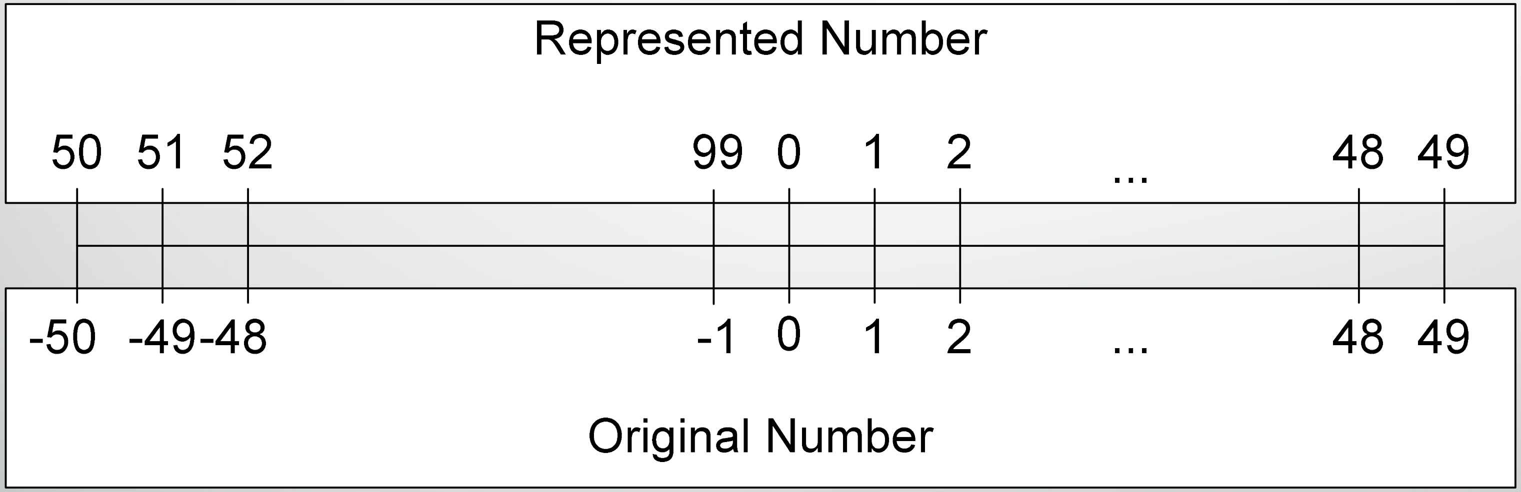
1's binary complement for 8 digit numbers

10’s Complement

- Again decide on the number of bits (word length) to represent numbers
- Then represent the negative numbers by the [largest number+1] minus the absolute value of the negative number
- Example
  - 2-digit 10’s complement of –12
    - 100 – 012 = 88
    - To get back the **abs value**, invert again; i.e. 100 – 88 = 12
- Most negative number:
  - representation   50 ……… 99 | 0 ……… 49
  - original number -50………..-1 | 0 ……… 49

10’s Complement

- Notice: unique representation of 0
- 10’s complement = 9’s complement +1



10's Complement for two digit numbers

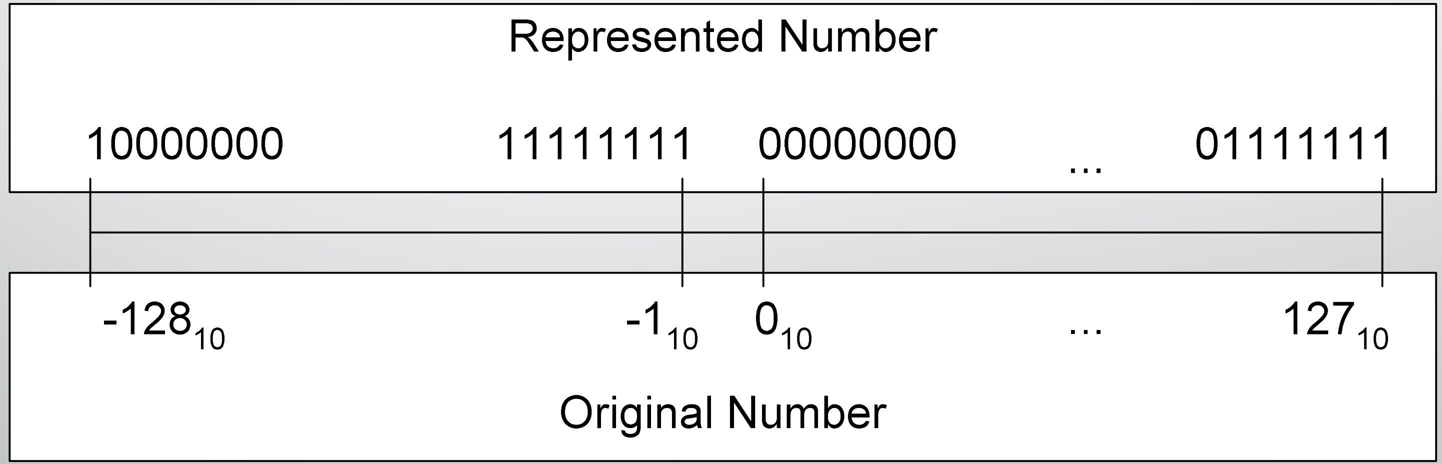
2’s Complement

OÉ Gaillimh
NUI Galway

- It is similar to 10’s complement representation for decimal.
- Decide on the number of digits (word length) to represent numbers
- Then represent the negative numbers by the [largest number + 1] minus the absolute value of the negative number.
- Example:
  - 8-digit 2’s complement of -5
    - 100000000 – 00000101 = 11111011(= 2^8 –5 in base 10)
    - To get back the **abs value**, subtract again from 2^8

2’s Complement

- The 2’s complement of a number can be found in two ways
  - Subtract the value from the modulus [largest number +1]
  - Find 1’s complement (by inverting the value) and adding 1 to the result (2’complement = 1’s complement +1)



2's complement for 8 digit numbers

1’s Complement versus 2’s Complement

OÉ Gaillimh
NUI Galway

- Both methods are used in computer design
- 1’s complement
  - Offers a simpler method to change the sign of a number
  - Algorithm must test for and convert -0 to 0 at the and of each operation.
- 2’s complement
  - Simplifies the addition operation
  - Additional add operation required every time a sign change is required (by inverting and adding 1)

Binary Complements Tips and Tricks

OÉ Gaillimh NUI Galway

- Positive numbers are always represented by themselves
- Small negative numbers (close to 0) have representations that start with large numbers of 1's. The number -2 in 8 bit 2's complement is represented as 11111110
- Since there is only a difference in value of 1 between 1's and 2's complement representations of negative numbers (of course the positive representations are always the same), you could get a quick idea of the value (in either of the representations) by inverting all the 1's and 0's and *approximating* the value from the result

OÉ Gaillimh
NUI Galway

- CT101 -
Computing Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Overflow and Carry Conditions

- Overflow occurs when the result of the calculation doesn’t fit into the fixed number of bits available for the result.
  - In 2’s complement, an addition or subtraction overflow occurs whenever the result overflows into the *sign bit* – if the sign of the result is different than the sign of the both operands
- In addition, computing systems provide for a **carry flag** that is used to correct for carries and borrows that occurs when large number have to be separated into parts to perform additions and subtractions.

Overflow and Carry Conditions

- Example:
  - CPU has only 32-bit wide instructions, but has to add 64-bit numbers
  - The 64-bit number are divided in two 32 bits parts, the least significant 32-bit parts are added with carry, and the most significant parts are added using also as input any carry that was generated from the previous addition operation.

- Carry and overflow are occurring independently of each other:
  - **Overflow** occurs when you cannot properly represent the result as a _*signed*_ value (you overflowed into the sign bit).
  - **Carry** occurs when you cannot properly represent the result as an _*unsigned*_ value (no sign bit required).

Overflow and Carry Conditions for **4-bit** numbers

(+4) + (+2)
0100
0010
0110 = (+6)
no overflow
no carry
the result is correct

(+4) + (+6)
0100
0110
1010 = (-6)
overflow
no carry
the result is incorrect

(-4) + (-2)
1100
1110
11010 = (-6)
no overflow
carry
Ignoring the carry, the result is correct

(-4) + (-6)
1100
1010
10110 = (+6)
overflow
carry
Ignoring the carry, the result is incorrect

More Overflow and Carry (on 8-bit words)

	Binary	Hex	Unsigned	Signed	
(1)	1010 1000	xA8	168	-88	
	0010 1101	x2D	45	45	
	============	====	===	===	
	1101 0101	xD5	213	-43	**C = 0 V = 0**
(2)	1101 0011	xD3	211	-45	
	1111 0100	xF4	244	-12	
	============	====	===	===	
	11100 0111	x1C7	455	-57	**C = 1 V = 0**
(3)	0010 1101	x2D	45	45	
	0101 1000	x58	88	88	
	============	====	===	===	
	1000 0101	x85	133	-123	**C = 0 V = 1**
(4)	1101 0011	xD3	211	-45	
	1010 1000	xA8	168	-88	
	============	====	===	===	
	10111 1011	x17B	379	-133	**C = 1 V = 1**

Note:
Producing a carry, C = 1, indicates **unsigned** overflow, it does **not** indicate **signed** overflow.
To recognize signed overflow, two conditions must be present:
The operands must have the **same sign**, and
the **sum** must have the **opposite sign**.

C= Carry V= Overflow

Floating Point Numbers

OÉ Gaillimh
NUI Galway

- Real or floating-point number are used in computing systems when the number to be expressed is outside of the integer range or when the number contains a decimal fraction
- The number is represented by a fixed number of digits of precision together with a power that shifts the point to make the number larger or smaller
- We need to understand the properties of floating-point numbers, how they are represented and how calculations are performed
- First, as usual, we will present the techniques in base 10, since working with decimal numbers is more familiar. Then we will extend the discussion to binary numbers.

Review of Exponential Notation

- Consider the number 12345. Here are a number of alternative representations:
  - $12345 * 10^{0}$
  - $0.12345 * 10^{5}$
  - $123450000 * 10^{-4}$
  - $0.0012345 * 10^{7}$ (OR $0.00123 * 10^{7}$ if we have limited digits of magnitude)
- The way of representing the above number is known as exponential notation (scientific notation)

Exponential Notation

- Four components are required to define a number using this notation:
  - The sign of the number (+ in our example)
  - The magnitude of the number (known as mantissa, 12345 in our example)
  - The sign of the exponent (+ in our example)
  - The magnitude of the exponent (say it is 3)

Exponential Notation

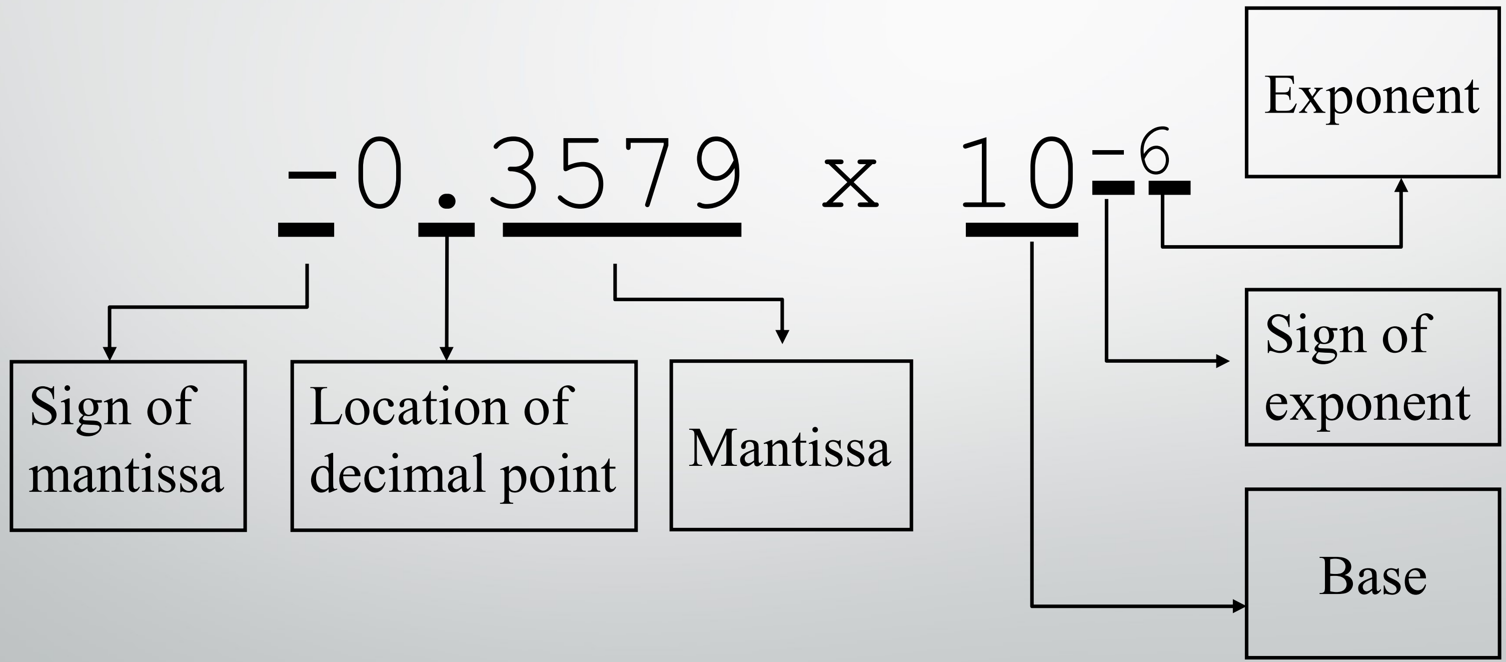
OÉ Gaillimh
NUI Galway

- Two additional pieces of information are required to complete the representation:
  - The base of the exponent (in this case 10) – in the computer is usually specified to be 2.
  - The location of the decimal (or binary point if we are working in base 2) point – in the computer the binary point is set at a particular location in the number, most common at the beginning or the end of the number. Since its location never changes, it is not necessary to actually store the point. Knowing the location of the point is essential

- In our example, the location of the decimal point was not specified, so reading the data suggests that the number might be $+12345 * 10^3$, which is wrong. The actual placement of the decimal point should be $12.345 * 10^3$

Example

- The number to be represented is -0.0000003579
- One possible representation of this number is:



Floating Point Format

OÉ Gaillimh
NUI Galway

$$-0.35790 \times 10^{-6}$$

- Typical representation is using 8 digits: SEEMMMMM, where:
  - S – one digit for the sign of the mantissa
  - EE – two digits for the exponent
  - MMMMM- the mantissa representation

Floating Point Format

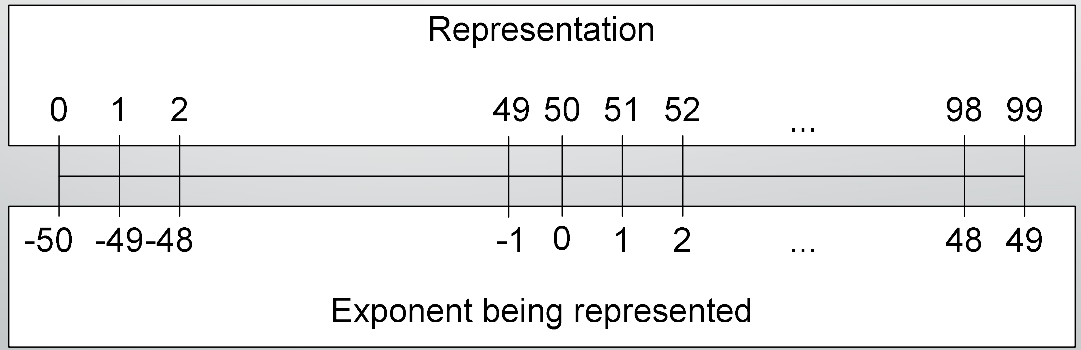
- There is no provision for the sign of the exponent. Use some method that includes it:

- One method, is to use complementary representation for the exponent
- Another method is to use an **offset** representation: if we pick a value somewhere in the middle of the possible values of the exponent (0-99), say 50 and declare that this value corresponds to 0, then every value lower than that will be negative and those above will be positive.

Floating Point Format – Excess N Representation

- This method is known as Excess-N notation, where N is the chosen mid-value
- It is simpler to use for exponents than the complementary form and appropriate for the calculations required on exponents
- Allows to store an exponential range of -50 to 49
- If we assume that the decimal point is located at the beginning of the five-digit mantissa, excess-50 notation allows us magnitude range of

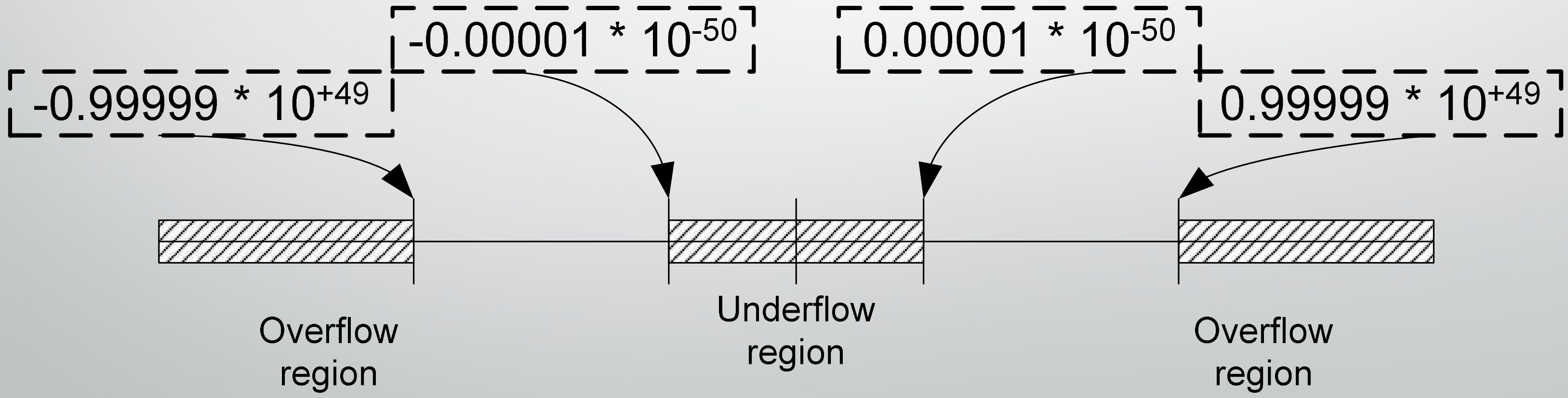
$$0.00001 * 10^{-50} < \text{number} < 0.99999 * 10^{+49}$$



Scale for the offset technique

Floating Point Exceptions

- **Overflow** – using/resulting in a number of magnitude too large to be stored
- **Underflow** - where the number is a decimal fraction with magnitude too small to be stored



Examples: SEEMMMMM

- The exponent is represented in excess of 50.
- The computer is aware of storing only numbers, no signs nor position of the decimal point.
  - Decimal point is at the beginning of the mantissa
  - Sign is represented as: 0 a positive sign, 5 represents a negative sign (arbitrary representation in base 10)

    - 05324657 = 0.24657 * 10^{3} = 246.57
    - 54810000 = -0.10000 * 10^{-2} = - 0.001
    - 55555555 = -0.55555 * 10^{5} = 55555
    - 04925000 = 0.25000 * 10^{-1} = 0.025

Normalization and Formatting

- The number of digits used will be determined by the desired precision
- To maximize precision for a given number of digits, numbers will be stored with no leading zeros.
- **Normalization –** when necessary, numbers are shifted left by increasing the exponent until leading zeros are eliminated
- Example **–** our format will consist of a sign, five digits with the decimal point located at the beginning of the number and two exponent digits:

$$.\mathbf{MMMMM} * \mathbf{10}^{\mathbf{EE}}$$

Normalization and Formatting (SEEMMMMM)

- Normalization (**246.8035**)
  1. Provide an exponent of 0 for the number if an exponent wasn't already specified (**246.8035 * 10^0**)
  2. Shift the decimal point left or right by increasing or decreasing the exponent, until the decimal point is in the proper position (**0.2468035 * 10^3**)
  3. Shift the decimal point right, if necessary, until there are no leading zeros in the mantissa (no adjustment required)
  4. Correct the precision by adding or discarding digits as necessary, to meet the specification (**0.24680 * 10^3**)
- Formatting:

  Put it into a standard exponential form, by converting the exponent into 50-excess notation and place the digits into their correct locations in the word (**0 53 24680**)

positive

Exponent is 3

- **Overflow** occurs when you cannot properly represent the result as a signed value
  - (Meaning you overflowed into the sign bit).
- **Carry** occurs when you cannot properly represent the result as an unsigned value
  - (Meaning no sign bit is required).

The following table of binary representations is important to understand these examples:

(**Remember**: we are flipping all the bits and adding one to get the negative binary representation)

Positive Decimal Number	Binary Representation	Binary Representation	Negative Decimal Number
0	0000		
1	0001	1111	-1
2	0010	1110	-2
3	0011	1101	-3
4	0100	1100	-4
5	0101	1011	-5
6	0110	1010	-6
7	0111	1001	-7
		1000	-8

In the following example, we have the addition of 0100 (+4) and 0010 (+2). The result is 0110 (+6) which is correct and does not involve an overflow nor a carry.

(+4) + (+2)
0100
0010
0110 = (+6)
no overflow
no carry
the result is correct

In the following example, we have the addition of 0100 (+4) and 0110 (+6) which results in an overflow (into the sign bit). The is no carry in this case as there is no extra carried number brought to the left if we treat the calculation as unsigned numbers.

```
(+4) + (+6)
0100
0110
-------------
1010 = (-6)
overflow
no carry
the result is incorrect
```

In the following example, we have the addition of 1100 (-4, see table above) and 1110 (-2). There is no overflow because the signed value (-6) is given as the result. There is a carry in this case as there is a one “carried over” to the left after the last addition. If we ignore this carried digit, and stick with our 4-bit number then the result is correct.

```
(-4) + (-2)
  1100
  1110
-------------
11010 = (-6)
no overflow
carry
Ignoring the carry, the result is
correct
```

In the following example, we have the addition of 1100 (-4) and 1010 (-6). There is overflow because the addition directly affects the sign bit. Again like the previous example, we have a carry from the calculation but this time, if we ignore it, the result will still be incorrect.

```
(-4) + (-6)
  1100
  1010
-------------
10110 = (+6)
overflow
carry
Ignoring the carry, the result is
incorrect
```

- CT101 -
Computing Systems

Dr. Fatemeh Ahmadi Zeleti

Fatemeh.ahmadizeleti@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Number Systems

Week 7

Part 9 and Part 10

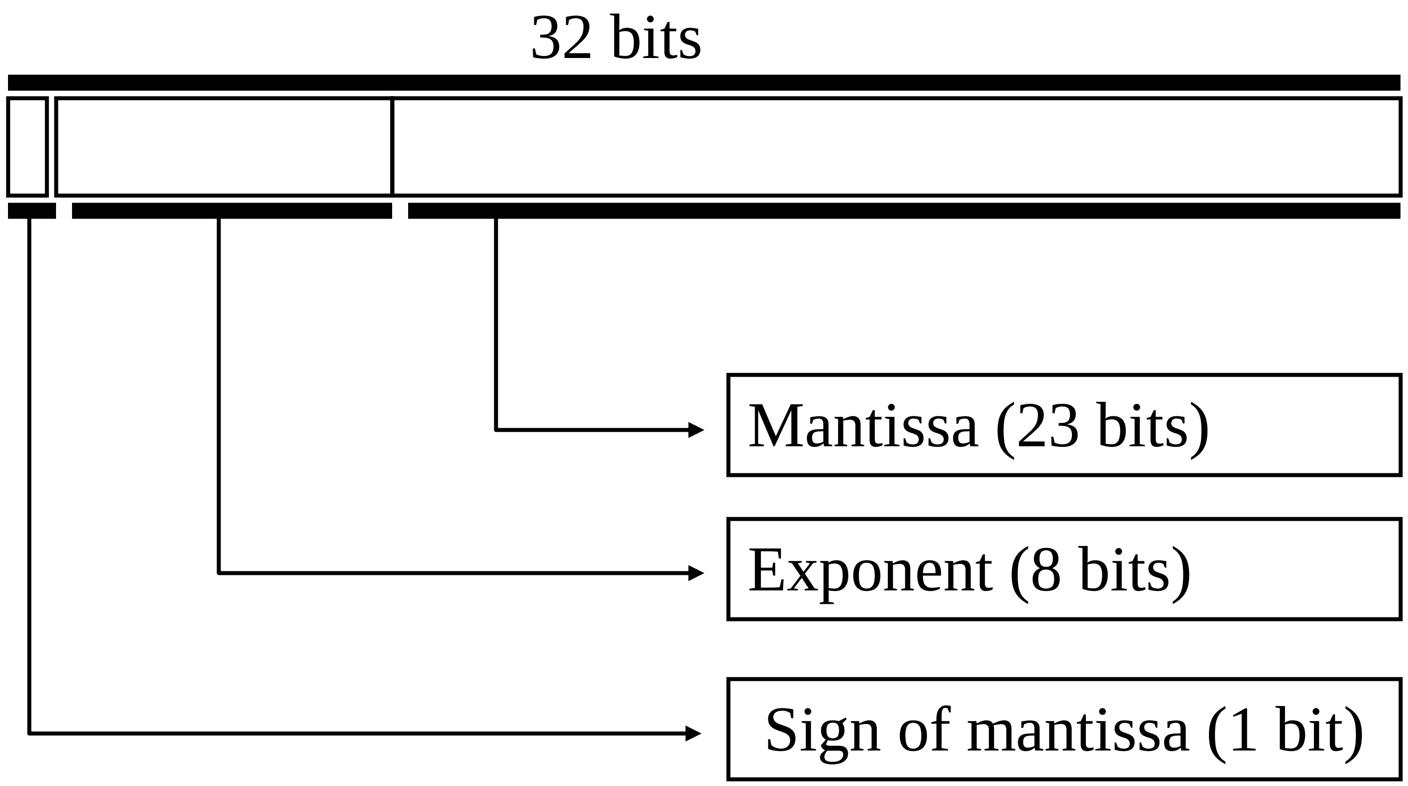
Floating Point in Computing Systems

- The leading bit of the mantissa must be 1 if the number is normalized
- The leading bit can be treated implicitly (similar with the binary point)
- Disadvantages:
  - Leading bit is always 1, means that we can't represent too small numbers, limiting the small end of the range
  - Any format that might require a 0 in the leading bit can't be represented
  - This method requires that we provide a separate way to store the number 0.0, since the requirement to have the leading bit 1, makes the mantissa 0.0 an impossibility
- The additional bit doubles the available precision of the mantissa, the slightly narrowed range is usually an acceptable trade off. The number 0.0 is represented by selecting a particular 32-bit word and assigning it the value 0.0.

IEEE 754 Standard

- Most common standard for representing floating point numbers
- Single precision: 32 bits, consisting of...
  - Sign bit (1 bit)
  - Exponent (8 bits)
  - Mantissa (23 bits)
- Double precision: 64 bits, consisting of…
  - Sign bit (1 bit)
  - Exponent (11 bits)
  - Mantissa (52 bits)

Single Precision Format



Single Precision Format

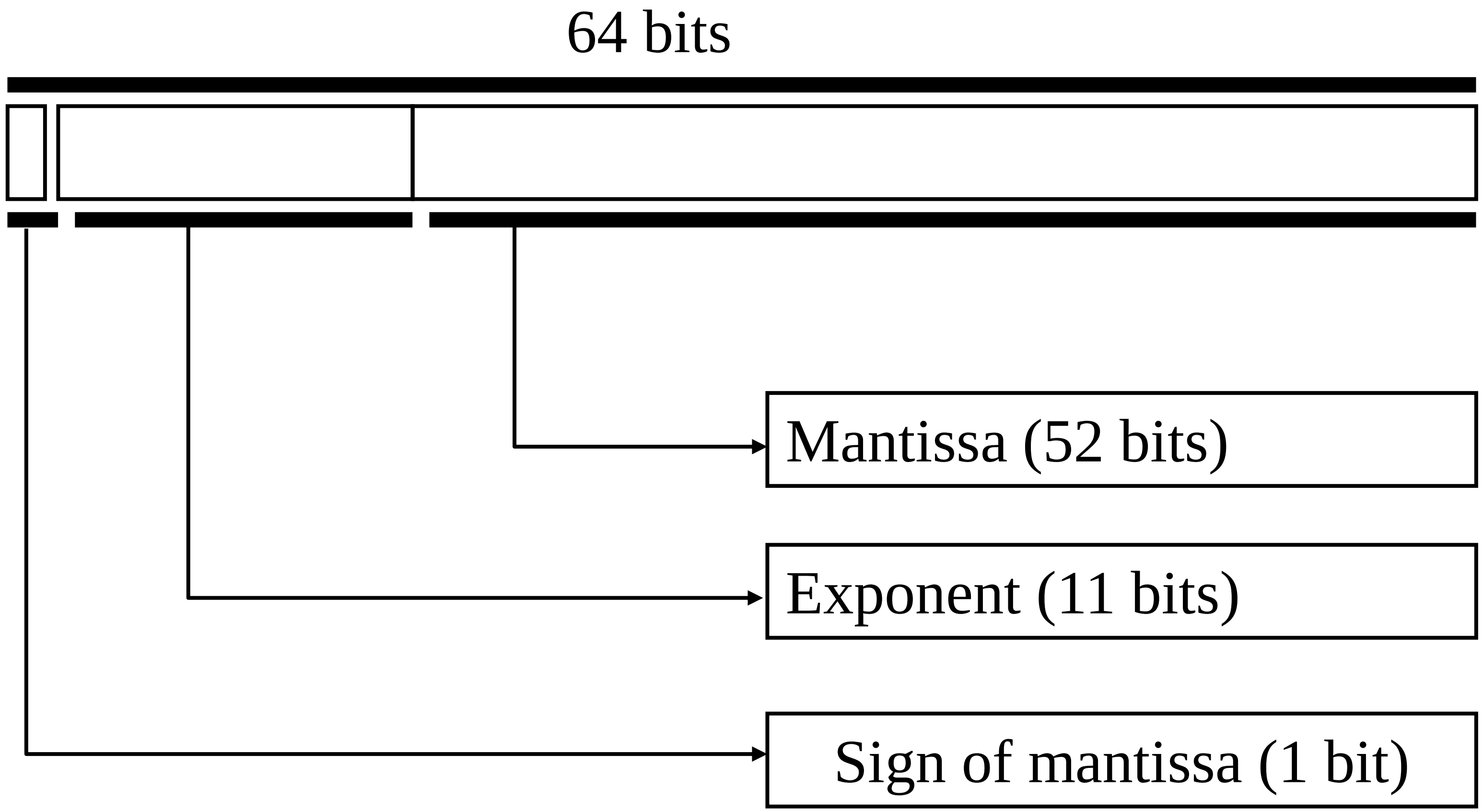
- The mantissa is *normalized*
- Has an implied "1" on left of the point. Normalized form of the mantissa is 1.MMMMM…
- Example:
  - Mantissa: 10100000000000000000000
  - Representation: $1.101_2 = 1.625_{10}$

Note: convert each side of the point using techniques described in previous lectures

Single Precision Format

- The exponent is formatted using excess-127 notation, with an implied base of 2
  - Example:
    - Exponent: **10000111**
    - Representation: **135 – 127 = 8**
- The stored values 0 and 255 of the exponent are used to indicate special values, the exponential range is restricted to 2^{-126} to 2^{127}
- The number 0.0 is defined by a mantissa of 0 together with the special exponential value 0
- The standard allows also values +/-∞ (represented as mantissa +/-0 and exponent 255)
- Allows various other special conditions

Double Precision Floating Point



Double Precision Floating Point

- Same format as single precision floating point representation
- Excess-1023 exponent representation
- An implied base of 2 and an implied most significant bit at the left of an implied binary point
- Range of more than 10^{-300} to 10^{300}

Conversion between base 10 and base 2

- The whole and fractional parts of numbers with an embedded decimal or binary point must be converted separately

- Numbers in exponential form must be reduced to a pure decimal or binary mixed number or fraction before the conversion can be performed

Examples

Decimal value of 32-bit floating-point number:

1 10000010 11110110000000000000000

Mantissa: $1.1111011_2 = 1.9609375_{10}$

Mantissa conversion: 1. First the whole number: $1_2 = 1_{10}$ 2. Then the fractional number: 0.1111011_2 = ½ + ¼ + 1/8 + 1/16 + 1/64 + 1/128 = (64 + 32 + 16 + 8 + 2 + 1)/128 = 0.9609375

Exponent: $10000010_2 = 130_{10}$ (because is excess-127) = 3

Sign: 1 (negative number)

Answer: $- 1.9609375 * 2^3 = -15.6875$

Examples

- Express 3.14 as 32-bit floating-point number
- Note: use 10 significant bits for the mantissa
- Normalize the number
  - Convert the whole and fractional parts independently
    - $3_{10} = 11_2$
    - $0.14_{10} = 0.00100011110000000000000_2$, this is obtained using the multiplication method presented in one of the previous lecturers (see the next slide)
  - 11.0010001111 = 1.10010001111000000000000 * 2
  - The exponent is 1, represented in excess-127 is: 10000000
  - the mantissa is 10010001111000000000000
  - The sign is positive (0)
- Answer: 0 10000000 10010001111000000000000

Reminder for fraction decimal to binary conversion

0.0010001111

0.14 * 2
0.28 * 2
0.56 * 2
1.12 * 2
0.24 * 2
0.48 * 2
0.96 * 2
1.92 * 2
1.84 * 2
1.68 * 2
1.36 * 2 …

Conversion – Class Exercise

- Convert 45.45 to IEEE 754 single precision format

= 0100 0010 0011 0101 $\mathbf{\overline{1100}}$

Arithmetic with Floating Point Numbers

- On the computer, it is more difficult than integers
- Addition & Subtraction:
  - ..need to “line up” the exponents (by making the **smaller** one match the **larger** one, moving the point in the mantissa) and perform addition on the mantissa
- Multiplication & Division
  - ..need to do separate operations to mantissa and exponent: multiply/divide mantissa and correspondingly add/subtract and adjust the exponent

Addition Example

- Perform the addition $3_{10} + 1.5_{10}$
- Convert the numbers in floating point representation
  - First Operand N1 = $3_{10} = 11_2$
    - N1 = 11.000000… = $1.100000000000000000000000000 * 2^1$
    - The exponent is E1 = 1, represented in excess-127 is: 1000 0000
    - The mantissa is M1 = 1.100 0000 0000 0000 0000 0000
    - The sign is positive (0)
  - Second Operand N2 = $1.5_{10} = 1.1_2$, this is obtained using the multiplication method presented in one of the previous lecturers
    - N2 = 1.100000… = $1.100000000000000000000000000 * 2^0$
    - The exponent is E2 = 0, represented in excess-127 is: 0111 1111
    - The mantissa is M2 = 1.100 0000 0000 0000 0000 0000
    - The sign is positive (0)

Addition Example

- In order to perform the addition, we need to “line up” the exponents (by *making the smaller one match the larger one*, moving the point in the mantissa) and perform addition on the mantissa
- E1 is the largest exponent, so we will make the modifications on the second number:
  - E2’ = 1 (1000 0000)
  - M2’ = 0.110 0000 0000 0000 0000 0000
- Perform the addition on the mantissas:
  - M1 + M2’ = 10.010 0000 …

$$\begin{array}{r} 1.10000..+ \\ 0.11000.. \\ \hline 10.01000.. \end{array}$$

- Remember that the common exponent is 1000 0000
- We need to normalize again the result, so the mantissa of the resulting number is M = 1.001, and the exponent of the result is E = 1000 0001
- The answer is 0 1000 0001 001 0000 0000 0000 0000 0000
  - $1.001 * 2^2{}_2 = 4.5_{10}$

Addition – Class Exercise

- Add X and Y

X = 0100 0010 0000 1111 0000 0000 …

Y = 0100 0001 1010 0100 000 0000…

0100 0010 0110 0001 0000 0000…

- CT101 -
Computing Systems

Dr. Fatemeh Ahmadi Zeleti

Fatemeh.ahmadizeleti@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Number Systems

Multiplication of Floating-Point Numbers

Multiplication Example

- Perform the multiplication 3_{10} * 1.5_{10}
- Convert the numbers in floating point representation
  - First Operand N1 = 3_{10} = 11_2
    - N1 = 11.000000… = 1.10000000000000000000000 * 2^1
    - The exponent is E1 = 1, represented in excess-127 is: 1000 0000
    - The mantissa is M1 = 1.100 0000 0000 0000 0000 0000
    - The sign is positive (0)
  - Second Operand N2 = 1.5_{10} = 1.1_2 , this is obtained using the multiplication method presented in one of the previous lecturers
    - N2 = 1.100000… = 1.10000000000000000000000 * 2^0
    - The exponent is E2 = 0, represented in excess-127 is: 0111 1111
    - The mantissa is M2 = 1.100 0000 0000 0000 0000 0000
    - The sign is positive (0)

Multiplication Example

- We need to do separate operations to mantissa and exponent:
  - multiply/divide mantissa
    - M1 * M2 = 1.1 * 1.1 = 10.01

$$
\begin{array}{r}
1.1\ * \\
1.1 \\
\hline
1\ 1 \\
1\ 1\ \ \\
\hline
10.0\ 1
\end{array}
$$

- Correspondingly add/subtract and adjust the exponent
  - E1 + E2 -127 = 1000 0000 + 0111 1111 = 1111 1111 – 127 = 1000 0000
  - Normalize the number:
    - M = 1.001
    - E = 1000 0001
    - Resulting number: 0 1000 0001 001 0000 0000 0000 0000 0000
    - The resulting number is 4.5_{10} representation

OÉ Gaillimh
NUI Galway

- CT101 -
Computing Systems

Dr. Fatemeh Ahmadi Zeleti

Fatemeh.ahmadizeleti@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Computer Systems Organization

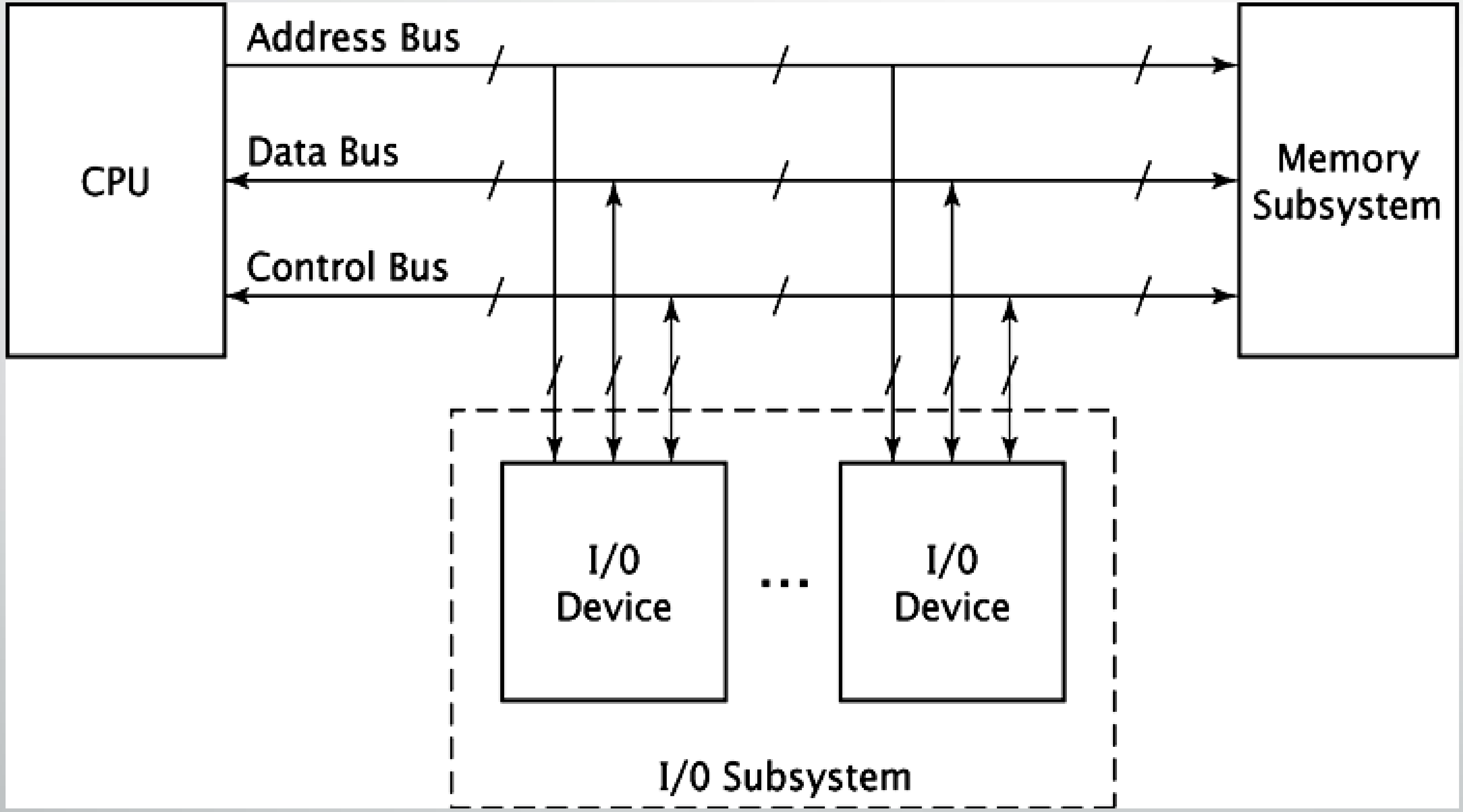
Contents

- Describe the operation of a computer at the functional level
- Explain the function of the main components of a computer system
- Detail the interaction of computer sub-systems
- Detail the organization of computer sub-systems

Computer Organization

- Instruction set architecture (ISA) provides a good understanding of what a microprocessor can do; provides no information on how to use the processor in a bigger system.
- In order to design a computing system, **more** information is needed than the instruction set
- Computer system subsystems:
  - CPU
  - Buses
  - Memory
  - Input/Output

Basic Computing Systems Organization



System Buses

- Set of wires, that interconnects all the components (subsystems) of a computer
  - A **source** component sources out data onto the bus
  - A **destination** component inputs data from the bus
- May have a hierarchy of buses
  - Address, data and control buses to access memory and an I/O controller.
  - Second set of buses from I/O controller to attached devices/peripherals
  - PCI (Peripheral Component Interconnect) bus is an example of a very common local bus

Address Bus

- CPU reads/writes data from the memory by addressing a unique location; outputs the *location* of the data (aka address) on the address bus; memory uses this address to access the proper data
- Each I/O device (such as monitor, keypad, etc) has a unique address as well (or a range of addresses); when accessing a I/O device, CPU places its address on the address bus. Each device will detect if it is its own address and act accordingly
- Devices always receive data from the CPU; CPU never reads the address bus (it is never addressed)

Data Bus

- When the <u>CPU fetches data from memory</u>, it first outputs the address on the address bus, then the memory outputs the data onto the data bus; the CPU reads the data from data bus
- When <u>CPU is writing data onto the memory</u>, the CPU outputs first the address on the address bus, then outputs the data onto the output bus; memory then reads and stores the data at the proper location
- The process to read/write to a I/O device is similar

Control Bus

- Address and data buses consist of **n** lines, which combine to transmit one n bit value; control bus is a collection of individual <u>control signals</u>

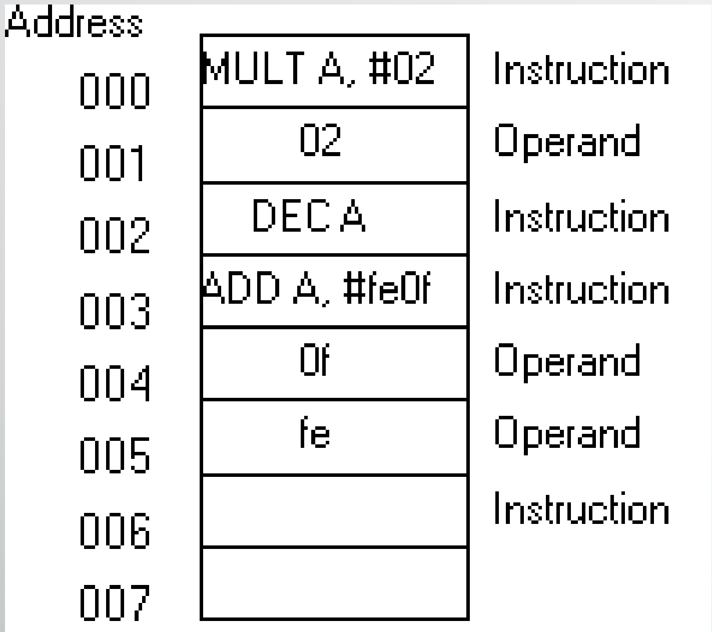
- These signals indicate whether the data is to be read into or written out the CPU, whether the CPU is accessing memory or an I/O device, and whether the I/O device or memory is ready for the data transfer

- This bus is mostly a collection of unidirectional signals

CPU

- The Central Processing Unit (CPU a.k.a. Processor) is the chip which acts as a control center for all operations. It executes instructions (a program) which are contained in the memory section.
- Basic operations involve
  - the transfer of data between itself and the memory section
  - manipulation of data in the memory section or stored internally
  - the transfer of data between itself and input/output devices
- The CPU is said to be the **brains** of any computer system. It provides all the timing and control signals necessary to transfer data from one point to another in the system.

Programs: Instructions and Operands



- A program = a number of CPU instructions.
- Instruction components:
  - an instruction code (aka OPCODE)
  - one or more operand's (data which the instruction manipulates)
- The instruction specifies to the CPU what to do, where the data is located, and where the output data (if any) will be put.
- Instructions are held in the memory section of the computer system. Instructions are transferred one at a time into the CPU, where they are decoded then executed. Instructions follow each other in successive memory locations.
- Memory locations are numbered sequentially. The CPU keeps track of the instruction it is executing by using an internal counter (location in the memory) known as the *program counter* (sometimes called *instruction pointer*).

Von Neumann and Harvard architectures

- Von Neumann
  - Allows instructions and data to be mixed and stored in the same memory module
  - More flexible and easier to implement
  - Suitable for most of the general-purpose processors

- Harvard:
  - Uses separate memory modules for instructions and for data
  - It is easier to pipeline and there are no memory alignment problems
  - Higher memory throughput
  - Suitable for DSP (Digital Signal Processors)

Computer Memory

- Memory contains instructions for the processor to execute or data it operates on
- **Address Locations -** Memory consists of a sequential number of locations, each of which are a specific number of bits wide.
  - 8 bits (PC-8088) which is a byte wide memory
  - 16 bits (XT-8086, AT-80286)
  - 32 bits (386DX, 486SX, 486DX)
  - 64 bits (Modern systems – Pentium and up)
- Each memory location is referred to as an **address**, and generally expressed in hexadecimal notation (using base 16 numbers).
- The size is denoted as the number of locations times the number of bits in each location - 32 bits limited to 2^32 which is equivalent to about 4GB of main memory
- The processor selects a specific address in memory by placing the address on the **address bus** . The value on this address bus is used by the memory system to find the data at the specific location

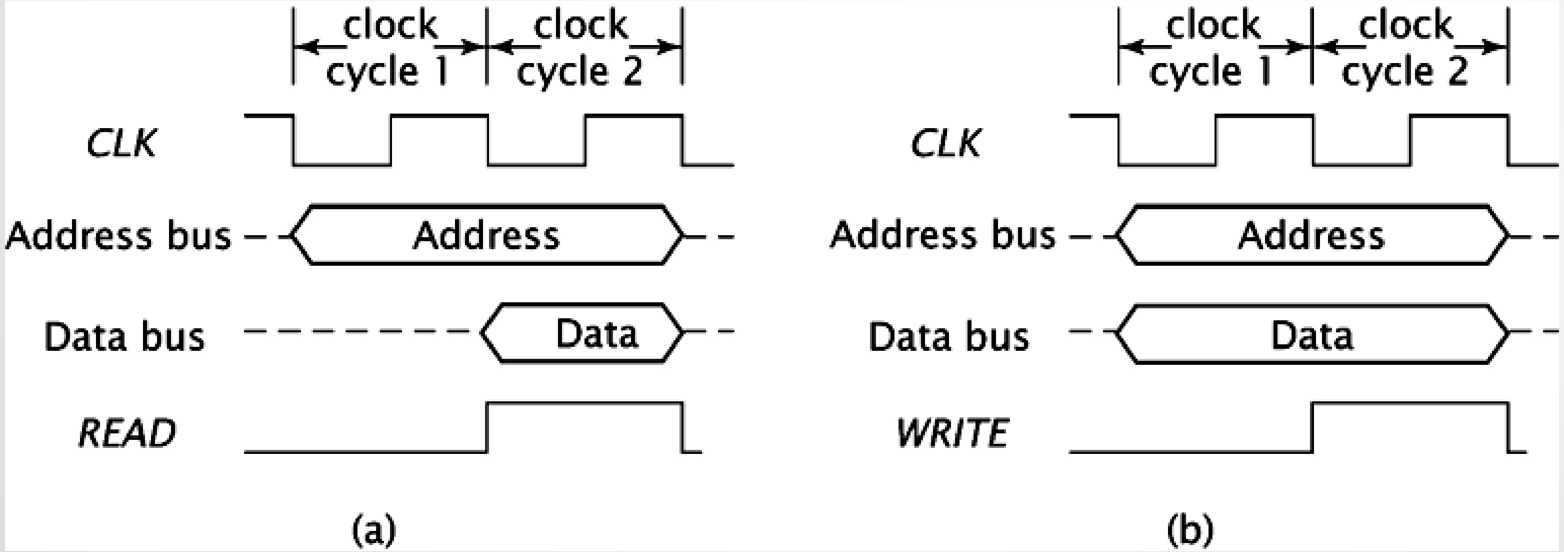
Computer Memory

- The total number of address locations which can be accessed by the processor is known as its **physical address space**. How large this is determined by the size of the address bus and is often expressed in terms of Kilobytes (x1024), Megabytes or Gigabytes.
  - 16 bits address bus = 64K (65536 locations)
  - 20 bits address bus = 1MB (IBM PC)
  - 32 bits address bus = 4GB (486DX)

- **Access Times -** Access time refers to how long it takes the processor to read or write to a specific memory location within a chip. The limiting factor is the type of technology used to implement the memory cells inside the chip.

- **Volatility -** This refers to whether or not the contents of the memory is lost when power is turned off. If the contents are lost, the memory is *volatile*. If the contents are retained, then the memory is *non-volatile*.

Memory Read/Write operations



- a) Memory read operation
- b) Memory write operation

Input/Output Devices

- Peripheral devices allow input and output to occur.
- Examples of peripheral devices are
  - disk drive controllers
  - keyboards
  - mice
  - video cards
  - parallel and serial cards
  - real-time clocks
- The processor is involved in the initialization and servicing of these peripheral devices.

I/O Read/Write Operations

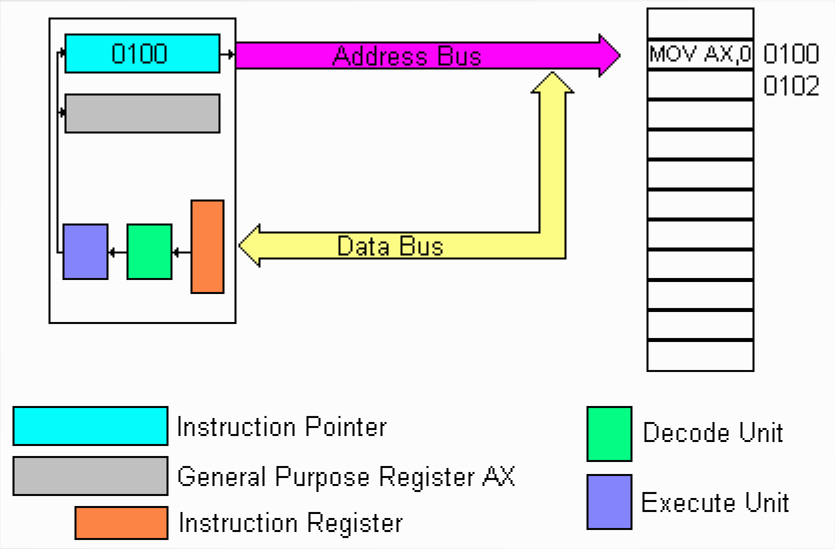
- The I/O read and write operations are similar to the memory read and write operations.

- A processor may use:
  - **memory mapped I/O** (when the address of the I/O device is in the direct memory space, and the sequence to read/write data in the device are the same with the memory read/write sequence)
  - **isolated I/O** – the process is similar, but the processor has a second set of control signals to make the distinction between a memory access and an I/O access (memory locations and I/O devices can be located at the same address, which makes this extra control signal necessary); for I/O operations, the processor holds IO/M (or similar) signal high for the duration of the I/O operation

CPU Function – Instruction Cycle

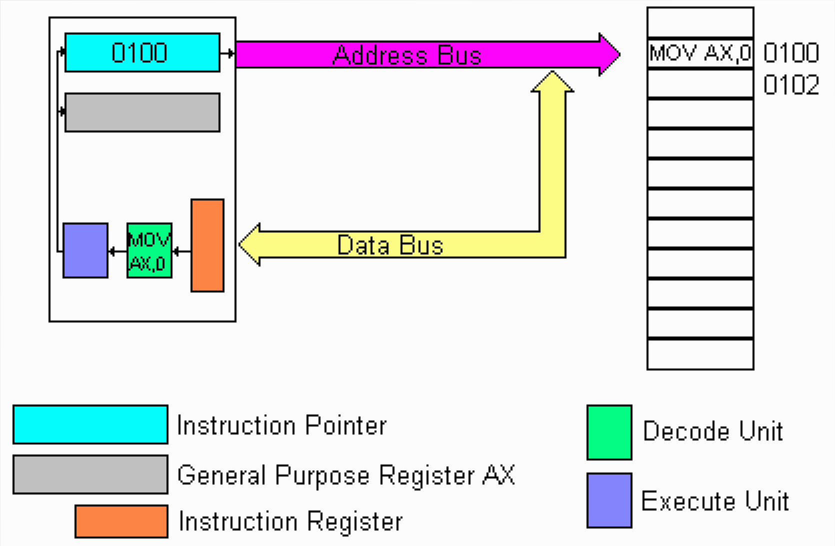
- The instruction cycle is the procedure of processing an instruction by the microprocessor:
  - Fetches (reads) the instruction from the memory
  - Decodes the instruction, determining which instruction is to be executed (what instruction has been fetched)
  - Executes the instruction – performs the operations necessary to complete what the instruction is supposed to do (different from instruction to instruction, may read data from memory, may write data to memory or I/O device, perform only operations within CPU or combination of those)
- Each of the functions fetch -> decode -> execute consist of a sequence of one or more operations inside the CPU (and interaction with the subsystems)

Fetch Cycle



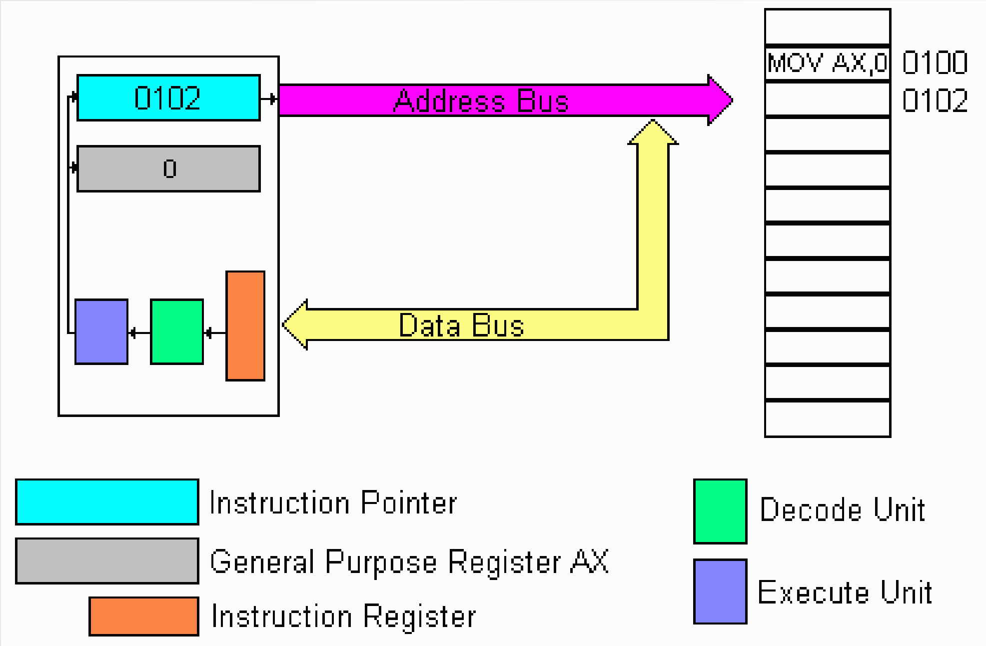
- In the first phase, the processor generates the necessary timing signals to fetch the next instruction from the memory system.
- The instruction is transferred from memory to an internal location inside the processor (the instruction register)
- In the above image, the processor is ready to begin the Fetch cycle. The current contents of the **instruction counter (program counter)** is address 0100. <u>This value is placed on the address bus</u>, and a <u>**READ signal** is activated on the control bus</u>. The memory receives this and finds the contents of the memory location 0100, which happens to be the instruction MOV AX, 0.
- The memory places the instruction on the Data Bus, and the processor then copies the instruction from the Data Bus to the Instruction Register.

Decode Cycle



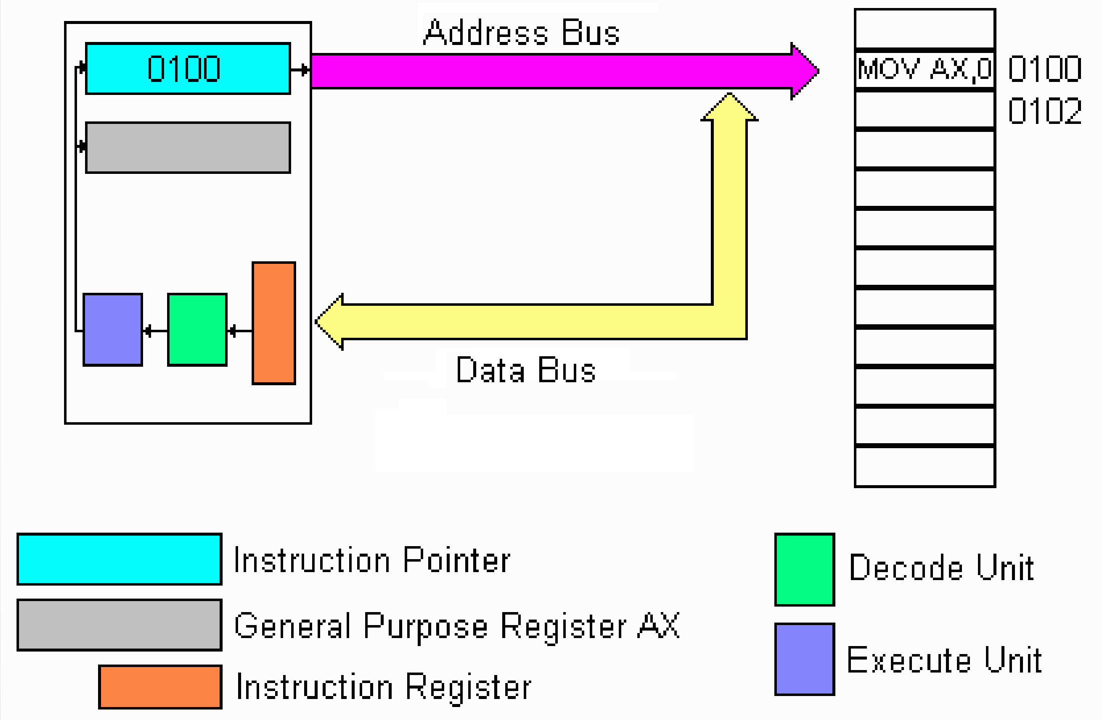
- The processor transfers the instruction from the instruction register to the Decode Unit.
- It compares the instruction to an internal table, and when a match is found, the table contains the list of macro instructions (a number of steps) which are required to perform the instruction.
  - In our case, the instruction means place the value 0 into the AX register. The decode unit now has all the details of how to do this.
- During this phase the processor (if required by the instruction) will get any operands required by the instruction.
  - The final effect of instruction MOV AX, 0 is to set the value of the AX register of the processor to the constant value 0. The processor has the instruction (MOV AX), but now needs the constant value 0 to complete the instruction before executing it. In this instance, the processor will fetch the constant value 0 from the next location in memory (it is found immediately after the instruction, in the next memory location 0101)

Execute Cycle



- In the last phase, the processor executes the instruction. In the example above, this involves setting the contents of the internal register AX to the constant value 0
- The final part of execute phase is to adjust the **Instruction Counter** to point to the next instruction to be executed, which is found at address 0102

Fetch/Decode/Execute



Animated fetch decode execute

References

- “Computer Systems Organization & Architecture”, John D. Carpinelli, ISBN: 0-201-61253-4
- “Operating Systems – A Modern Perspective”, Garry Nutt, ISBN 0-8053-1295-1

OÉ Gaillimh
NUI Galway

- CT101 -
Computing Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Sequential Logic Design

Contents

- Basic Sequential Components
  - Flip-Flop, Latches, Counters
- Programmable Logic Devices
  - PLD, PLA, CPLDs & FPGAs

Overview

- The most fundamental sequential components are the latch and flip-flop

- They store one bit of data and make it available to other components

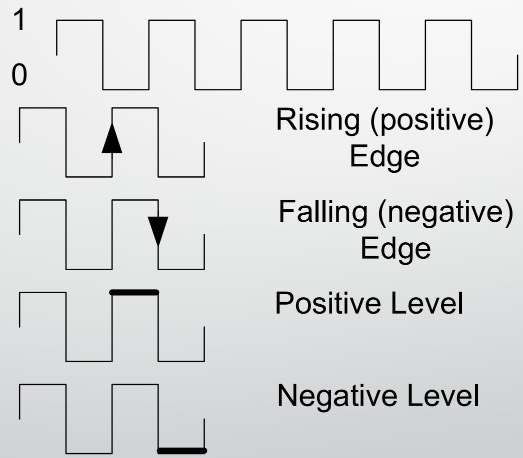
- The main difference between a latch and a flip-flop is that the former is *level-triggered* and the latter are *edge triggered*

- Flip-flops and latches have a **clock** input

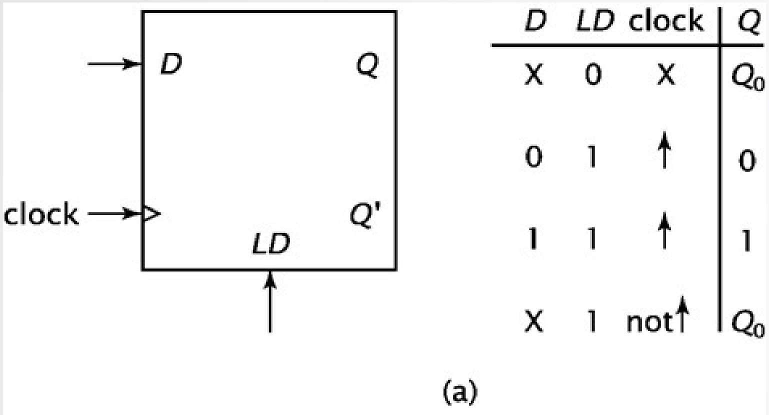
Clock

- It is usually derived from an oscillator or other circuitry that alternates its output between 1 and 0

- It is used to synchronize the flow of data in a digital system

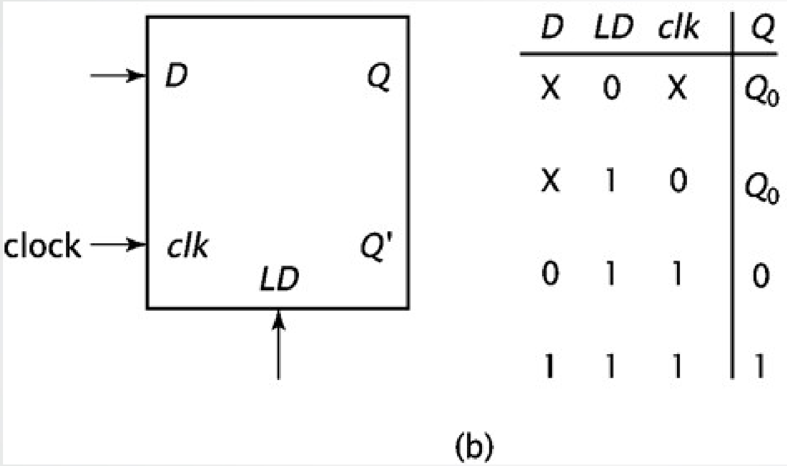


D flip-flop



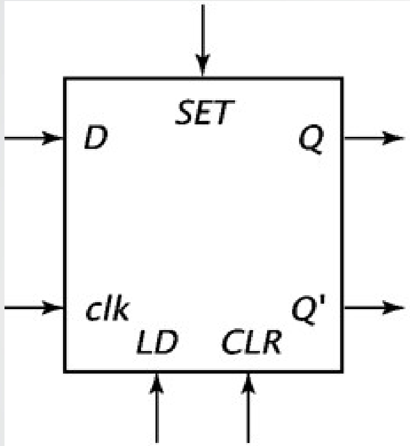
- Flip-flop:
  - One data input D
  - When the clock input changes from 0 to 1 (positive edge), the data on the D input is loaded
  - The data is made available via output Q and its complement via Q'
  - Some variations have also a load signal (LD) that has to be high (active) in order for data to be loaded into the flip-flop

D latch



- Positive level triggered latch
  - It loads data as long as both its clock and load signals are 1. If both are one, the value of data D is passed to the Q output. If D changes while clock and load are 1, then the output changes accordingly
  - If either the clock or load signals go to 0, the Q value is latched and held

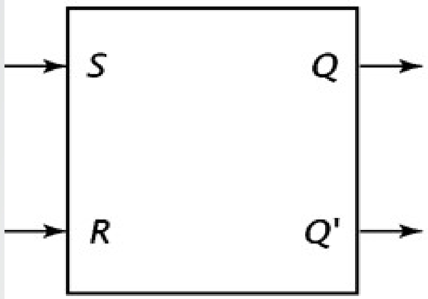
D latch with clear/set capabilities



D	LD	clk	SET	CLR	Q
X	X	X	1	X	1
X	X	X	0	1	0
X	0	X	0	0	Q_0
X	1	0	0	0	Q_0
0	1	1	0	0	0
1	1	1	0	0	1

- Some variants of D latch and flip-flops have asynchronously set and clear capabilities – they can be set and clear regardless of the value of the other inputs to the latch (including the clock and load inputs)

SR latch

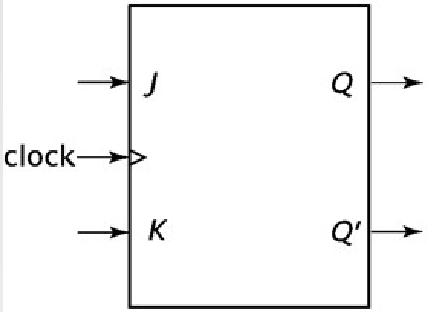


S	R	Q
0	0	Q_0
0	1	0
1	0	1
1	1	Undefined

- The S input sets the latch to 1 and the R input resets the latch to 0
  - When both S and R are 0 the output remains unchanged

- Doesn’t have a clock input
  - Only sequential component without a clock input
  - The output of the latch is undefined when both the S and R are 1; the designer has to ensure that S and R inputs are never set to 1

JK flip-flop

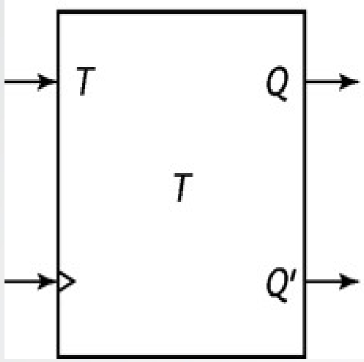


J	K	clock	Q
X	X	not $\uparrow$	Q_0
0	0	$\uparrow$	Q_0
0	1	$\uparrow$	0
1	0	$\uparrow$	1
1	1	$\uparrow$	Q_0'

- Resolves the problem of undefined outputs associated with SR latch
  - J=1 sets the output to 1 and K=1 resets the output to 0. JK=11 inverts the stored current value of the output

- It is often used instead of SR latch

T (toggle) flip-flop



T	Q
0	Q_0
1	Q_0'

- The T input doesn't specify a value for its output, it specifies only whether or not the output should be changed
- On the rising edge of the clock, if T = 0 then the output of the flip-flop is unchanged; if T=1, the output is inverted.

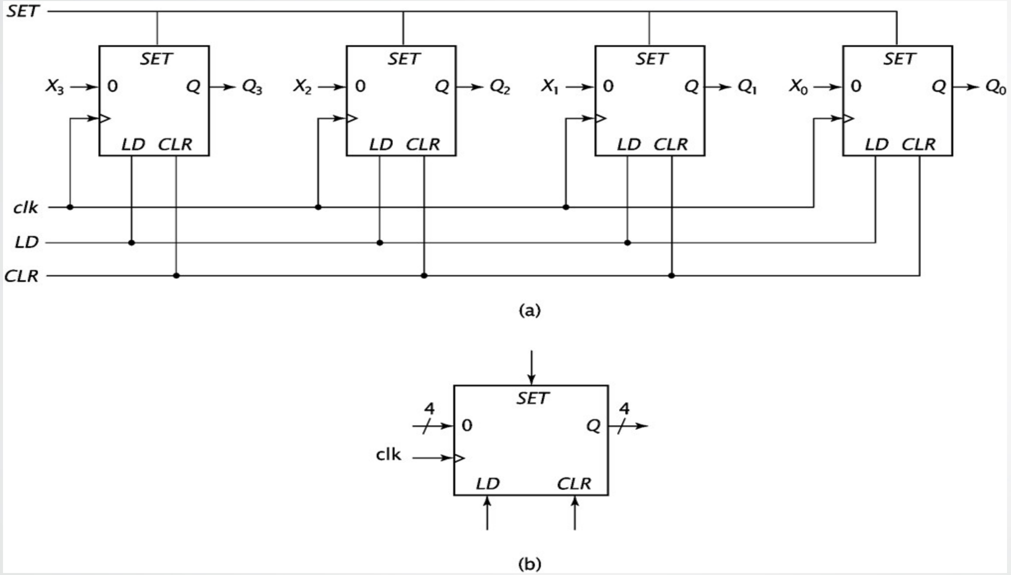
Observations

- All of the flip-flops and latches shown so far are positive edge triggered or positive level triggered. They also have active high load, set and clear inputs.

- It is possible for those components to be negative edge triggered or negative level triggered and have active low control signals as well.

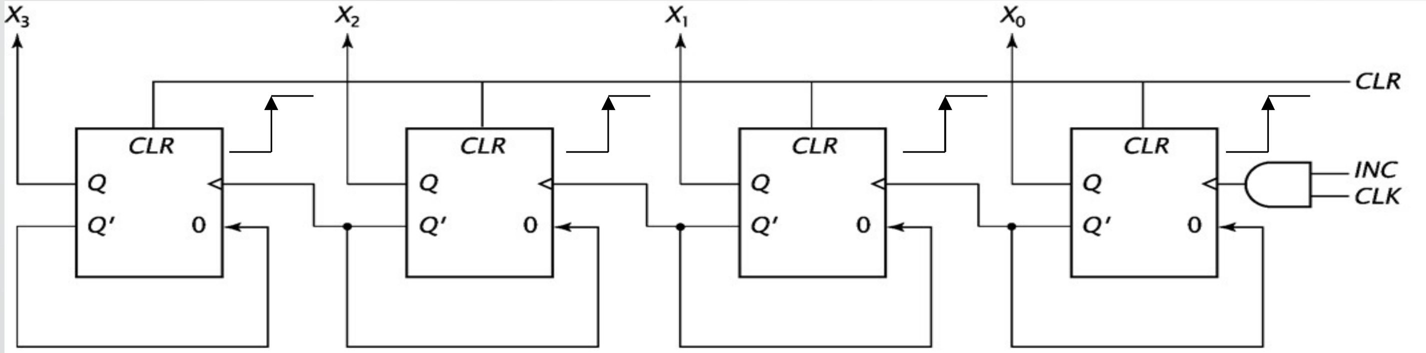
- Flips-flops and latches can be combined in parallel to store data with more than one bit

4-bit D flip-flop



- Control signals are tied together
- Act as one unified data register
- They usually output only the data (not the complement of the data as the 1-bit flip-flops)

Counters



(a)

CLR	INC	CLK	X_{3-0}
1	X	X	0
0	0	X	X_{3-0}
0	1	not↑	X_{3-0}
0	1	↑	$X_{3-0}+1$

(b)

INC=1

Current Counter Value: 1111

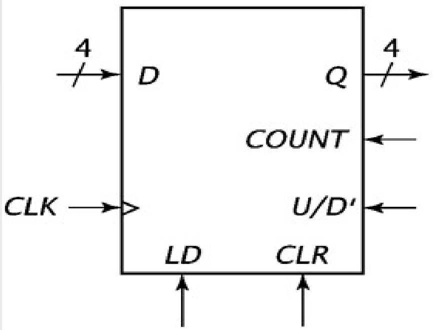
Next Counter Value: 0000

- Store a binary value and when signaled to do so, it increments or decrements its value
- Can be loaded with an externally supplied value

Counters

- Counters can be designated as *asynchronous* or *synchronous*
- Asynchronous counters are relatively slow because the output from one flip-flop triggers a change in the status of the next flip-flop
- In a synchronous counter, all of the flip-flops change state at the same time. This is the kind used in CPUs

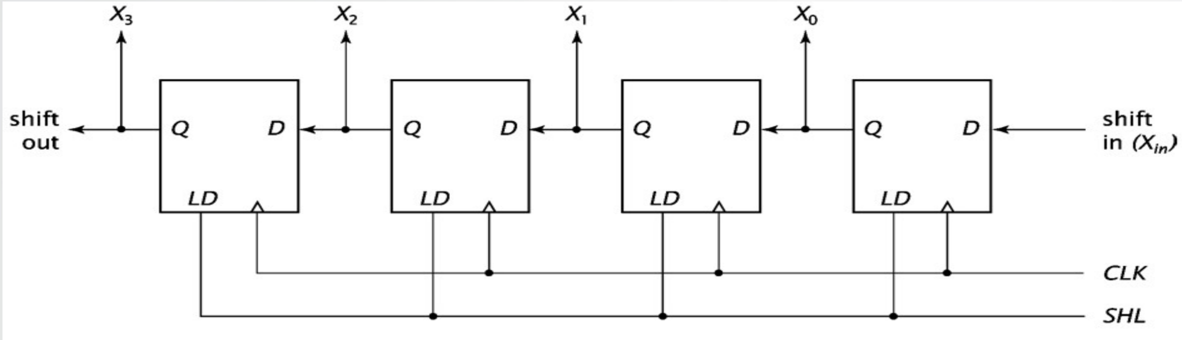
Up/down counter with parallel load



CLR	LD	COUNT	U/D'	CLK	Q
1	X	X	X	X	0
0	X	X	X	not $\uparrow$	Q_0
0	1	X	X	$\uparrow$	D
0	0	0	X	$\uparrow$	Q_0
0	0	1	0	$\uparrow$	$Q_0 - 1$
0	0	1	1	$\uparrow$	$Q_0 + 1$

- Ability to load external data as well as count
- Down counter decrements its value rather than increment and generates a borrow rather than a carry out
- Up/down counter can do both operations according to the signal U/D'

Shift Registers



SHL	CLK	X_{3-0}
0	X	X_{3-0}
1	not↑	X_{3-0}
1	↑	$X_2\, X_1 X_0\, X_{in}$

- Can shift its data one-bit position to the right or left
- It is useful for hardware multipliers/dividers
- It may shift left, right or both directions under certain control conditions (like the up/down counter)

OÉ Gaillimh
NUI Galway

- CT101 -
Computing Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Programmable Logic Devices

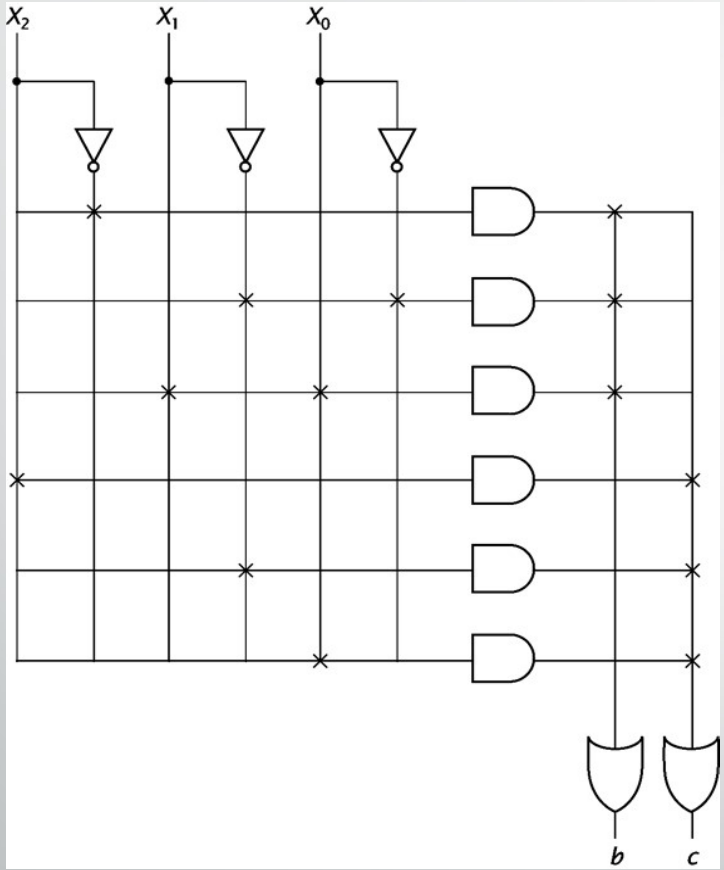
- Most of the circuits presented so far are available on a TTL IC chip. Circuits can be constructed using these chips and wiring them together

- An alternative to this method would be to program all the components into a single chip, *saving wiring, space and power*

- One type of such device is PLA (***P***rogrammable ***L***ogic ***A***rray) that contains one or more *and/or* arrays.

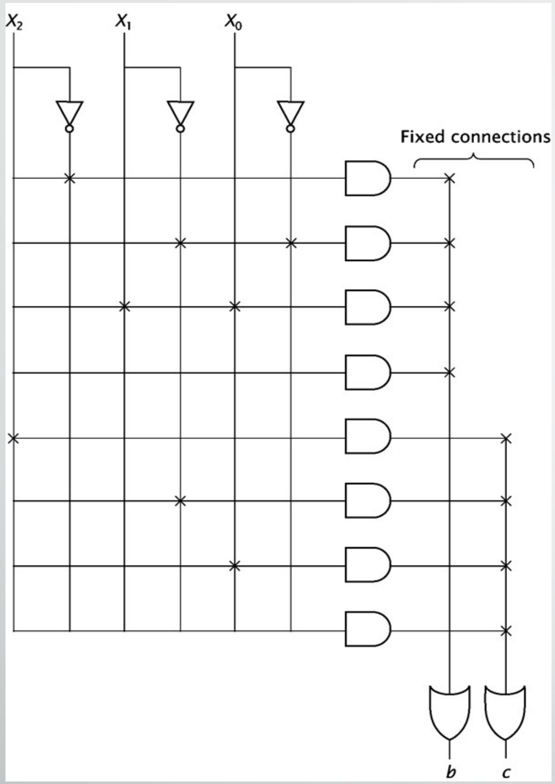
Programmable Logic Array (PLA)

OÉ Gaillimh
NUI Galway



- The inputs and their complements are made available to several AND gates.
  - An X indicates that the value is input to the AND gate
  - The output from the AND gates are input into the OR gates, which produce the chip's outputs
- Functions:
  - b = X2' + X1'X0'+X1X0
  - c = X2 + X1' + X0

Programmable Array Logic (PAL)



- ***P***rogrammable ***A***rray of ***L***ogic – its **OR** blocks are **not** programmable
  - Certain AND gates serve as input to specific OR gates
  - Same b and c function implementation:

  b = X2’ + X1’X0’+X1X0

  c = X2 + X1’ + X0

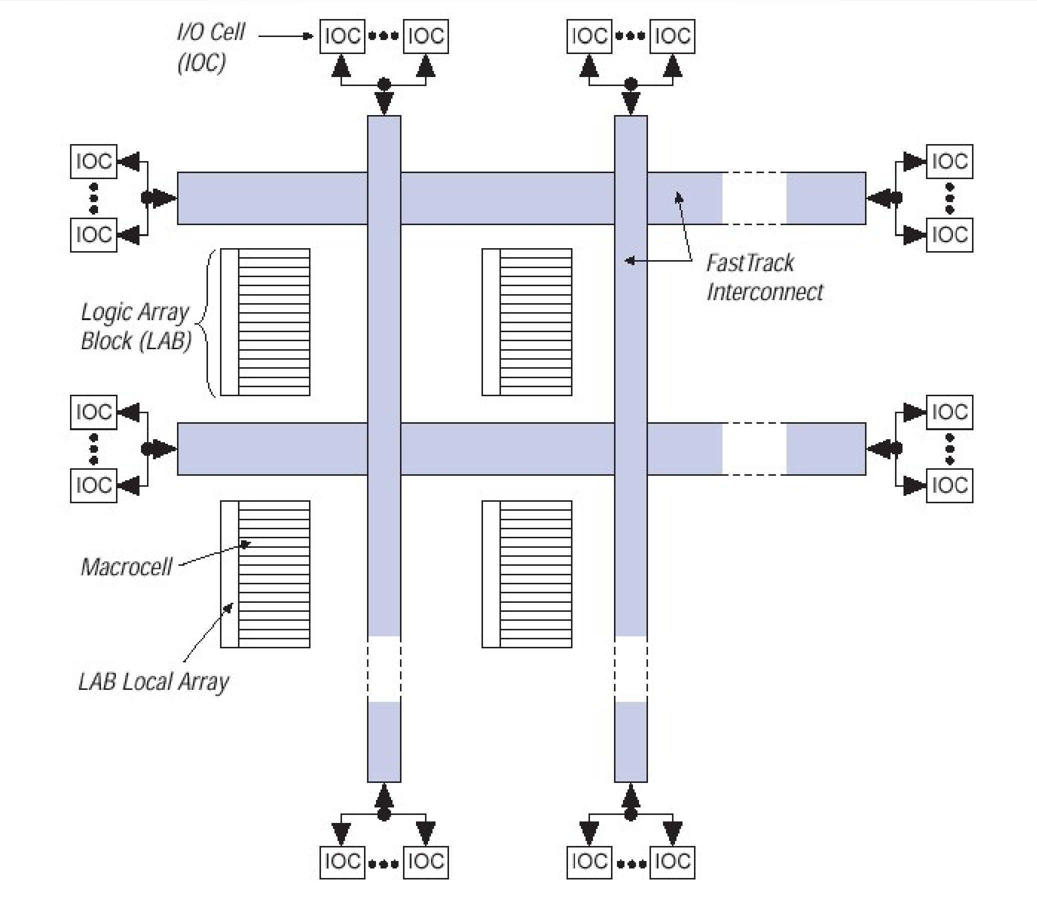
- PLA and PAL are limited because they can implement only combinatorial logic, they don’t contain any latches nor flip-flops

Programmable Logic Device (PLD)

- ***P***rogrammable ***L***ogic ***D***evice is a more complex component that is needed to realize sequential circuits
- It is usually made up of logic blocks with the possibility to interconnect them.
- Each logic block is made out of macro cells, that may be equivalent to a PAL with an output flip-flop
- The input/output pins of an PLD can be configured to the desired function (unlike for PLA or PAL, where they are fixed)
- Used in more complex design than the PAL or PLA

Complex Programmable Logic Device (CPLDs)

- Array of PLDs
- Has global routing resources for connections between PLDs and between PLDs to/from IOs



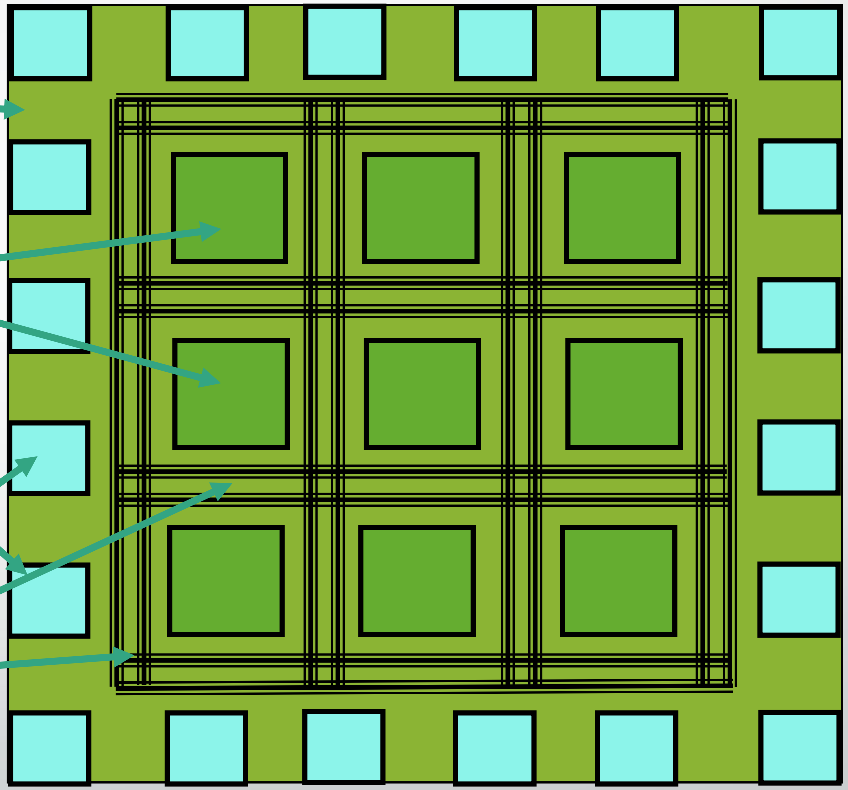
Field Programmable Gate Array (FPGAs)

OÉ Gaillimh
NUI Galway

- ***F***ield ***P***rogrammable ***G***ate ***A***rray is one of the most powerful and complex programmable circuit available
- Contain an array of cells, each of which can be *programmed to realize a function*
- There are programmable interconnects between the cells, allowing connections to each other
- Includes flip-flops allowing the design and implementation of complex sequential circuit on a chip (of the complexity of a processor)
- Often contains the equivalent of 100k to a few million simple logic gates on a single chip

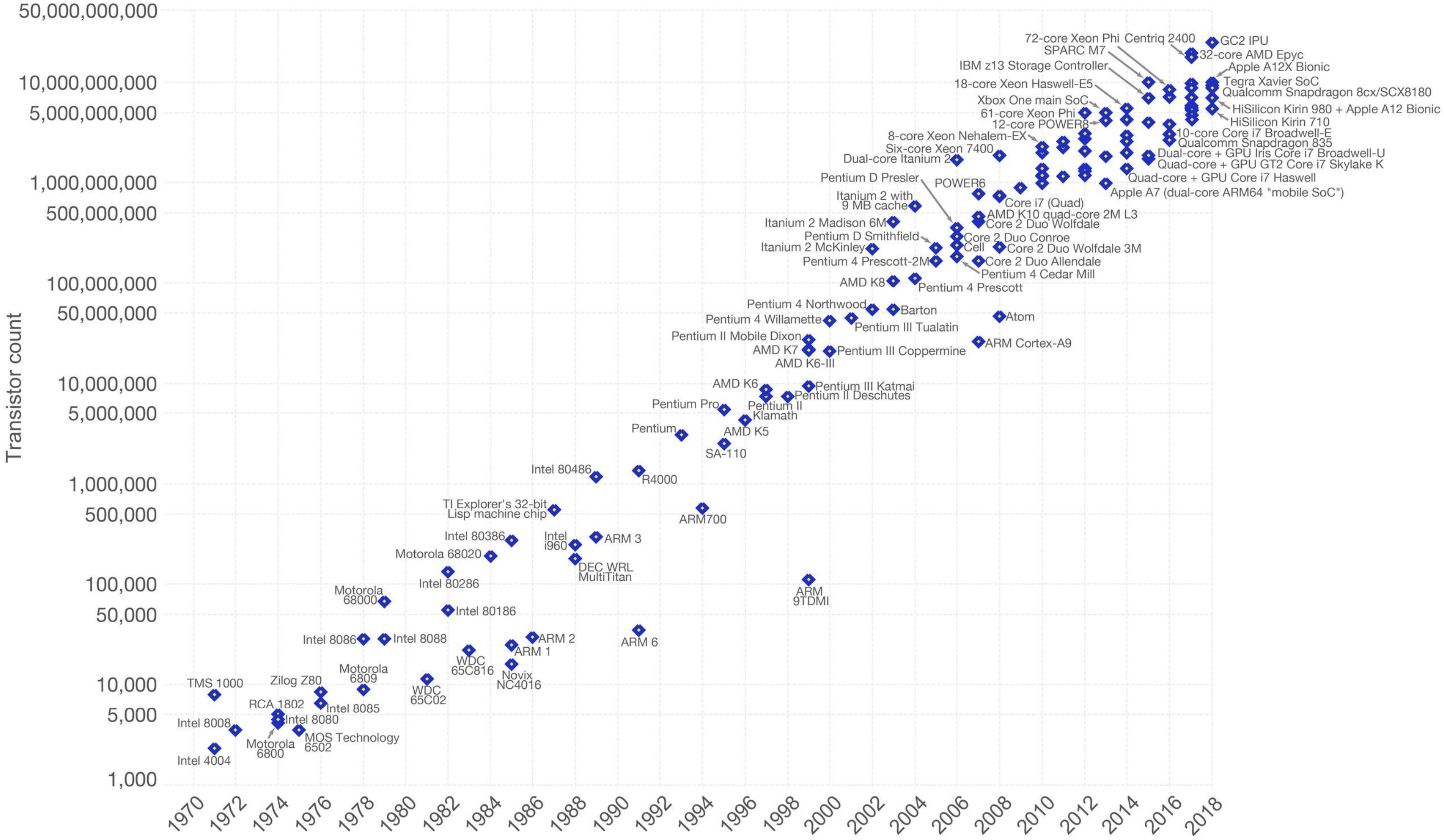
FPGAs

- Configuration Memory
- Programmable Logic Blocks (PLBs)
- Programmable Input/Output Cells
- Programmable Interconnect



Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.



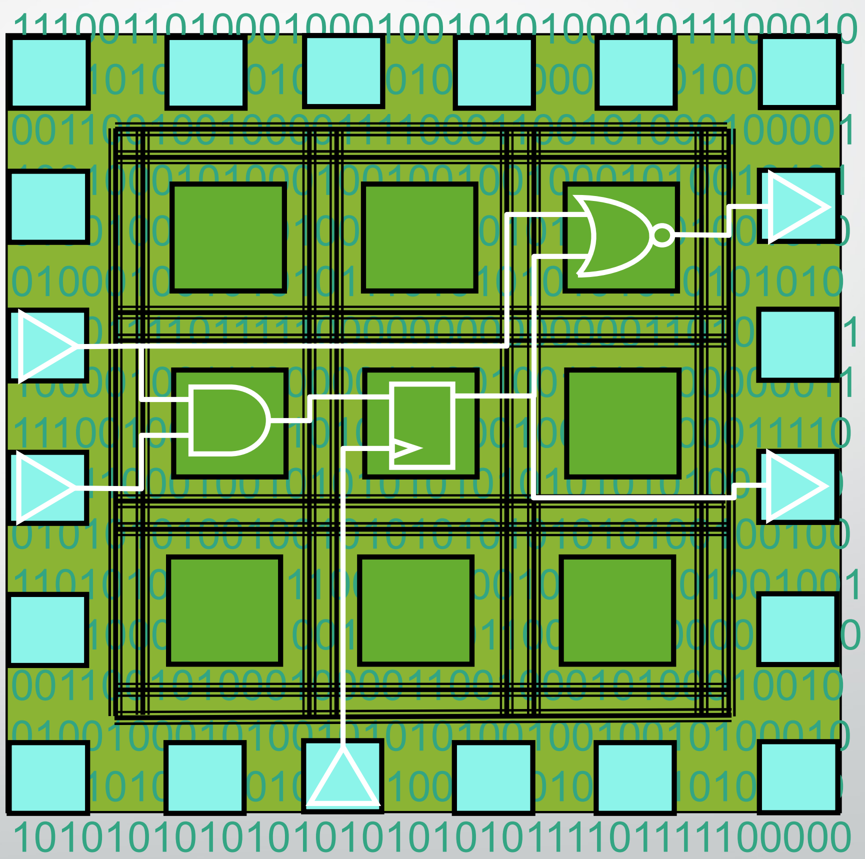
Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Our World in Data

illimh
alway

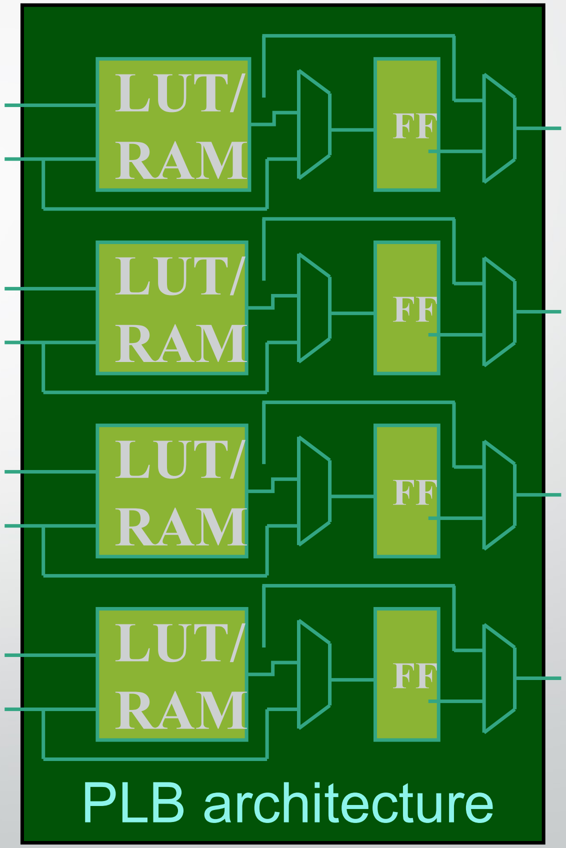
Basic FPGA Operation

- Load Configuration Memory
  - Defines system function (Input/Output Cells, Logic in PLBs, Connections between PLBs & I/O cells)
- Changing configuration memory => changes system function
- Can change at anytime
  - Even while system function is in operation
  - Run-time reconfiguration (RTR)



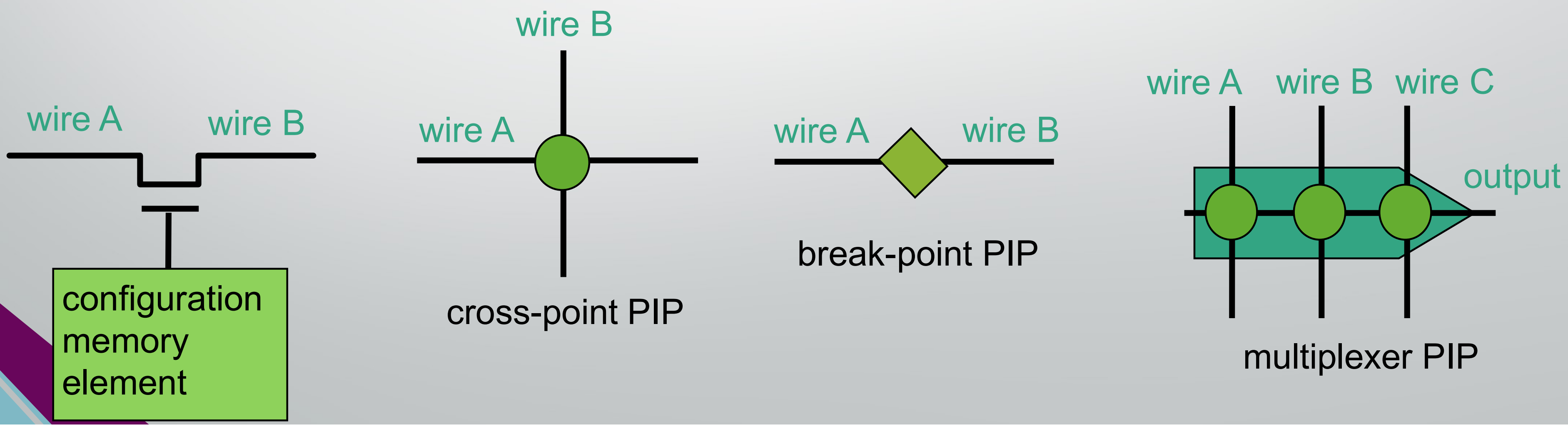
Programmable Logic Blocks

- PLBs can perform any logic function
  - Look-Up Tables (LUTs)
    - Combinational logic
    - Memory (RAM)
  - Flip-flops
    - Sequential logic
  - Special logic
    - Add, subtract, multiply
    - Count up and/or down
- #PLBs per FPGA: 100 to 500,000



Programmable Interconnect

- Wire segments & Programmable Interconnect Points (PIPs)
  - cross-point PIPs – connect/disconnect wire segments
    - To turn corners
  - break-point PIPs – connect/disconnect wire segments
    - To make long and short signal routes
  - multiplexer (MUX) PIPs select 1 of many wires for output
    - Used at PLB inputs
    - Primary interconnect media for new FPGAs



References

- “Computer Systems Organization & Architecture”, John D. Carpinelli, ISBN: 0-201-61253-4

- CT101 -
Computing Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Finite State Machine Design

Contents

OÉ Gaillimh
NUI Galway

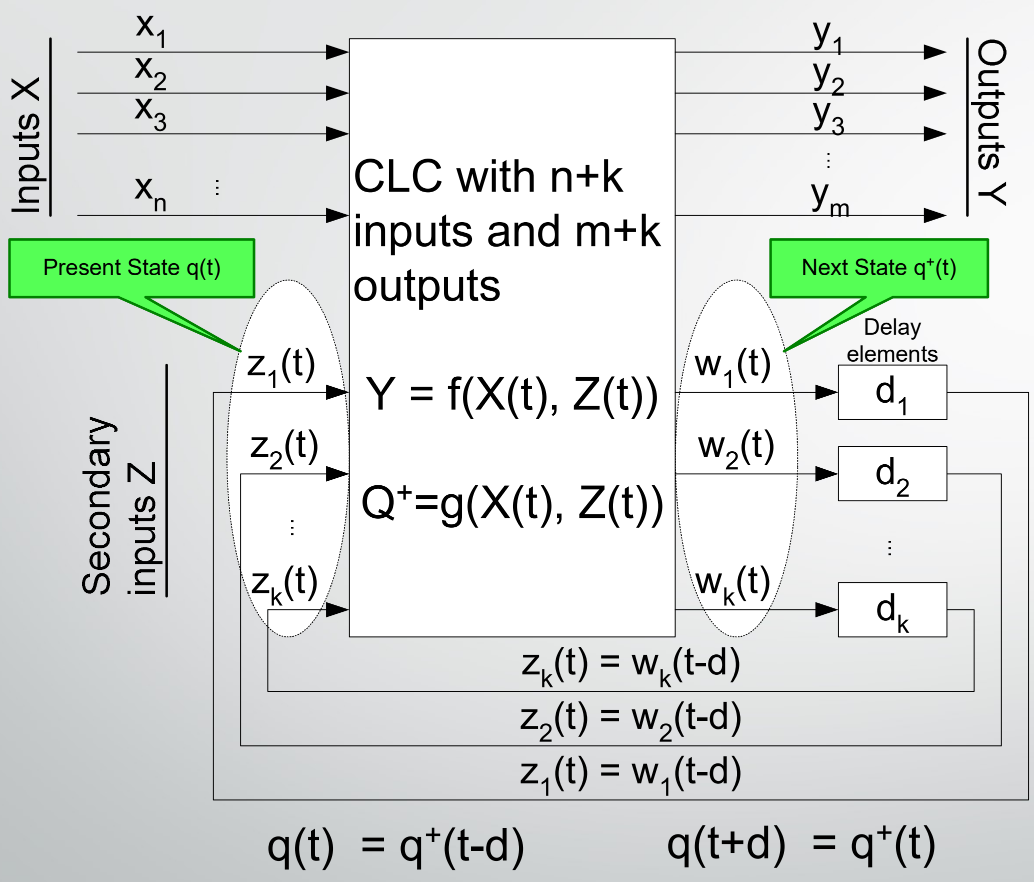
- Finite State Machine (FSM) theory
- Front to end design of FSMs, both Mealy and Moore types.
  - Design example is provided for a Modulo 6 counter
- Other Design Examples
  - String Checker
  - Tollbooth Controller

FSM Overview

OÉ Gaillimh
NUI Galway

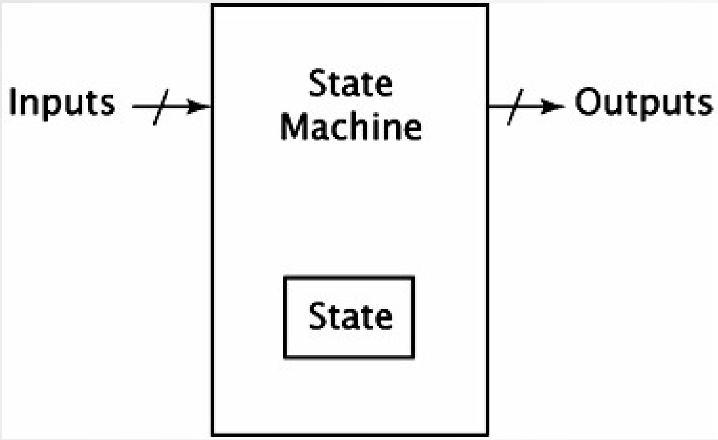
- ***F**inite **S**tate **M**achine* is a tool to model the desired behavior of a sequential system.
  - The designer must develop a finite state model of the system behavior and then designs a circuit that implements this model
- An FSM consists of several ***states***. ***Inputs*** into the machine are combined with the current state of the machine to determine the new state or ***next*** state of the machine.
- Depending on the state of the machine, outputs are generated based on either the state or the state and inputs of the machine.

FSM Structure



- X represents the range of possible **input** values (2^n)
- Y represents the range of **output** values (2^m)
- Q represents the range of the possible **states** of the system (2^k)
- Transfer functions:
  - f: X x Q -> Y
  - g: X x Q -> Q

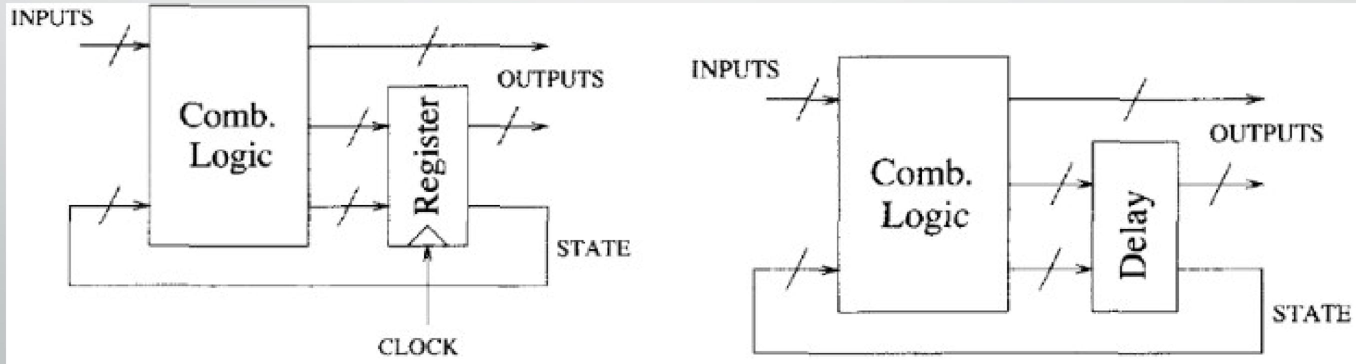
FSM Representation



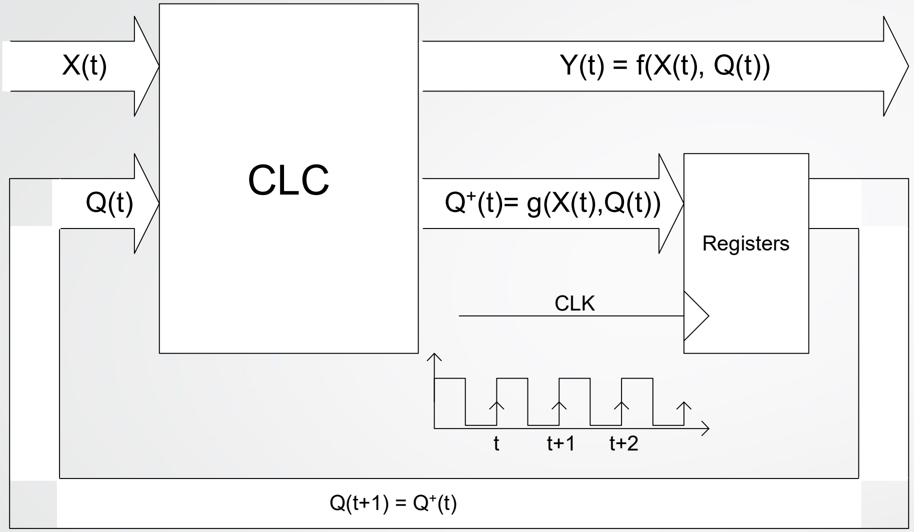
- FSM = (X, Y, Q, f, g)
  - If there is no state in the Q range (Q≡Ø, the circuitry has no history), then:
    - g: X x Ø->Ø, there is no state transition function
    - f: X x Ø -> Y is becoming f: X -> Y
  - In this case, the FSM is equivalent to a CLC
    - FSM| $_{Q≡Ø}$ = CLC = (X, Y, f)

Asynchronous vs. Synchronous

- **Async FSM** – the next state becomes the present state after the delays through the delay elements
- **Sync FSM** – obtained by replacing the delay elements d_i with memory elements (registers).
  - The w_i bits of the next state will be written in the registers (memory elements) *only* on the clock (on edge or level).



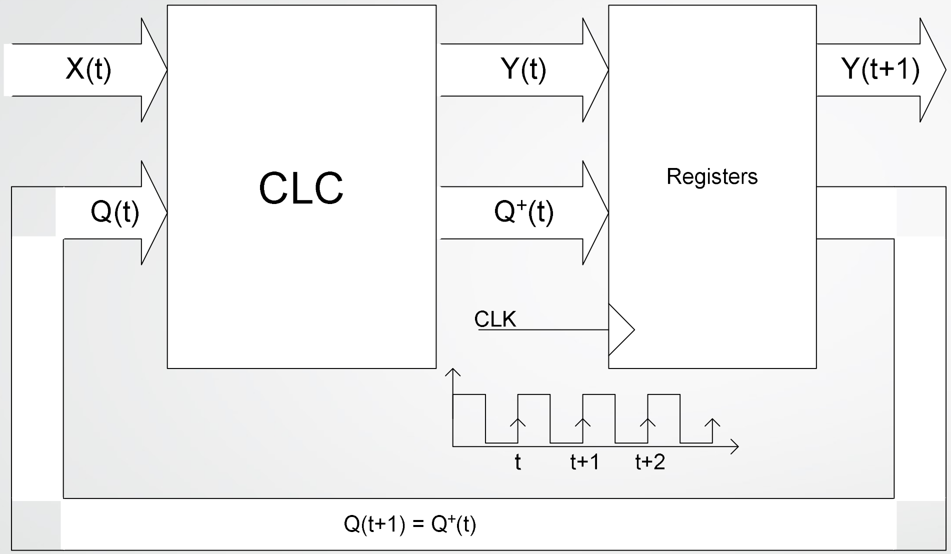
Sync FSM with Immediate Outputs



Synchronous FSM with immediate outputs

The FSM where the outputs, after they have been calculated, are used immediately (of course in the *stable* period of the state interval), is called an immediate state machine.

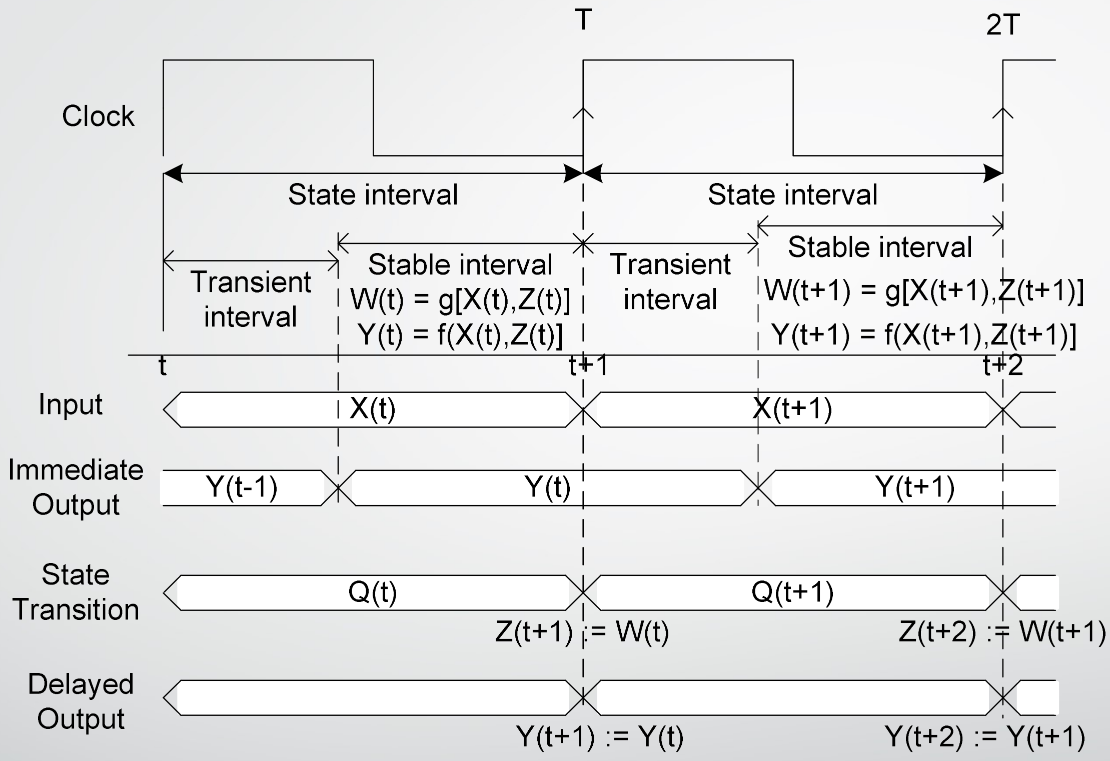
Sync FSM with Delayed Outputs



Synchronous FSM with delayed outputs

The next state is assigned as present state on the next clock cycle. Similarly, we can proceed with the outputs, obtaining the delayed state machine. Each bit of the output is passed through a memory element.

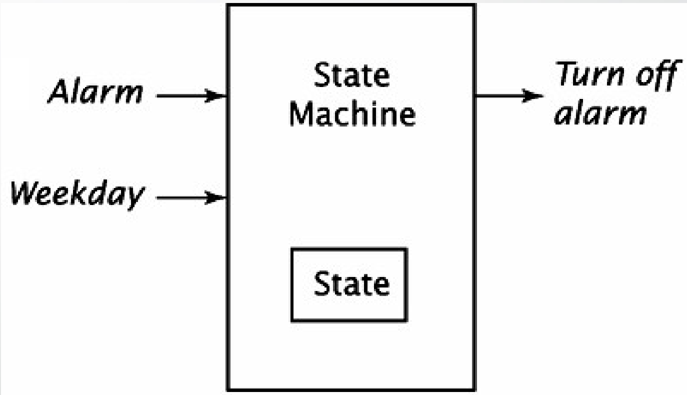
Timing diagram for Synchronous FSM



Timing diagram for synchronous FSM

FSM Example

- Events:
  - Wake up at fixed time every day
  - Weekends: you don't need alarm, so you wake up, turn off the alarm and resume sleep
- FSM modeling this chain of events, with:
  - Three states:
    - Asleep
    - Awake but still in bed
    - Awake and up
  - Inputs:
    - Alarm
    - Weekday (determines you how to react to alarm)
  - Outputs:
    - Turn off the alarm



State Tables

Present State	Inputs	Next State	Outputs

- Similar to the truth table
  - Doesn't contain the system clock when specifying its transitions (it is implicit that transitions occur only when allowed by clock)

- Unless otherwise stated, all the transitions are occurring on the *positive edge* of the clock

Alarm Clock State Table

OÉ Gaillimh
NUI Galway

Present State	Alarm	Weekday	Next State	Turn off alarm
Asleep	On	X	Awake in bed	Yes
Awake in bed	Off	Yes	Awake and up	No
Awake in bed	Off	No	Asleep	No

- When you are asleep and alarm goes on, you go from being asleep to being awake in bed; you also turn off the alarm
- The next two rows encode your actions:
  - You get up
  - You go back to sleep
- This table doesn't cover what you wouldn't do…(i.e. if you are asleep and the alarm doesn't go off, you remain asleep, etc..)

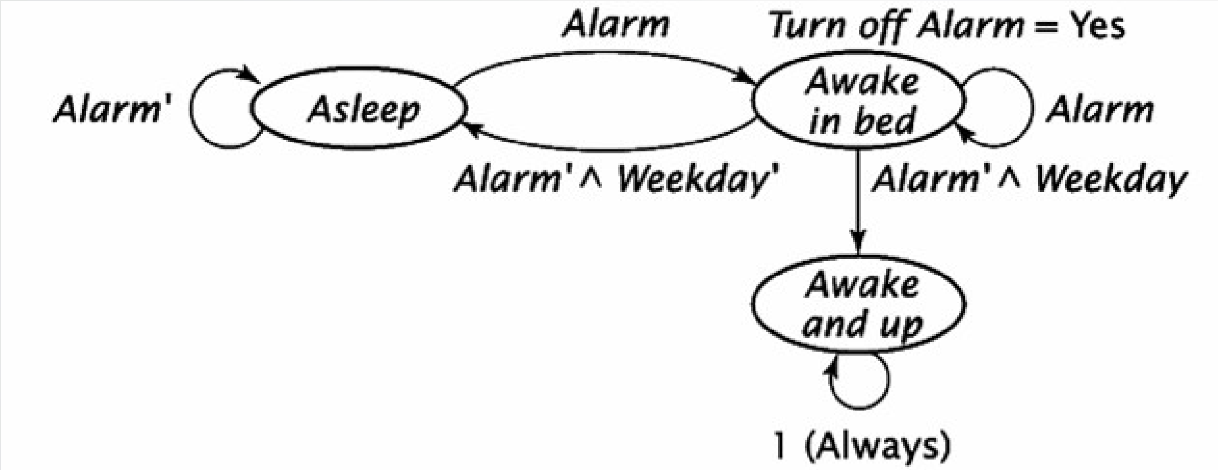
Alarm Clock State Table

OÉ Gaillimh
NUI Galway

Present State	Alarm	Weekday	Next State	Turn off alarm
Asleep	Off	X	Asleep	No
Asleep	On	X	Awake in bed	Yes
Awake in bed	On	X	Awake in bed	Yes
Awake in bed	Off	Yes	Awake and up	No
Awake in bed	Off	No	Asleep	No
Awake and up	X	X	Awake and up	No

- Covers all the cases
  - First row covers the situation you are asleep, the alarm **doesn't** go off and you remain asleep
  - Last row covers the situation you are awake and up and you remain awake and up
  - The third row covers the case you are already up and the alarm goes off. You turn it off and remain Awake in bed

State Diagram

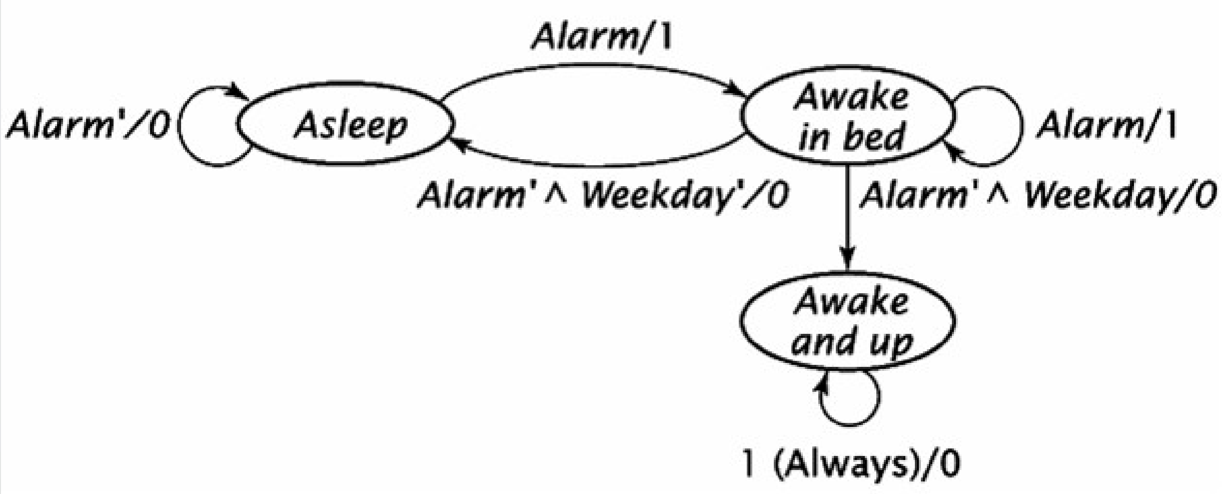


- Graphical representation of the state table
  - Each state is represented by a circle **vertex**
  - Each row of the state table is represented as a directed **arc** from present state vertex to the next state vertex

- In this diagram, the outputs are associated with the states

School of Computer Science

Alternative State Diagram



- The outputs are associated with the arcs
  - An output of 1 represents that "turn off the alarm" is Yes
  - By convention, inputs which we don't care about and inactive outputs are not shown.

- CT101 -
Computing Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Mealy and Moore machines

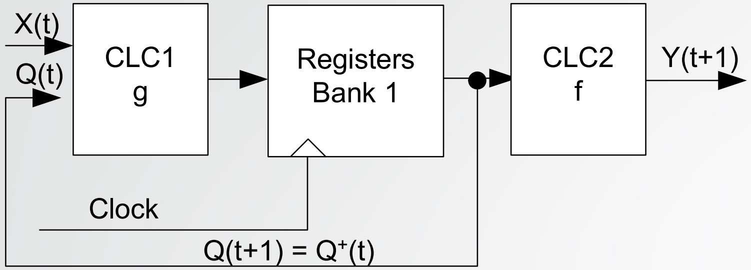
OÉ Gaillimh
NUI Galway

- **Moore machine:**
  - Associates its outputs with states
  - The outputs are represented either within the vertex corresponding to a state or adjacent to the vertex

- **Mealy machine:**
  - Associates its outputs with the transitions
  - In addition to the input values, each arc also shows the output values generated during the transition; the format of the label of each arc is Inputs/Outputs

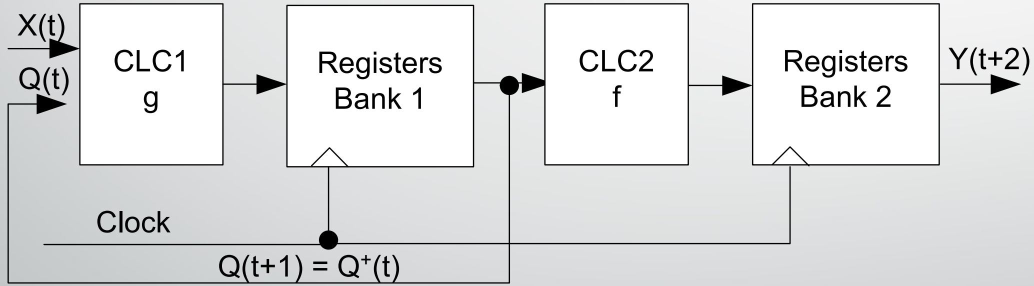
- Both can be used to represent any sequential system and each has its advantages.

Moore FSM



Moore with immediate output

$Q^+(t) = g[(X(t), Q(t)]$
$Q(t+1) = Q^+(t)$
$Y(t+1) := f[Q^+(t)]$



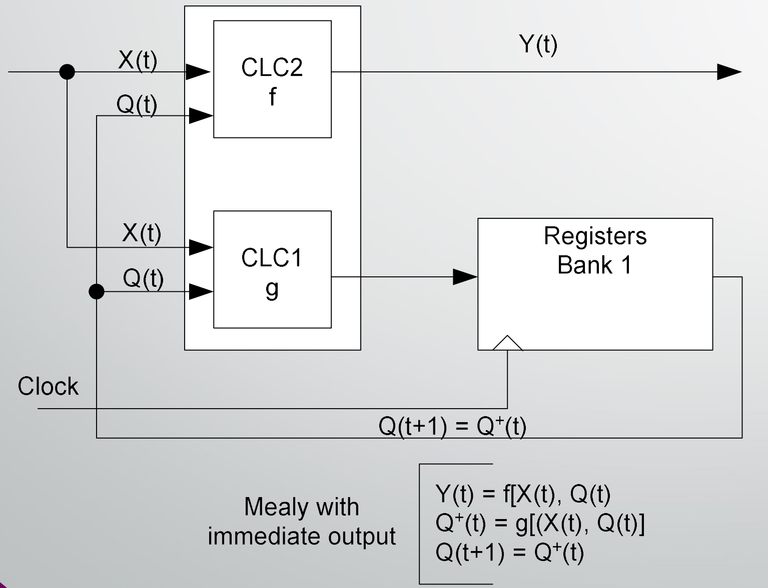
Moore with delayed output

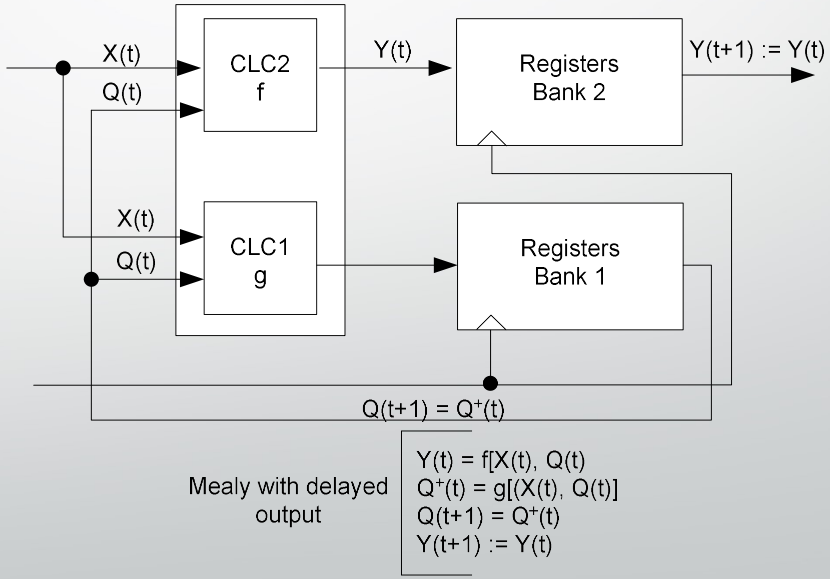
$Q^+(t) = g[(X(t)]$
$Q(t+1) = Q^+(t)$
$Y(t+2) := f[Q^+(t)]$

- Output is dependent **only** on the current state
- *Immediate* Moore FSM: the output is obtained with a clock period delay, since the next state becomes present state
- *Delayed* Moore FSM: the output is actually obtained with two clock period delay, because of the Registers Bank 2

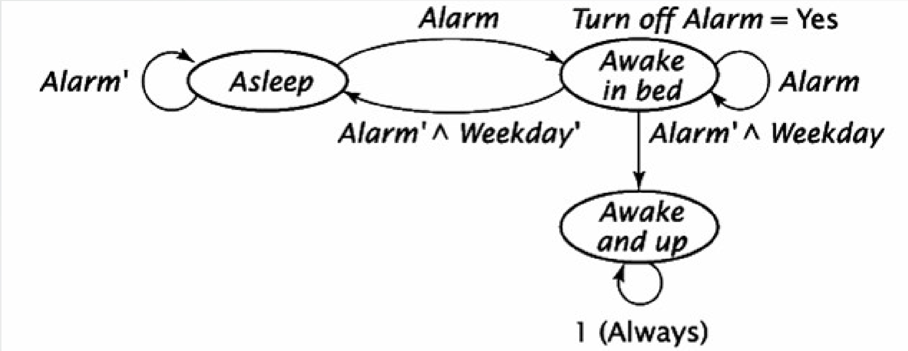
Mealy FSM

- Output is dependent on the inputs **and** the current state
- Delayed output FSM implies the fact that the calculated output for an input, applied at time t, is assigned at time t+1. This is correct for a Mealy FSM





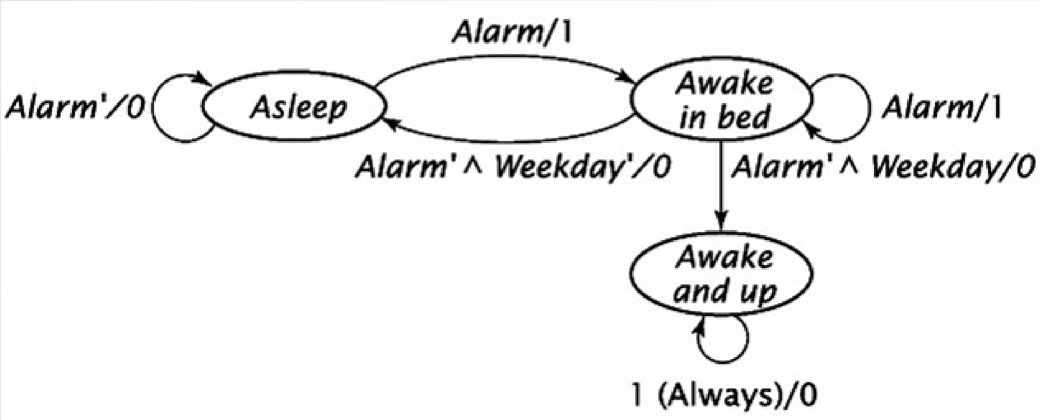
Moore Machine Diagram



- Self arcs can be missing (since its outputs are associated with the states and not with the arcs)
- Offers a simpler implementation when the output values depend only on the state and not on the transitions
- It is well suited for representing the control units of microprocessors

Mealy Machine Diagram

OÉ Gaillimh
NUI Galway



- Self arcs must be shown (because the output values are shown on the arcs)

- Can be more compact than Moore machine, especially when two or more arcs with different output values go into the same state

Modulo 6 Counter - Specification

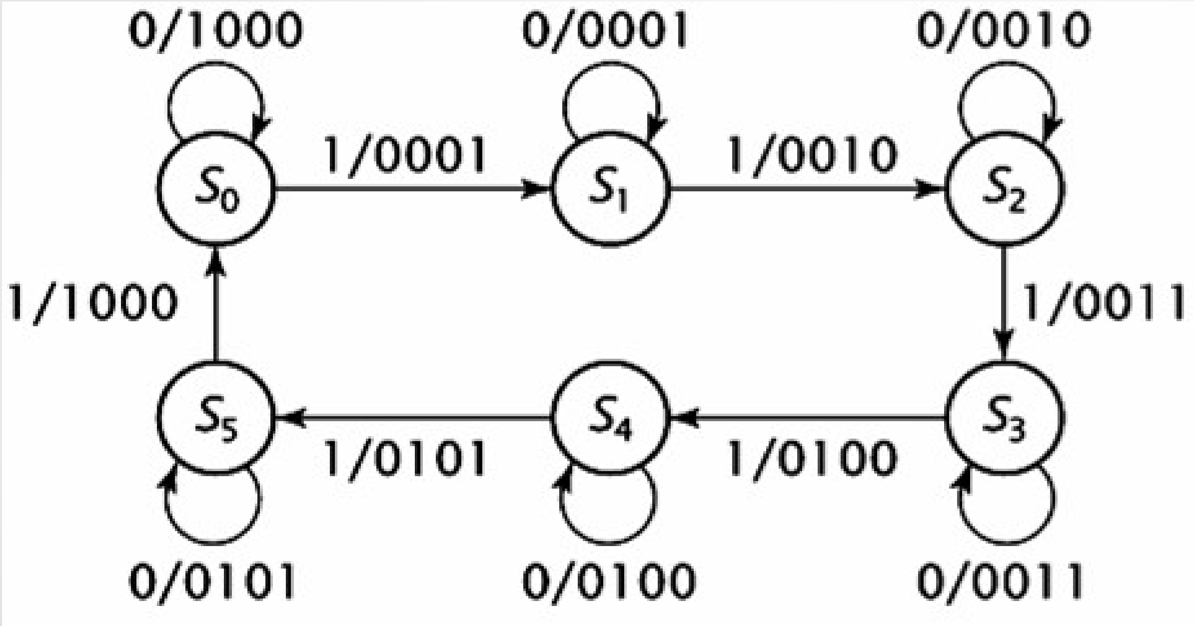
- A modulo 6 counter is a 3-bit counter that counts through the following sequence:
  - 000->001->010->011->100->101->000->…
  - 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 0 …
- It doesn't use value 6 (110) nor 7 (111)
- It has an input U that controls the counter:
  - When U=1 the counter increments its value on the rising edge of the clock
  - When U=0 the counter retains its value on the rising edge of the clock
- The value of the count is represented as three-bit value (V2V1V0)
- There is an additional output **C** (Carry) that is 1 when going from 5 to 0 and 0 otherwise (the C output remains 1 until the counter goes from 0 to 1)

Modulo 6 Counter – State Table

Present State	U	Next State	C	V2V1V0
S0	0	S0	1	000
S0	1	S1	0	001
S1	0	S1	0	001
S1	1	S2	0	010
S2	0	S2	0	010
S2	1	S3	0	011
S3	0	S3	0	011
S3	1	S4	0	100
S4	0	S4	0	100
S4	1	S5	0	101
S5	0	S5	0	101
S5	1	S0	1	000

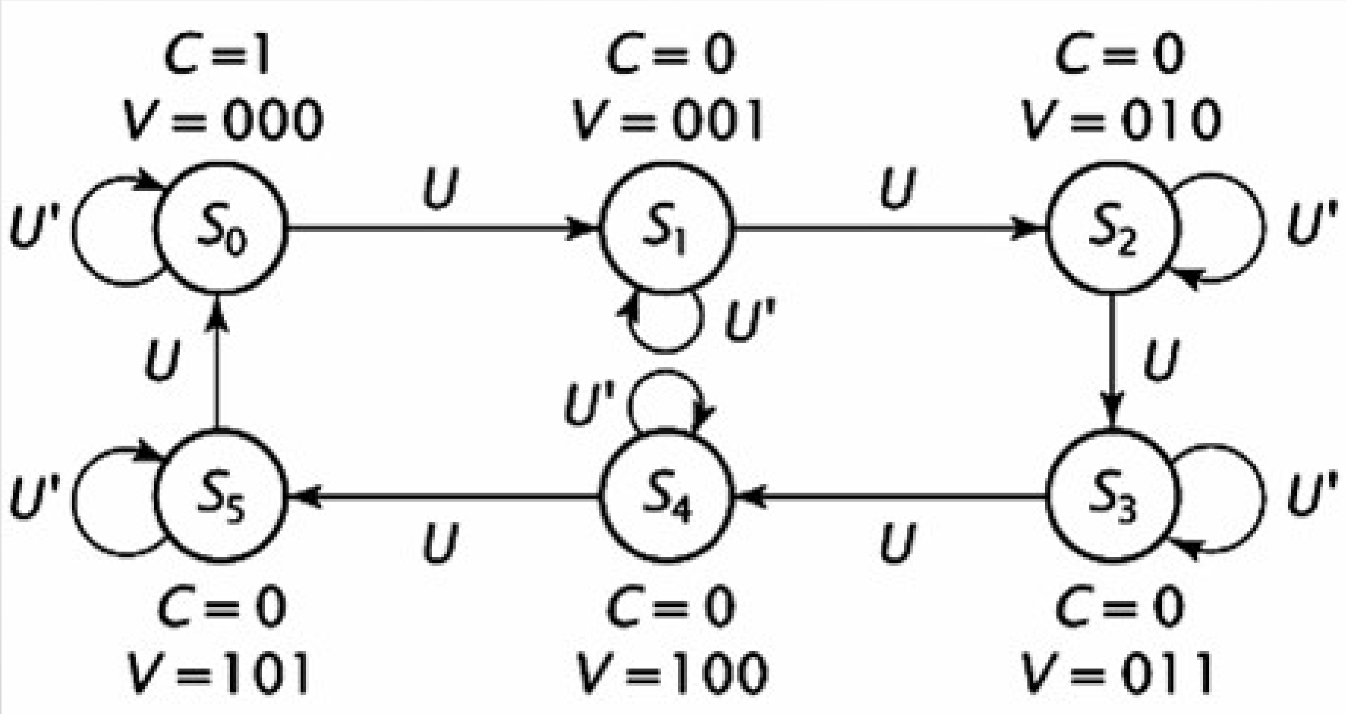
- For each state examine what happens for all possible values of the inputs
  - In state S0 input U can be either 0 or 1
  - If U=0 the state machine remains in state S0 and outputs C=1 and V2V1V0=000
  - If U=1 the state machine goes in state S1, outputs C=0 and V2V1V0=001
- In the same manner, each state goes to the next state if U=1 and remains in the same state if U=0

Modulo 6 Counter - Mealy State Diagram



- The outputs are represented on the arcs as U/CV2V1V0

Modulo 6 Counter – Moore state diagram

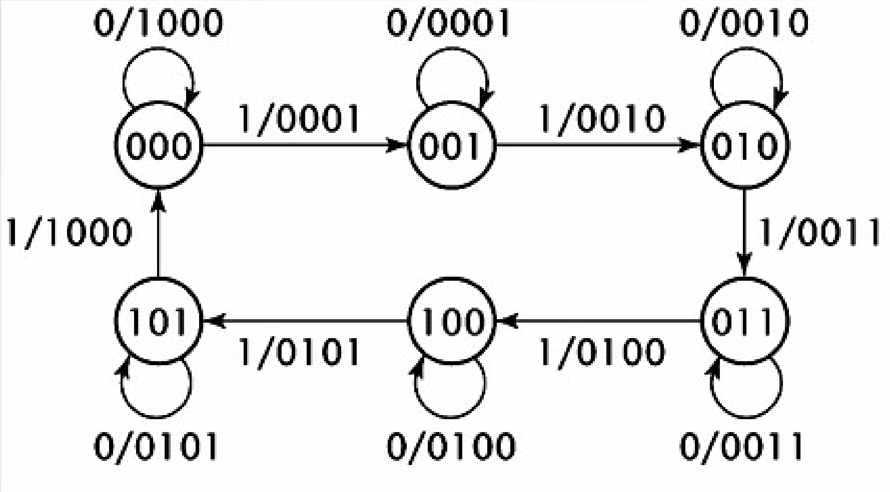


- The outputs are represented adjacent to the state
- The inputs are represented on the arcs

FSM Implementation

- Converting a problem to an equivalent state table and state diagram is just the *first* step in the design process
- The next step is to design the system hardware that implements the state machine.
- This section deals with the process involved to design the digital logic to implement a finite state machine.
- First step is to assign a unique binary value to each of the states that the machine can be in. The state must be encoded in binary.
- Next, we design the hardware to go from the current state to the correct next state. This logic converts the current state and the current input values to the next state values and stores that value.
- The final stage would be to generate the outputs of the state machine. This is done using combinatorial logic.

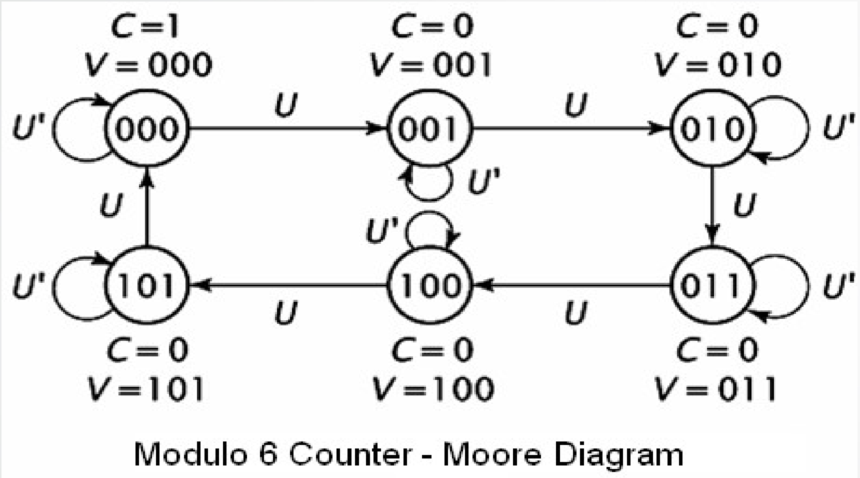
Assigning State Values



Modulo 6 Counter - Mealy Diagram

- Each state must be assigned to a unique binary value; for a machine with n states we have $[\log_2 n]$ bits;
- For the modulo 6 counter example, we have six states. We will assign state value 000 to S0, 001 to S1, and so on, up to 101 to S5.

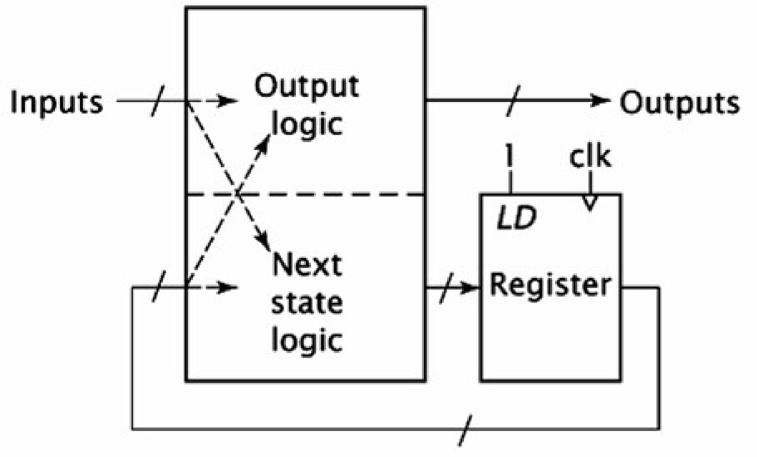
Assigning State Values



- Any values can be assigned to the states, some values can be better than others (in terms of minimizing the logic to create the output and the next state values)
- This is an iterative process: first the designer creates a preliminary design to generate the outputs and the next states, then modifies the state values and repeats the process. There is a rule of thumb, that simplifies the process: whenever possible, the state should be assigned the same with the output values associated with that state. In this case, the same logic can be used to generate the next state and the output

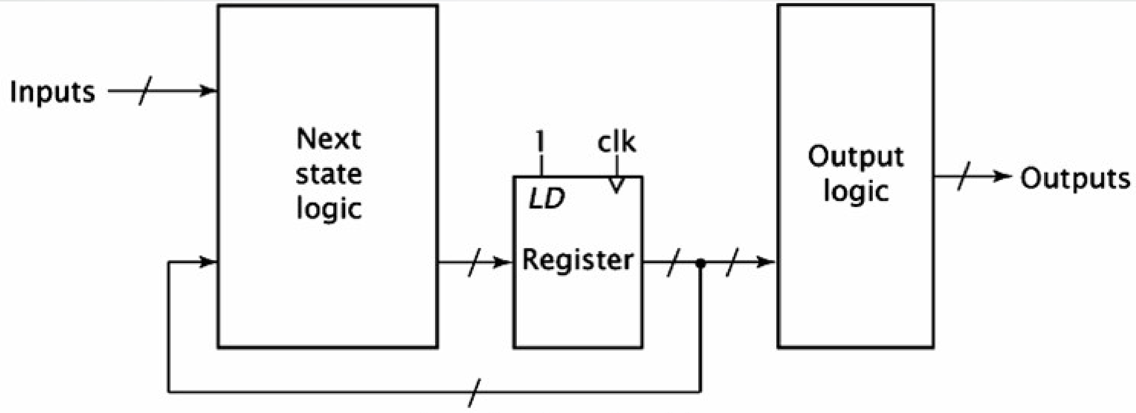
Mealy and Moore Machine Implementations

OÉ Gaillimh
NUI Galway



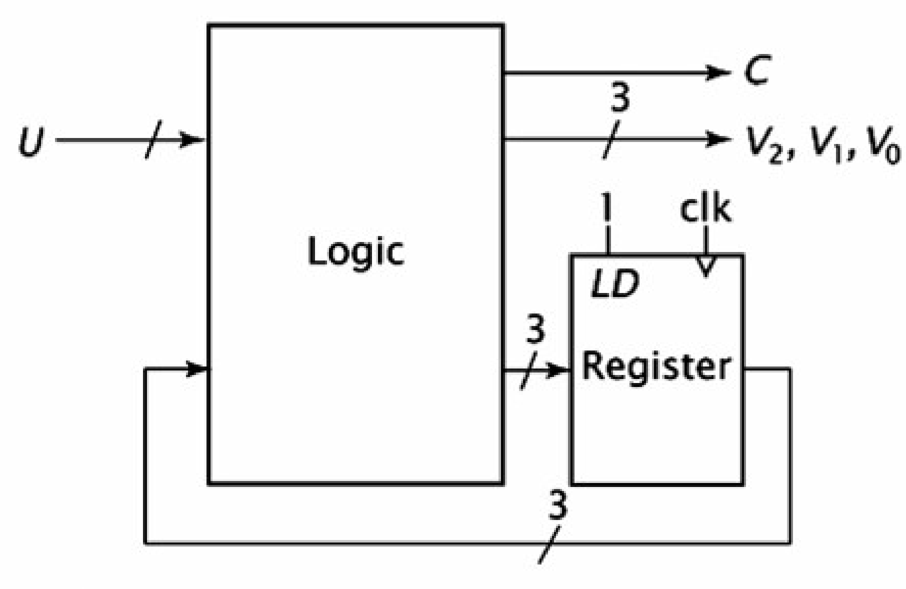
Generic Mealy Machine

- The current state value is stored into the register
- The state value together with the machine inputs, are input to a logic block (CLC) that generates the next state value and machine outputs
- The next state is loaded into the register on the rising edge of the clock signal



Generic Moore Machine

Mod 6 Counter – Mealy Implementation

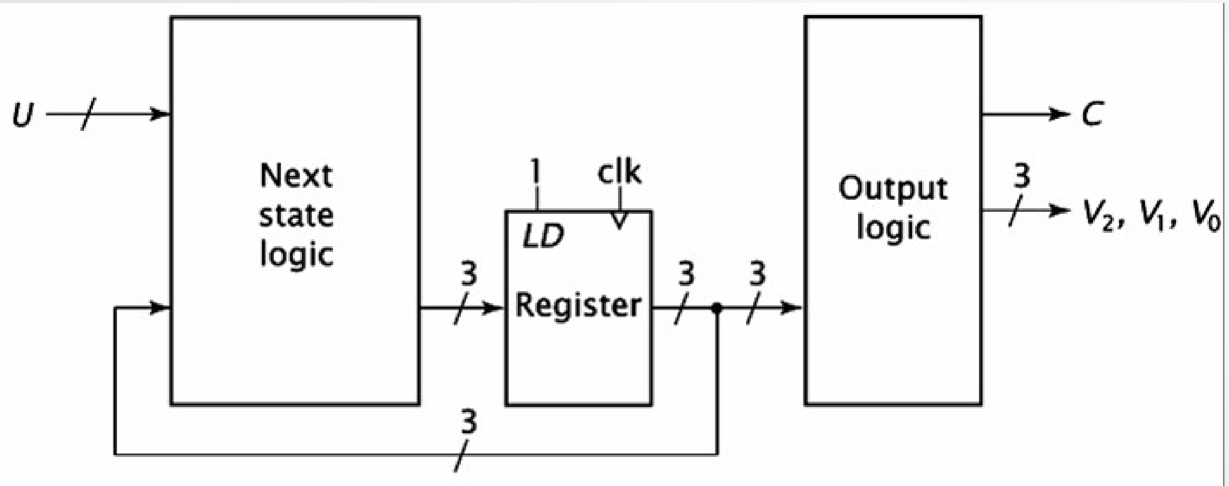


Modulo 6 Counter - Mealy Implementation

- The logic block (CLC) is specific to every system and may consist of combinatorial logic gates, multiplexers, lookup ROMs and other logic components
- The logic block can't include any sequential components, since it must generate its value in one clock cycle
- The logic block contains two parts:
  - One that generates the **outputs** (f function, CLC1)
  - One that generates the **next state** (g function, CLC2)

Mod 6 Counter – Moore Implementation

OÉ Gaillimh
NUI Galway



Modulo 6 Counter - Moore Implementation

- The outputs depend **only** on the present state and not on its inputs
- Its configuration is different than the Mealy machine
  - The system output depends only on the present state, so the implementation of the output logic is done separately
  - The next state is obtained from the input and the present state (same as for the Mealy machine)

Generating the Next State

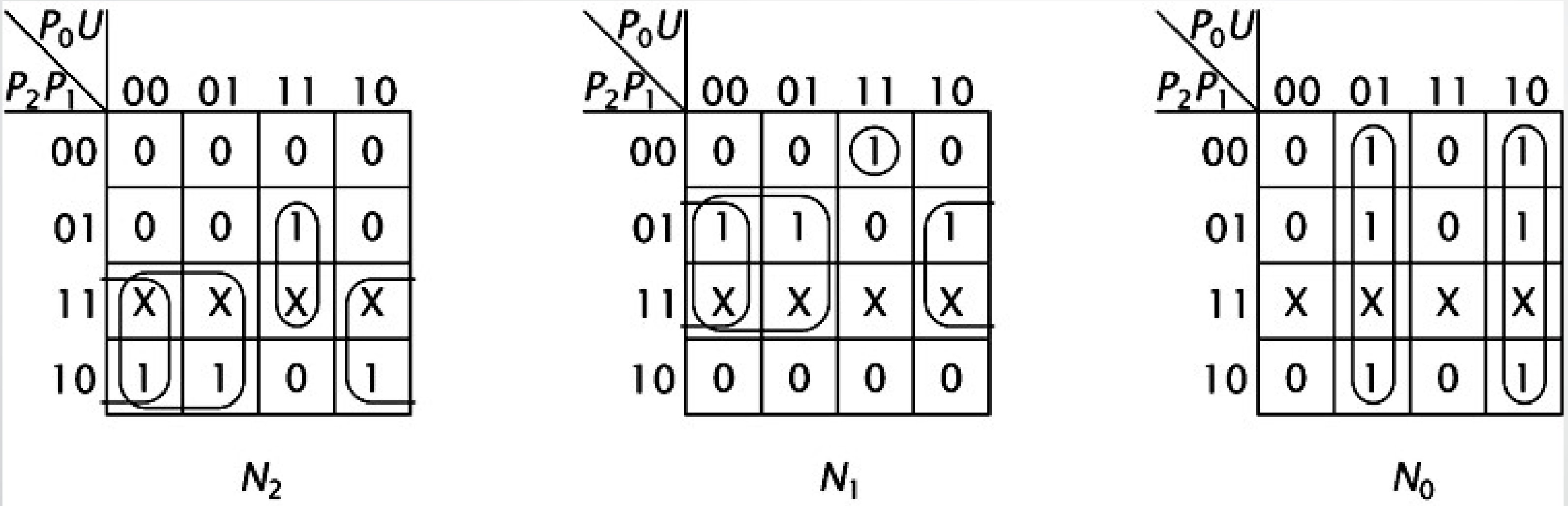
- Since the Mealy and Moore machines must traverse the same states under the same conditions, their next state logic is identical
- We will present three methods to generate the **next state logic**:
  - (i) Combinatorial logic gates
  - (ii) Using multiplexers
  - (iii) Using lookup ROM
- To begin with, we need to setup the truth table for the next state logic

Modulo 6 Counter - Next State Logic (i)

Present State P2P1P0	U	Next State N2N1N0
000	0	000
000	1	001
001	0	001
001	1	010
010	0	010
010	1	011
011	0	011
011	1	100
100	0	100
100	1	101
101	0	101
101	1	000

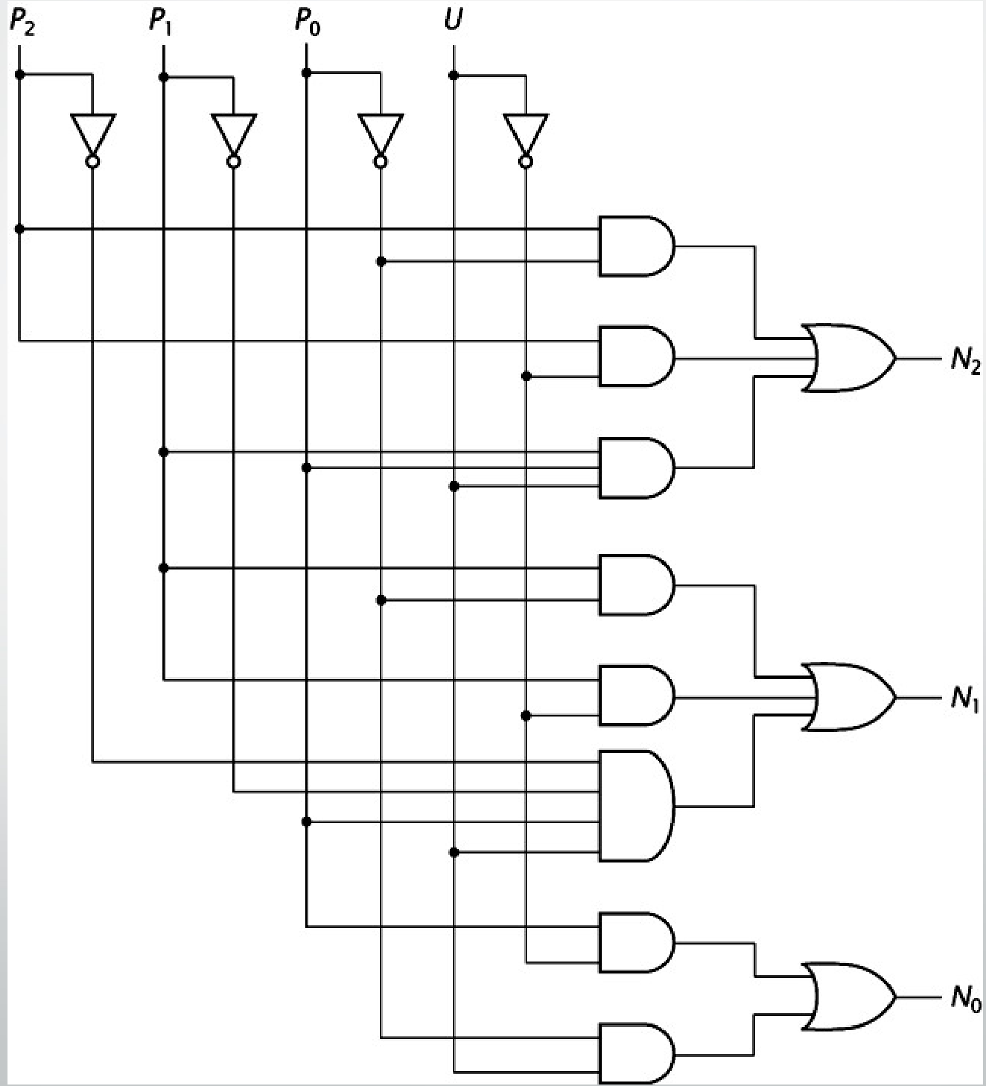
- The system inputs and the present states are the inputs of the truth table
- Next state bits are the outputs
- We have to construct a Karnaugh map for each output bit and obtain its equation
- After that we design the logic to match the equations

Modulo 6 Counter – Next State Logic (i)



- N2 = P2P0’ + P2U’ +P1P0U
- N1 = P1P0’ + P1U’ + P2’P1’P0U
- N0 = P0’U + P0U’

- Modulo 6 Counter – Next State implementation using logic gates (i)



Modulo 6 Counter – Next State Logic (ii)

- An alternative approach to design the next state logic is to use multiplexers.
- Each input to the multiplexer corresponds to the next state under one possible value of the system inputs; the inputs drive the input signals of the multiplexer
- For the modulo 6 counter, we use the U input to drive the multiplexer; U is choosing one of two possible next states, the next state if U=0 and the next state if U = 1
- To determine the inputs of the multiplexer we begin with splitting the truth table into multiple truth tables, one for each possible value of the system inputs
- Then we follow the procedure we have used to obtain the next state using combinatorial logic gate

Modulo 6 Counter – Next State Logic (ii)

OÉ Gaillimh
NUI Galway

Present State P2P1P0	Next State N2N1N0
000	000
001	001
010	010
011	011
100	100
101	101

U = 0

Present State P2P1P0	Next State N2N1N0
000	001
001	010
010	011
011	100
100	101
101	000

U = 1

- Initial truth table is broken into two tables:
  - One for U=0
  - One for U=1
- Create Karnaugh maps from these tables to obtain the equations for N2, N1 and N0 when U=0 and when U=1

Modulo 6 Counter – Next State Logic (ii)

- **U = 0** we observe that the next state is the same as current state:
  - N2 = P2
  - N1 = P1
  - N0 = P0
- **U = 1**:
  - N2 = P2P0'+P1P0
  - N1 = P1P0' + P2'P1'P0
  - N0 = P0'

N2

P2 \ P1P0	00	01	11	10
0	0	0	1	0
1	1	0	x	x

N1

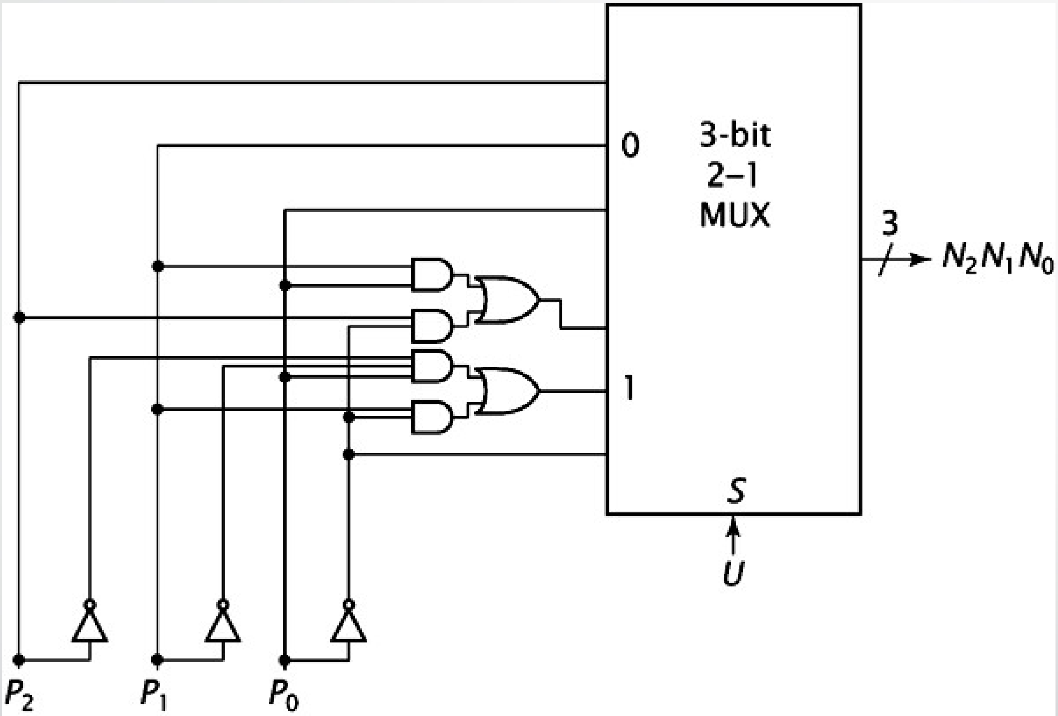
P2 \ P1P0	00	01	11	10
0	0	1	0	1
1	0	0	x	x

N0

P2 \ P1P0	00	01	11	10
0	1	0	0	1
1	1	0	x	x

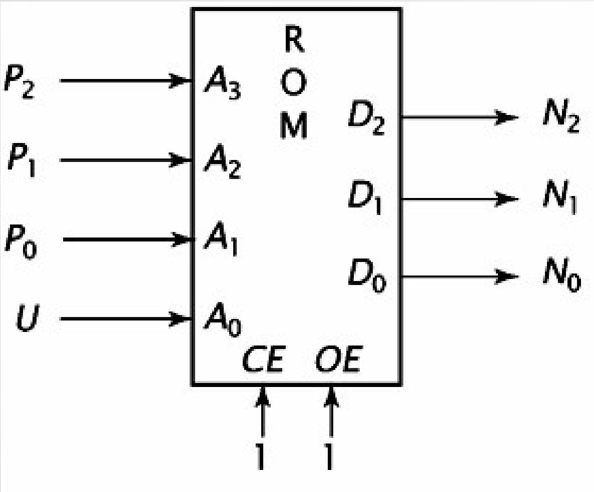
U=1

Modulo 6 Counter – Next State Logic (ii)



Next state logic implementation using multiplexers and logic gates. Please note that using multiplexers simplifies the combinatorial logic circuitry

Modulo 6 Counter – Next State Logic (iii)



- Another approach to generate the next state logic for an FSM is to use a lookup ROM.
- In this approach, the present state values and inputs are connected to the address bus of a ROM; the next state is obtained from the ROM outputs
- The correct value must be stored in each location of the ROM to ensure proper operation

Modulo 6 Counter – Next State Logic (iii)

OÉ Gaillimh
NUI Galway

Address					D_2	D_1	D_0
0	0	0	0	(0)	0	0	0
0	0	0	1	(1)	0	0	1
0	0	1	0	(2)	0	0	1
0	0	1	1	(3)	0	1	0
0	1	0	0	(4)	0	1	0
0	1	0	1	(5)	0	1	1
0	1	1	0	(6)	0	1	1
0	1	1	1	(7)	1	0	0
1	0	0	0	(8)	1	0	0
1	0	0	1	(9)	1	0	1
1	0	1	0	(10)	1	0	1
1	0	1	1	(11)	0	0	0

- The three bits that encode the present state (P2P1P0) are connected to the three high-order address inputs to the ROM (A3A2A1)
- The one condition bit U is connected to the low order address bit A0
- The data in each location is the value of the next state for present state and the input values

- CT101 -

Computing Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Generating System Outputs

- For both Mealy and Moore machines, we follow the same design procedure to develop their output logic
- There are two approaches to generate the output (similar to generate the next state logic):
  - Using combinatorial logic gates
  - Using lookup ROM
- We begin by creating the truth table:
  - For a Mealy machine, the truth table inputs will be the present state **and** the system inputs, and the table outputs are the system outputs
  - For a Moore machine, only the state bits are inputs of the truth table, since only these bits are used to generate the system outputs; the table outputs are the system outputs

Modulo 6 Counter Outputs (i)

OÉ Gaillimh
NUI Galway

P2P1P0	U	C	V2V1V0
000	0	1	000
000	1	0	001
001	0	0	001
001	1	0	010
010	0	0	010
010	1	0	011
011	0	0	011
011	1	0	100
100	0	0	100
100	1	0	101
101	0	0	101
101	1	1	000

Mealy

P2P1P0	C	V2V1V0
000	1	000
001	0	001
010	0	010
011	0	011
100	0	100
101	0	101

Moore

Outputs - Mealy (i)

P2P1 \ P0U	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	X	X	X	X
10	1	1	0	1

V2 = P2P0’ + P1P0U + P2U’

P2P1 \ P0U	00	01	11	10
00	0	0	1	0
01	1	1	0	1
11	X	X	X	X
10	0	0	0	0

V1 = P2P1’P0U + P1P0’ + P1U’

P2P1 \ P0U	00	01	11	10
00	0	1	0	1
01	0	1	0	1
11	X	X	X	X
10	0	1	0	1

V0 = P0’U + P0U’

P2P1 \ P0U	00	01	11	10
00	1	0	0	0
01	0	0	0	0
11	X	X	X	X
10	0	0	1	0

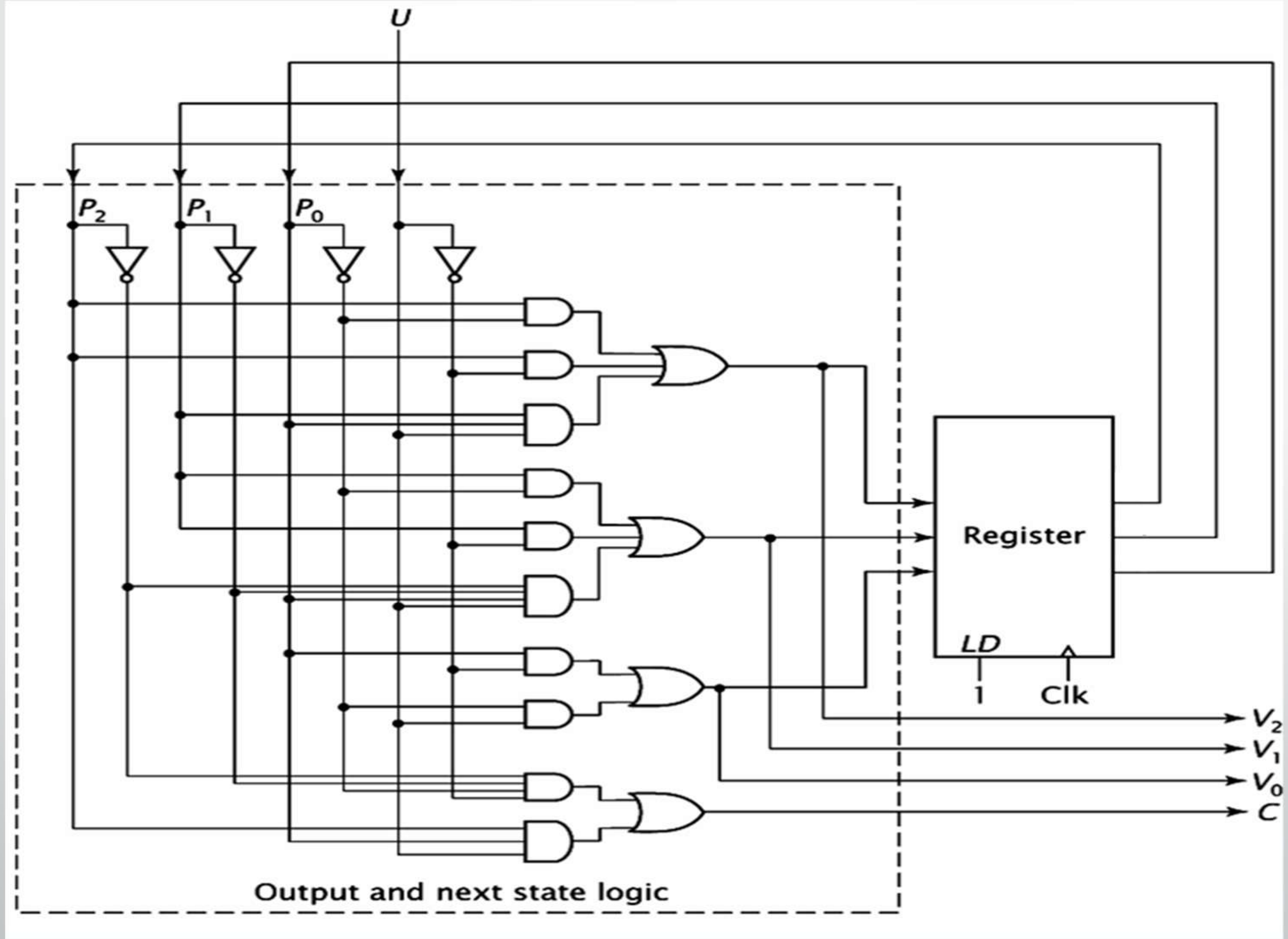
C = P2’P1’P0’U’ + P2P0U

Modulo 6 Counter Outputs (i)

- Mealy machine (note that the equations for V2, V1, V0 are *exactly the same* as for the N2, N1, N0. This is the result of **optimally assigning** the state values. Same combinatorial logic can be used to obtain the outputs):
  - V2 = P2P0'+P2U'+P1P0U
  - V1 = P1P0'+P1U'+P2P1'P0U
  - V0 = P0'U+P0U'
  - C = P2'P1'P0'U'+P2P0U
- Moore machine:
  - V2 = P2
  - V1 = P1
  - V0 = P0
  - C = P2'P1'P0' = (P2+P1+P0)'

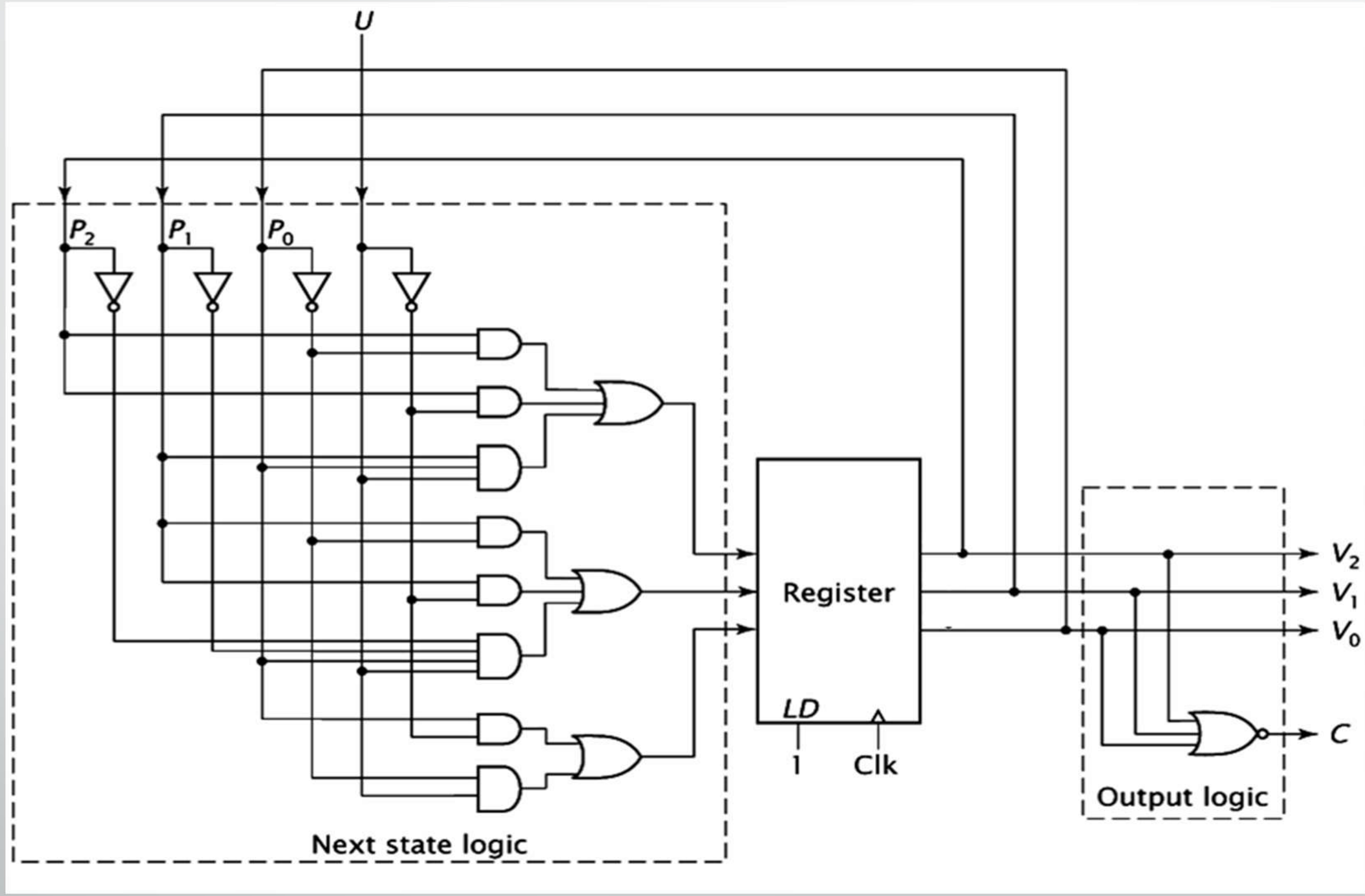
Modulo 6 Counter – Mealy Implementation (i)

OÉ Gaillimh
NUI Galway



Modulo 6 Counter – Moore Implementation (i)

OÉ Gaillimh
NUI Galway

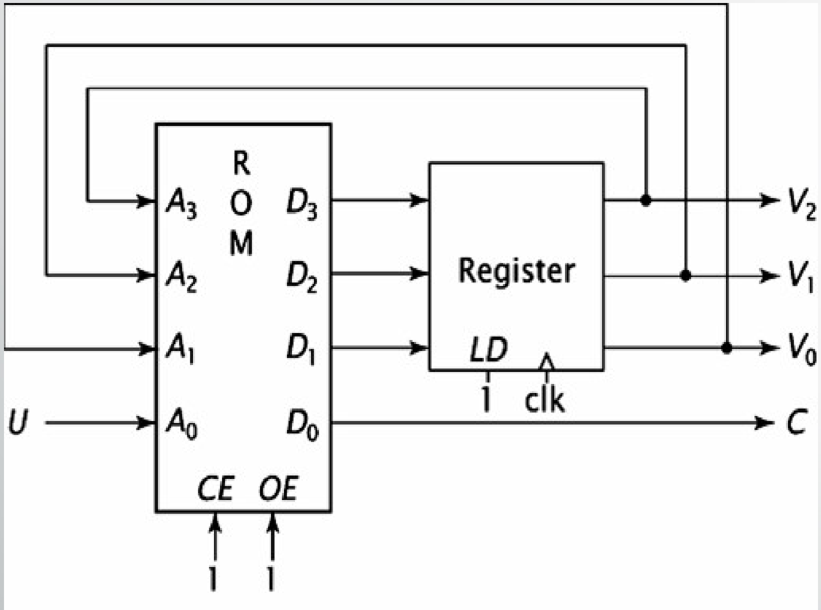


Modulo 6 Counter System Outputs (ii)

- It is possible to generate the system outputs using a lookup ROM
- The inputs of the lookup ROM are the present states and the system inputs. The outputs of the ROM are the system outputs
- We can use same ROM to generate next state and system outputs
- Since for the Mealy machine V2 = N2, V1 = N1 and V0 = N0, only one output is used for each pair. If the outputs weren't the same as the next state, separate output bits would be needed.

Modulo 6 Counter – Moore Implementation (ii)

OÉ Gaillimh
NUI Galway



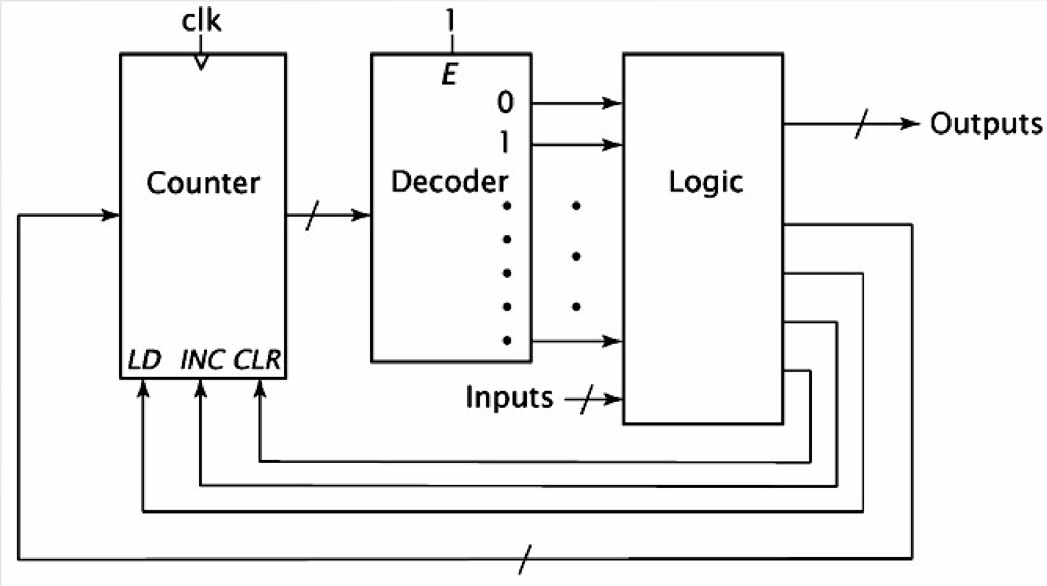
Address					D_3	D_2	D_1	D_0
0	0	0	0	(0)	0	0	0	1
0	0	0	1	(1)	0	0	1	1
0	0	1	0	(2)	0	0	1	0
0	0	1	1	(3)	0	1	0	0
0	1	0	0	(4)	0	1	0	0
0	1	0	1	(5)	0	1	1	0
0	1	1	0	(6)	0	1	1	0
0	1	1	1	(7)	1	0	0	0
1	0	0	0	(8)	1	0	0	0
1	0	0	1	(9)	1	0	1	0
1	0	1	0	(10)	1	0	1	0
1	0	1	1	(11)	0	0	0	0

FSM Alternative Design

OÉ Gaillimh
NUI Galway

- There are some other methods to implement an FSM; one of them is to use a counter to store the current state and a decoder to generate signals corresponding to each state
- The counter can be incremented, cleared or loaded with a value to go from one state to another.
- Unlike the other methods, you don’t have to generate the same state value in order to remain in the same state; this can be accomplished by neither incrementing, clearing nor loading the counter

FSM with Counter and Decoder

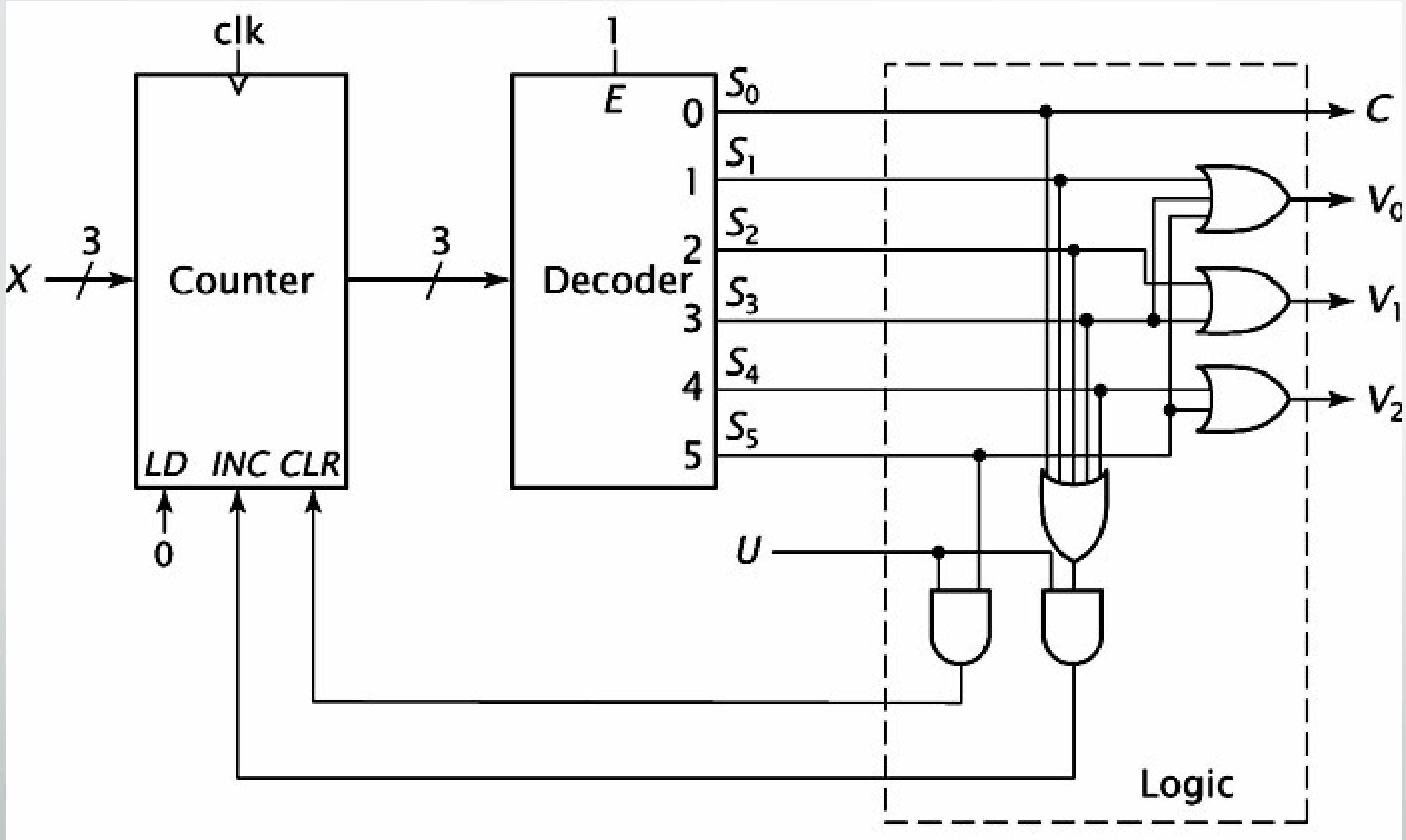


- The counter plays the role of the register in Mealy and Moore designs, as well as a portion of the next state logic
- The state value is input into a decoder; each output of the decoder represents one state
- The decoder outputs and system inputs are input to the logic bloc that generates the system outputs and the information needed to generate the next state value

FSM with Counter and Decoder

- If the system inputs are used to generate **both** the next state and the system outputs, this design can be used to implement a Mealy machine.
- If the system outputs are generated solely by using the state value, and the system inputs are used only to generate the next state, then it implements a Moore machine
- The Modulo 6 counter Moore implementation using this approach is as follows…

Modulo 6 Counter Moore Implementation with Counter and Decoder



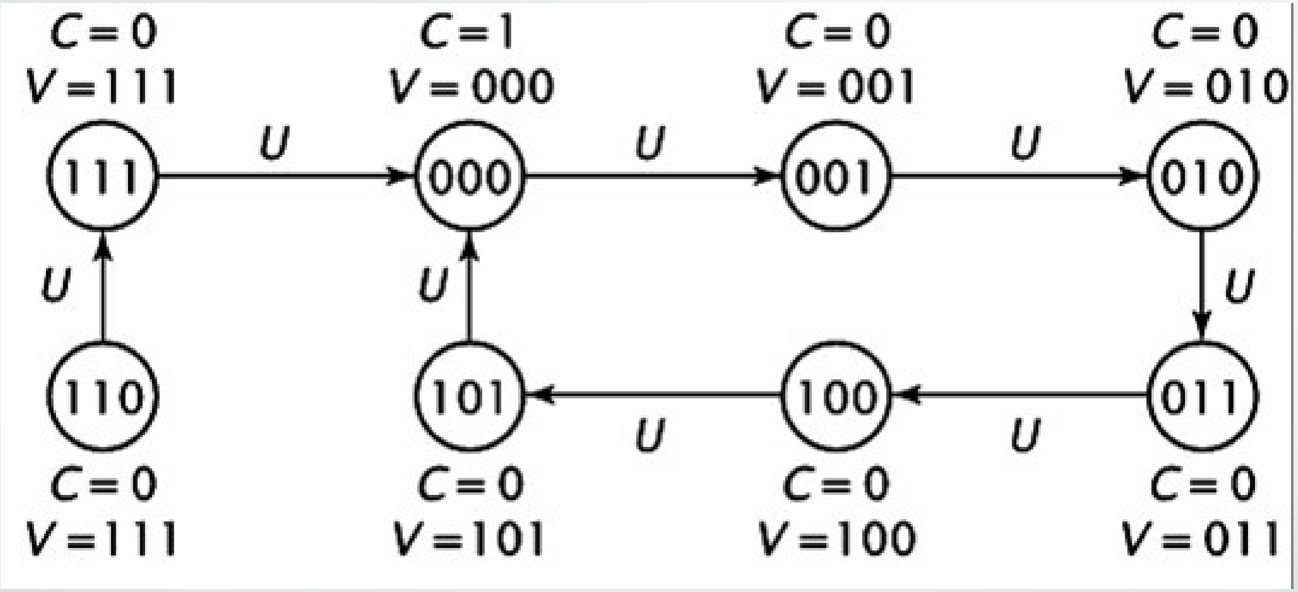
Unused States

- The FSM presented so far works well if it is in a known state
- There will be a problem if the machine enters an **unused** state, also called *unknown* state or *undefined* state
- This could be caused by a flaw in the design but most of the times, this happens when the machine powers-up.

Modulo 6 Counter Analysis

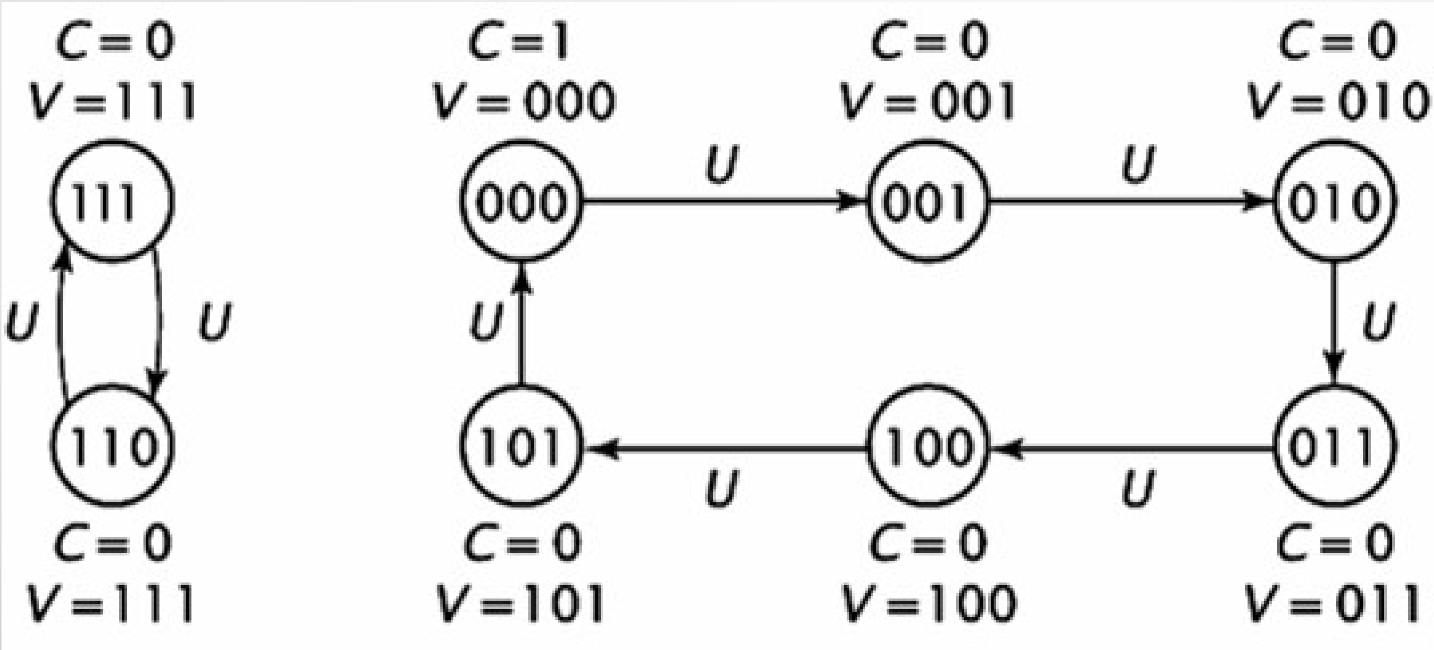
- The modulo 6 counter (consider Moore machine implementation) has six states with binary state values from 000 to 101
- The state value is stored in the register of the finite state machine hardware; an unused state is entered when an unused state is stored in this register;
- The unused states for this design example are 110 and 111

Modulo 6 Counter – Revised (acceptable) diagram



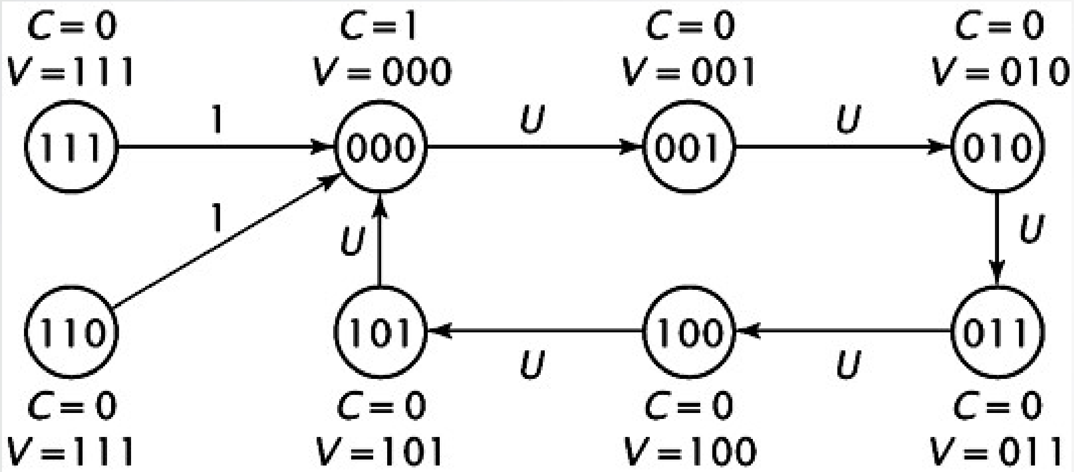
- When present state is 110, the next state is 110 if U=0 or 111 if U=1
- When present state is 111, the next state is 111 if U=0 or 000 if U=1

Modulo 6 Counter – Revised (wrong) Diagram



- If a circuit that implements this diagram powers-up in state 110 or 111 will never reach a valid state

Modulo 6 Counter – State Diagram with Dummy States



- Create dummy states for all unused states
- Each dummy state would go to a known state on the next clock cycle (usually to a reset state)
- Two dummy states: 110 and 111
- By convention, the values 1 on the arcs indicate that the transfer is unconditional – that is always taken
- Note also the output values: C=0 and 111 indicates to the user that the machine is in an invalid state (it is a design decision)

Modulo 6 Counter – State Table with Dummy States

P2P1P0	U	N2N1N0	C	V2V1V0
000	0	000	1	000
000	1	001	0	001
001	0	001	0	001
001	1	010	0	010
010	0	010	0	010
010	1	011	0	011
011	0	011	0	011
011	1	100	0	100
100	0	100	0	100
100	1	101	0	101
101	0	101	0	101
101	1	000	1	000
110	0	000	0	111
110	1	000	0	111
111	0	000	0	111
111	1	000	0	111

- Use this table to construct Karnaugh maps which yield to the following values for next state and outputs:
- Next state:
  - N2 = P2P1'P0' + P2P1'U' + P1P0U
  - N1 = P2'P1P0' + P2'P1U' + P2'P1'P0U
  - N0 = P2'P0'U + P1'P0'U + P1'P0U'
- Outputs:
  - C = P2'P1'P0'
  - V2 = P1
  - V1 = P1
  - V0 = P0 + P2P1

String Checker - Specification

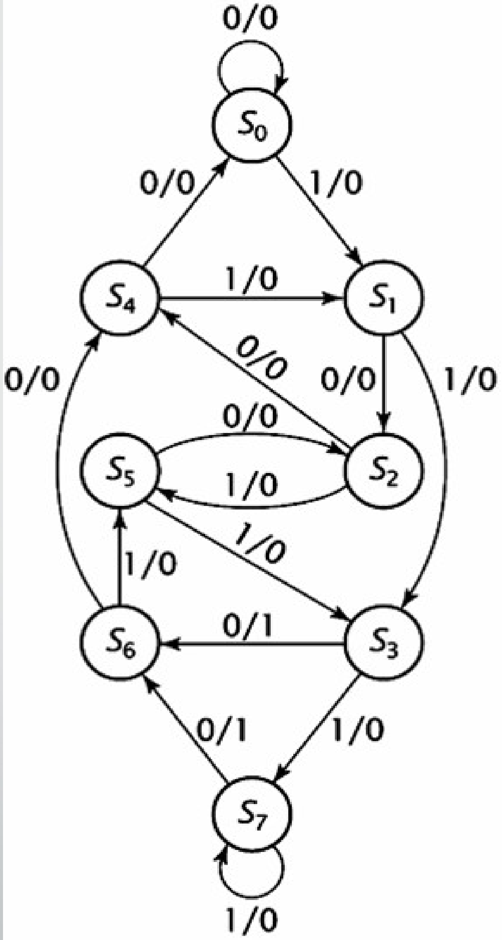
- Inputs a string of bits, one per clock cycle
- When the previous three bits form the pattern 110, it sets the output match M=1; otherwise M=0
- The pattern is checked continuously through the entire bit stream; the system **DOES NOT** check the first three bits and then the next three bits and so on.
- The system checks bits 123 and then bits 234 and then bits 345 and so on.

String Checker – State Table (i)

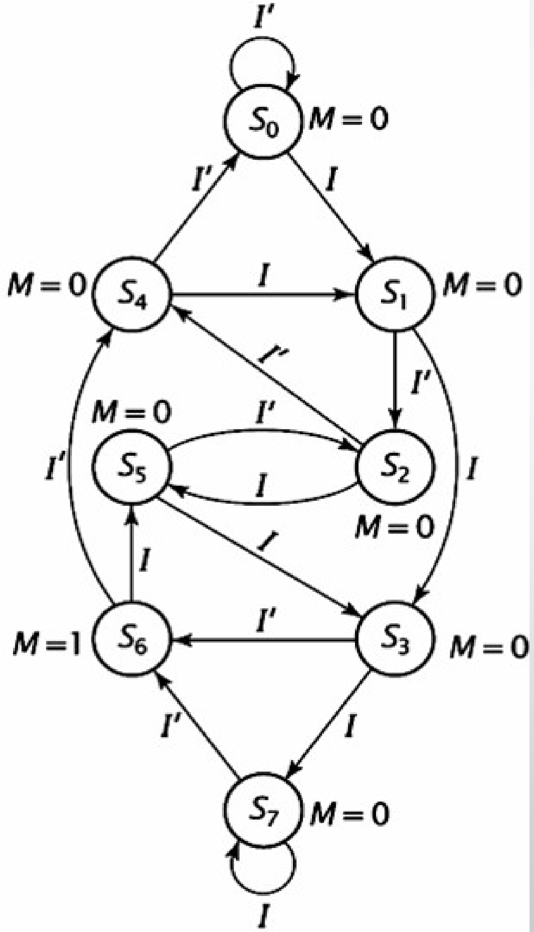
Present State	I	Next State	M
S0 (000)	0	S0	0
S0 (000)	1	S1	0
S1 (001)	0	S2	0
S1 (001)	1	S3	0
S2 (010)	0	S4	0
S2 (010)	1	S5	0
S3 (011)	0	S6	1
S3 (011)	1	S7	0
S4 (100)	0	S0	0
S4 (100)	1	S1	0
S5 (101)	0	S2	0
S5 (101)	1	S3	0
S6 (110)	0	S4	0
S6 (110)	1	S5	0
S7 (111)	0	S6	1
S7 (111)	1	S7	0

- The last three bits received represent the state of the system
- Bits are received from right to left (i.e. the current state is S0 (000), if a new bit with value 1 is received, then the next value of the state is S1 (001)
- Each state goes from one state in two possible next states, depending on the value of I
- Example S2 corresponds to the case where last three bits were 010:
  - I=0 next state is S4 (100), output is M=0
  - I=1 next state is S5 (101), output is M=0

String Checker – State diagrams (i)



Mealy State Diagram



Moore State Diagram

String Checker – Hardware Implementation

- Assign values to the states
  - S0 assign 000 and so on
- Start to design the hardware for this implementation, starting with generic
  - Design the next state logic
  - Design the output logic

P2P1P0	I	N2N1N0
000	0	000
000	1	001
001	0	010
001	1	011
010	0	100
010	1	101
011	0	110
011	1	111
100	0	000
100	1	001
101	0	010
101	1	011
110	0	100
110	1	101
111	0	110
111	1	111

String Checker – Next State Logic

N2

P2P1 \ P0I	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	1	1
10	0	0	0	0

N0

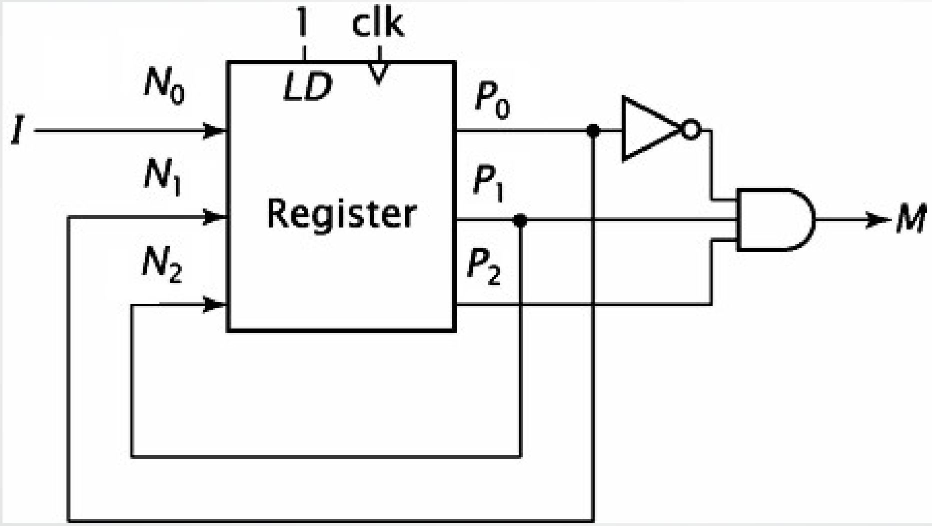
P2P1 \ P0I	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	1	1	0
10	0	1	1	0

N1

P2P1 \ P0I	00	01	11	10
00	0	0	1	1
01	0	0	1	1
11	0	0	1	1
10	0	0	1	1

- N2 = P1
- N1 = P0
- N0 = I

String Checker – Moore Machine



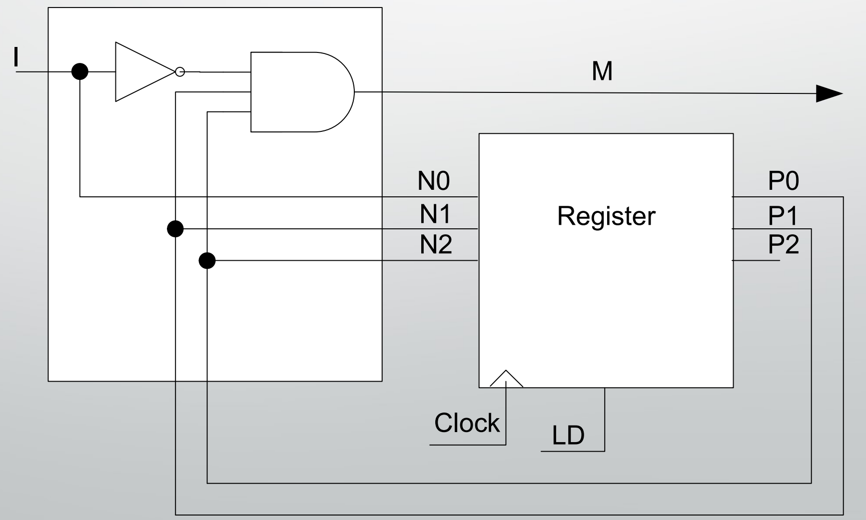
- The output logic is straight forward; when the machine is in state S6 the M is 1, otherwise is 0. This can be implemented as:
  - M = P2P1P0’

String Checker – Mealy Machine

P2P1P0	I	M
000	0	0
000	1	0
001	0	0
001	1	0
010	0	0
010	1	0
011	0	1
011	1	0
100	0	0
100	1	0
101	0	0
101	1	0
110	0	0
110	1	0
111	0	1
111	1	0

M

P2P1 \ P0I	00	01	11	10
00	0	0	0	0
01	0	0	0	1
11	0	0	0	1
10	0	0	0	0



- M = P1P0I’

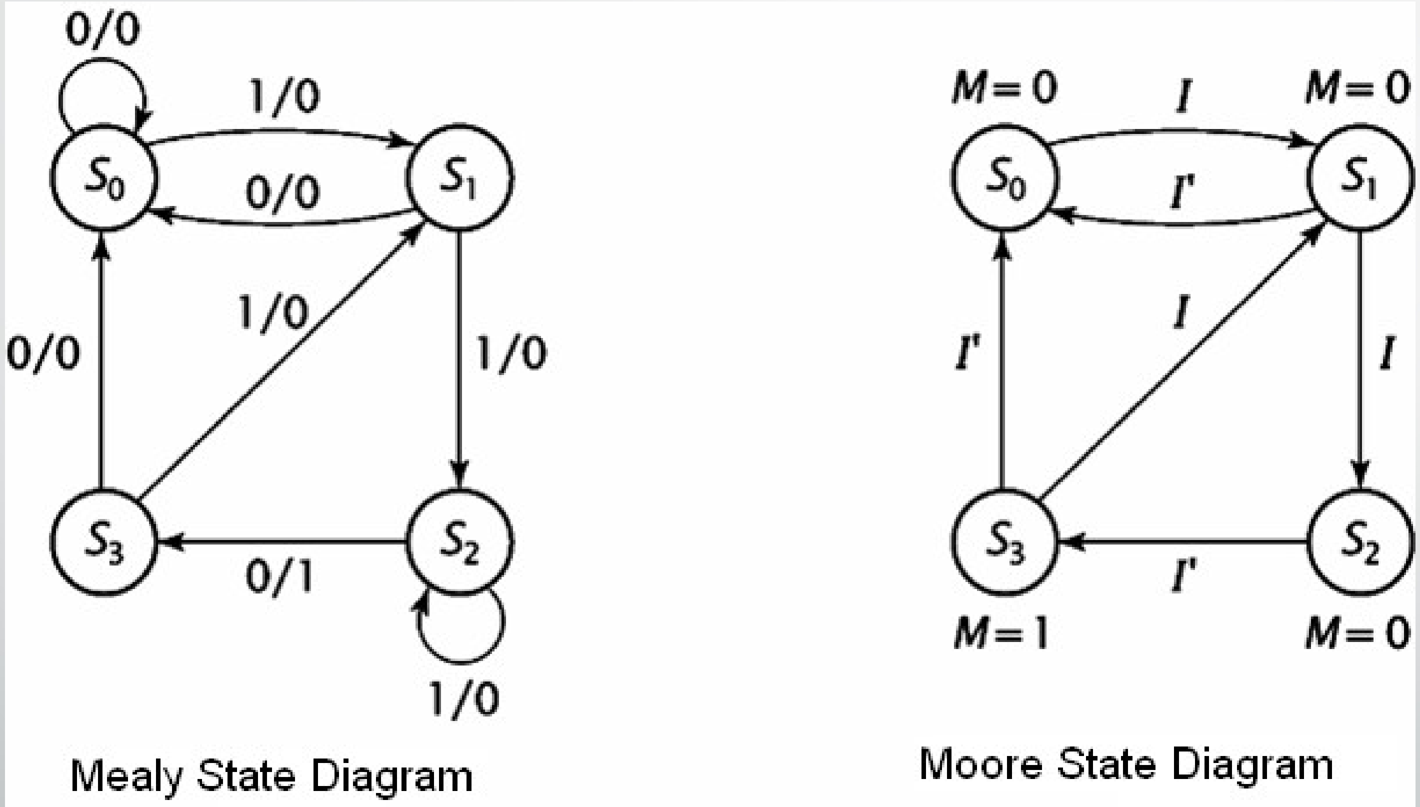
String Checker – State Table (ii)

OÉ Gaillimh
NUI Galway

Present State	I	Next State	M
S0 (---)	0	S0 (---)	0
S0 (---)	1	S1 (--1)	0
S1 (--1)	0	S0 (---)	0
S1 (--1)	1	S2 (-11)	0
S2 (-11)	0	S3 (110)	1
S2 (-11)	1	S2 (-11)	0
S3 (110)	0	S0 (---)	0
S3 (110)	1	S1 (--1)	0

- Sometimes there are simpler alternative methods:
  - S0 – no bits matched
  - S1 – one bit matched
  - S2 – two bits matched
- S3 – three bits matched
- In each state, consider the possible values of the input bit and determine which next state is appropriate

String Checker – State Diagrams (ii)



Toll Booth Controller - Specification

- Has two input sensors:
  - Car sensor C (car in toll booth) = 1 if there is a car or 0 if there is no car
  - Coin sensor (and its value):
    - I1I0 = 00 – no coin has been inserted
    - I1I0 = 01 – a 5 cents coin has been inserted
    - I1I0 = 10 – a 10 cents coin has been inserted
    - I1I0 = 11 – a quarter coin has been inserted
- Two output lights and one alarm output
  - When a car pulls into the toll booth, a red light (R) is lit until the driver deposits at least 35 cents, when the red light goes off and the green light (G) is lit;
  - The green light remains lit until the car leaves the toll booth, when this happen, the red light is lit again
  - If the car leaves the toll booth without paying the full amount, the red light is lit and the alarm (A) sound
  - The alarm remains active until another car pulls into the booth

Toll Booth Controller – States Definition

State	Condition	R	G	A
Snocar	No car in toll booth	1	0	0
S0	Car in toll booth, 0 cents paid	1	0	0
S5	Car in toll booth, 5 cents paid	1	0	0
S10	Car in toll booth, 10 cents paid	1	0	0
S15	Car in toll booth, 15 cents paid	1	0	0
S20	Car in toll booth, 20 cents paid	1	0	0
S25	Car in toll booth, 25 cents paid	1	0	0
S30	Car in toll booth, 30 cents paid	1	0	0
Spaid	Car in toll booth, toll paid	0	1	0
Scheat	Car left toll booth without paying full toll	1	0	1

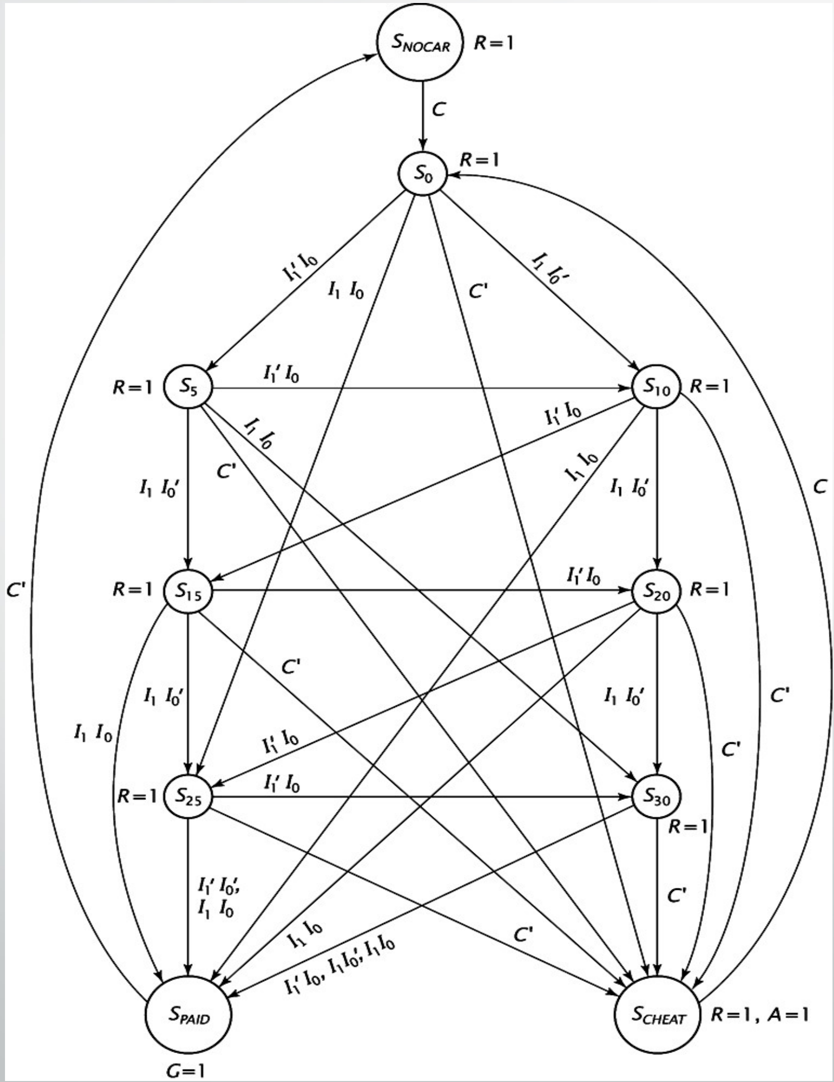
Toll Booth Controller – State table

OÉ Gaillimh NUI Galway

Current State	C	I1I0	Next state	R	G	A
Snocar	0	XX	Snocar	1	0	0
Snocar	1	XX	S0	1	0	0
Spaid	0	XX	Snocar	1	0	0
Spaid	1	XX	Spaid	0	1	0
Scheat	0	XX	Snocar	1	0	1
S0	0	XX	Scheat	1	0	1
S0	1	00	S0	1	0	0
S0	1	01	S5	1	0	0
S0	1	10	S10	1	0	0
S0	1	11	S25	1	0	0
S5	0	XX	Scheat	1	0	1
S5	1	00	S5	1	0	0
S5	1	01	S10	1	0	0
S5	1	10	S15	1	0	0
S5	1	11	S30	1	0	0
S10	0	XX	Scheat	1	0	1
S10	1	00	S10	1	0	0
S10	1	01	S15	1	0	0
S10	1	10	S20	1	0	0
S10	1	11	Spaid	0	1	0

Current State	C	I1I0	Next state	R	G	A
S15	0	XX	Scheat	1	0	1
S15	1	00	S15	1	0	0
S15	1	01	S20	1	0	0
S15	1	10	S25	1	0	0
S15	1	11	Spaid	0	1	0
S20	0	XX	Scheat	1	0	1
S20	1	00	S0	1	0	0
S20	1	01	S25	1	0	0
S20	1	10	S30	1	0	0
S20	1	11	Spaid	0	1	0
S25	0	XX	Scheat	1	0	1
S25	1	00	S25	1	0	0
S25	1	01	S30	1	0	0
S25	1	10	Spaid	0	1	0
S25	1	11	Spaid			
S30	0	XX	Scheat	1	0	1
S30	1	00	S30	1	0	0
S30	1	01	Spaid	0	1	0
S30	1	10	Spaid	0	1	0
S30	1	11	Spaid	0	1	0

Toll Booth Controller – Moore State Diagram



References

- “Computer Systems Organization & Architecture”, John D. Carpinelli, ISBN: 0-201-61253-4

- CT101 -
Computing Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

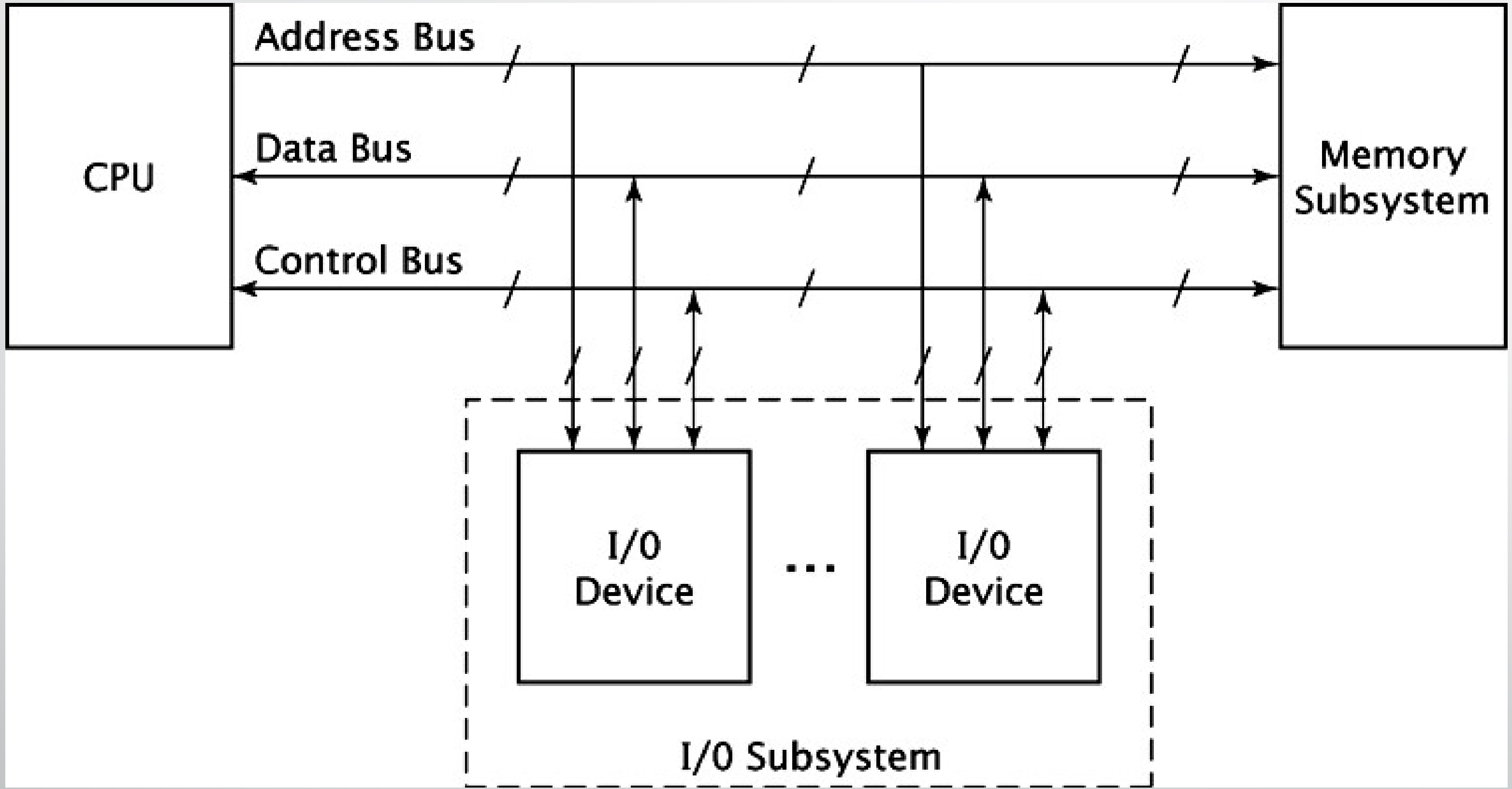
CPU Programming Models

Contents

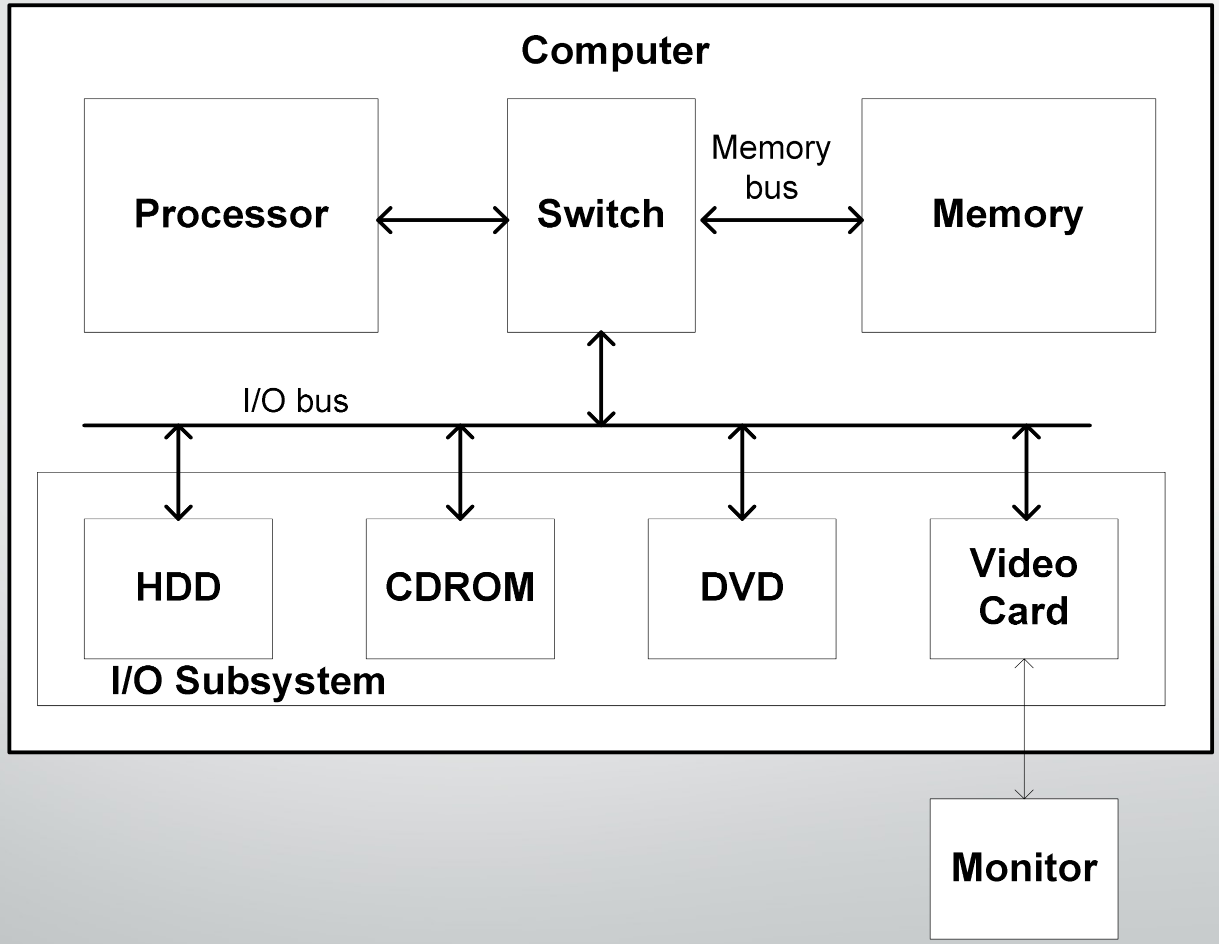
OÉ Gaillimh
NUI Galway

- Review of computer organization
- Processor instruction cycle and organization
- Stack and GPR processor architectures
- Stack used to implement procedure calls

Basic Computer Organization



Computer Organization



Von Neumann and Harvard Architectures

- Von Neumann:
  - Allows instructions and data to be mixed and stored in the same memory module
  - More flexible and easier to implement
  - Suitable for most of the general-purpose processors
- Harvard:
  - Uses separate memory modules for instructions and for data
  - It is easier to pipeline
  - Higher memory throughput
  - Suitable for DSP (Digital Signal Processors)

Programs

- Instruction sequences that tell computer what to do

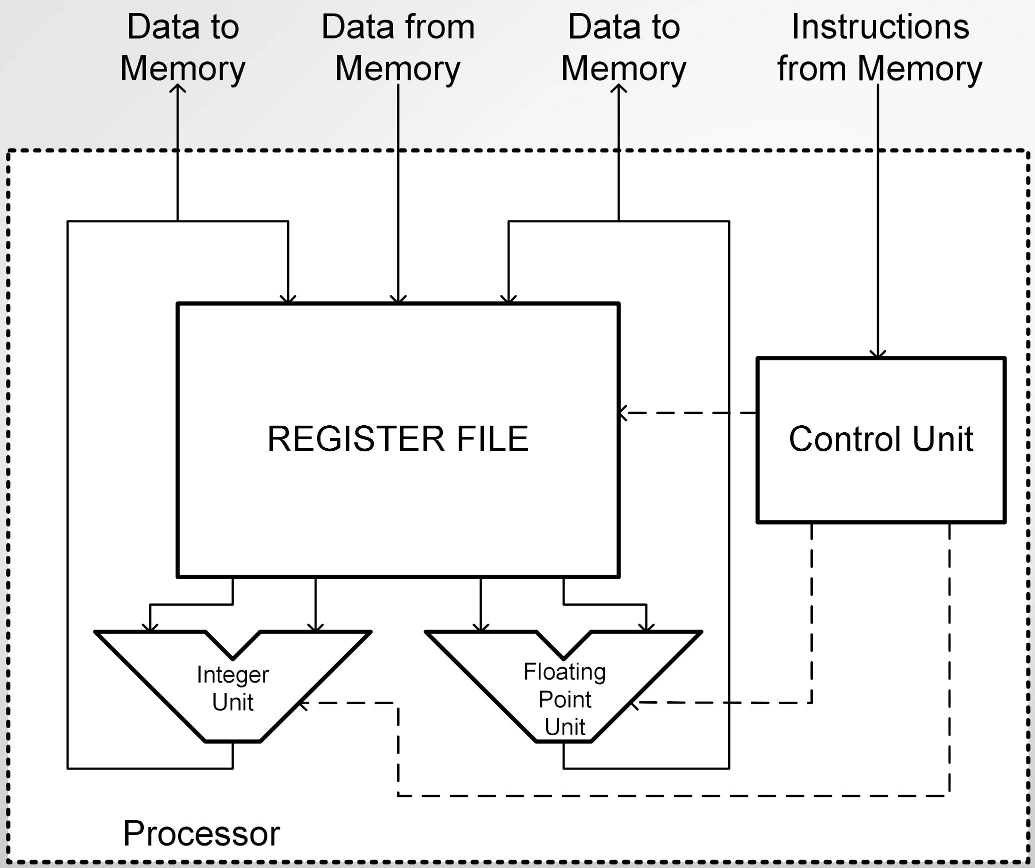
- To the computer, a program is made out of a sequence of numbers that represent individual operations.
  - Those operations are known as ***machine instructions*** or just ***instructions***
  - A set of instructions that a processor can execute is known as ***instruction set***

The Processor - Instruction Cycles

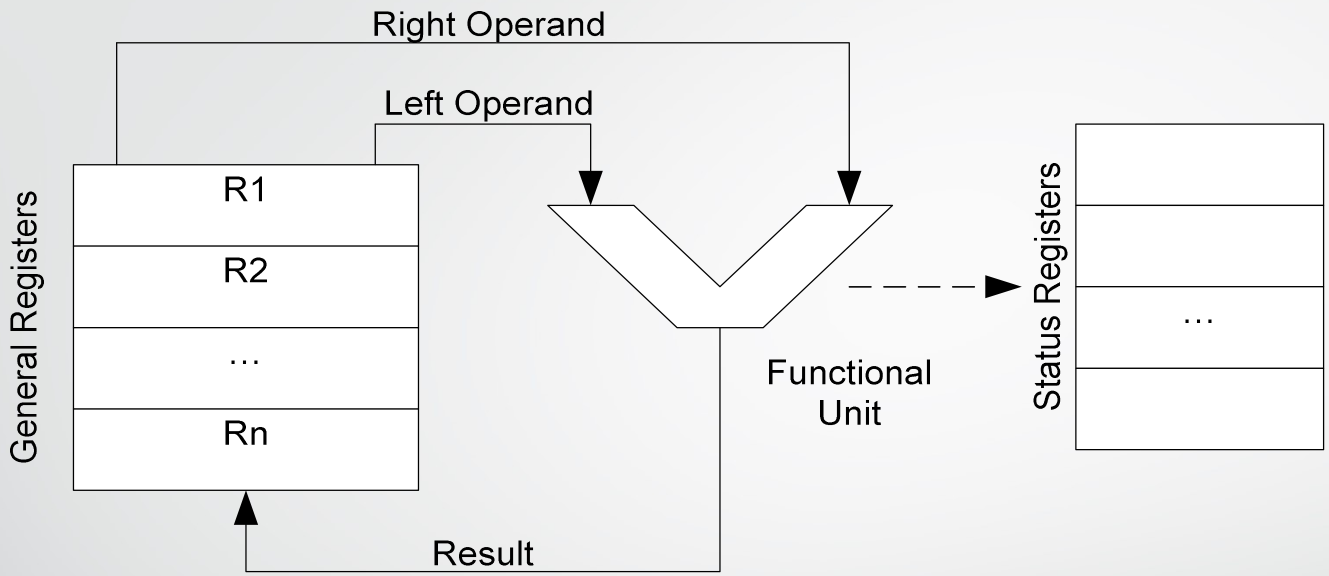
- The instruction cycle is the procedure of processing an instruction by the microprocessor:
  - Fetches or reads the instruction from the memory
  - Decodes the instruction, determining which instruction is to be executed (which instruction has been fetched)
  - Executes the instruction – performs the operations necessary to complete what the instruction is suppose to do (different from instruction to instruction, may read data from memory, may write data to memory or I/O device, perform only operations within CPU or combination of those)

- Each of the phases fetch -> decode -> execute consist of a sequence of one or more operations inside the CPU (and interaction with the subsystems)

Processor Organization



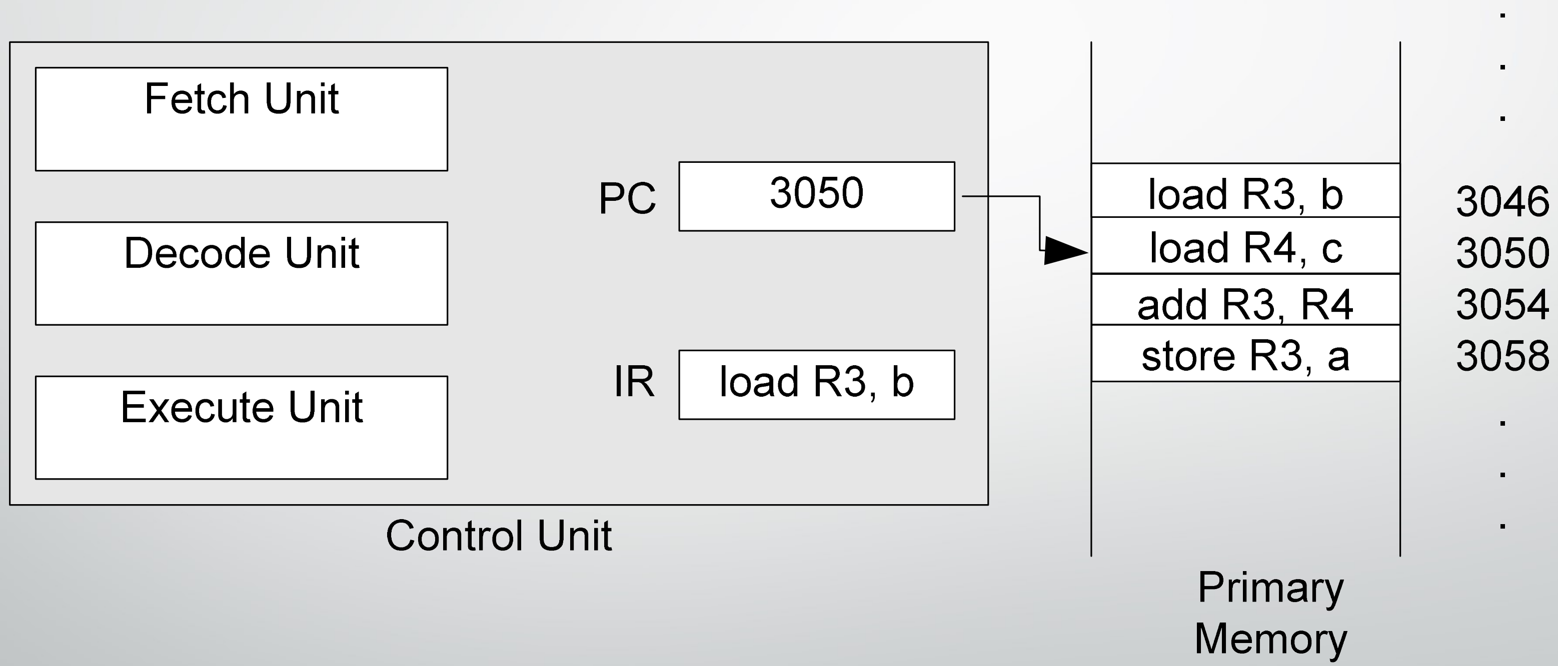
Execution Unit Example



- //Code for a = b + c
- LD R3, b //copy value b from memory to R3
- LD R4, c //copy value c from memory to R4
- add R3, R4 //sum placed in R3
- ST R3, a //store the result into memory

Control Unit

- The control unit controls the execution of the instructions stored in the main memory (retrieve and execute them)



Programming Models

OÉ Gaillimh
NUI Galway

- A processor programming model defines how instructions access their operands and how instructions are described in processor's assembly language
- Processors with different programming models can offer similar set of operations but may require very different approaches to programming
- We will study two different processor architectures and will learn what differences in programming for the stack vs GPR models

Stack Based Architectures

- The Stack
- Implementing Stacks
- Stack based architecture instruction set
- Programs in stack-based architecture

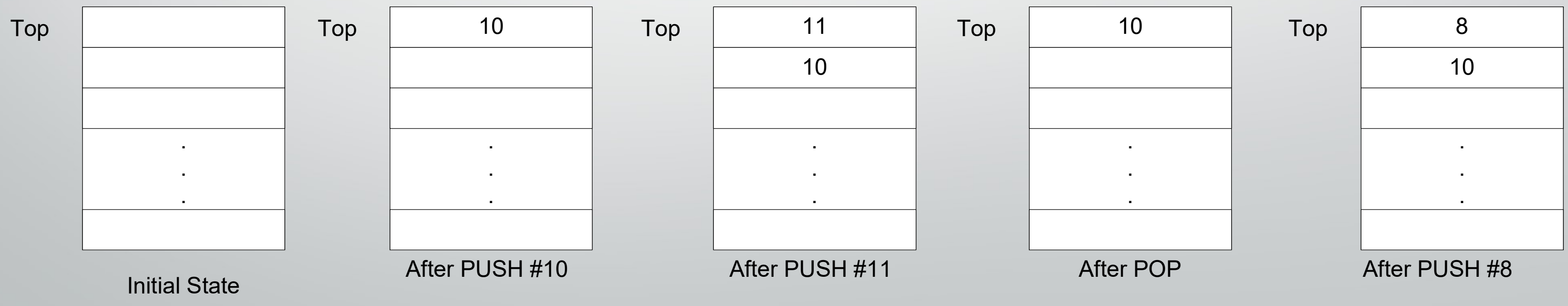
The Stack (1)

- Is a last in first out (LIFO) data structure
  - Consists of locations, each of which can hold a word of data
  - It can be used explicitly to save/restore data
- Supports two operations
  - **PUSH** – takes one argument and places the value of the argument in the top of the stack
  - **POP** – removes one element from the stack, saving it into a predefined register of the processor
- It is used implicitly by procedure call instructions (if available in the instruction set)

The Stack (2)

- When a new data is added to the stack, it is placed at the top of the stack, and all the contents of the stack is pushed down one location

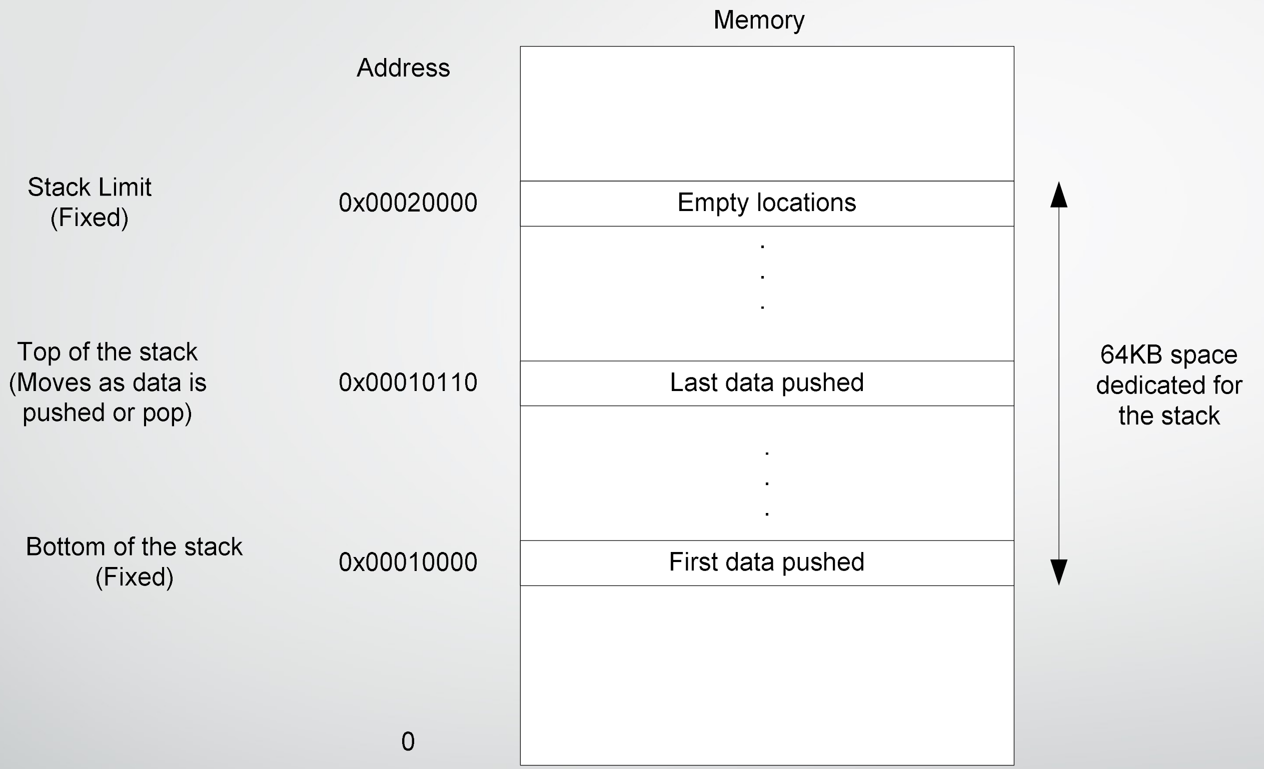
- Consider the code:
  - PUSH #10
  - PUSH #11
  - POP
  - PUSH #8



Implementing Stacks

- Dedicated hardware stack
  - It has a hardware limitation
  - Very fast
- Memory implemented stack
  - Limited by the physical memory of the system
  - Slow compared with hardware stack, since extra memory addressing has to take place for each stack operation
- ***Stack overflows*** can occur in both implementations
  - When the amount of data in the stack exceeds the amount of space allocated to the stack (or the hardware limit of the stack)

Stack Implemented in Memory



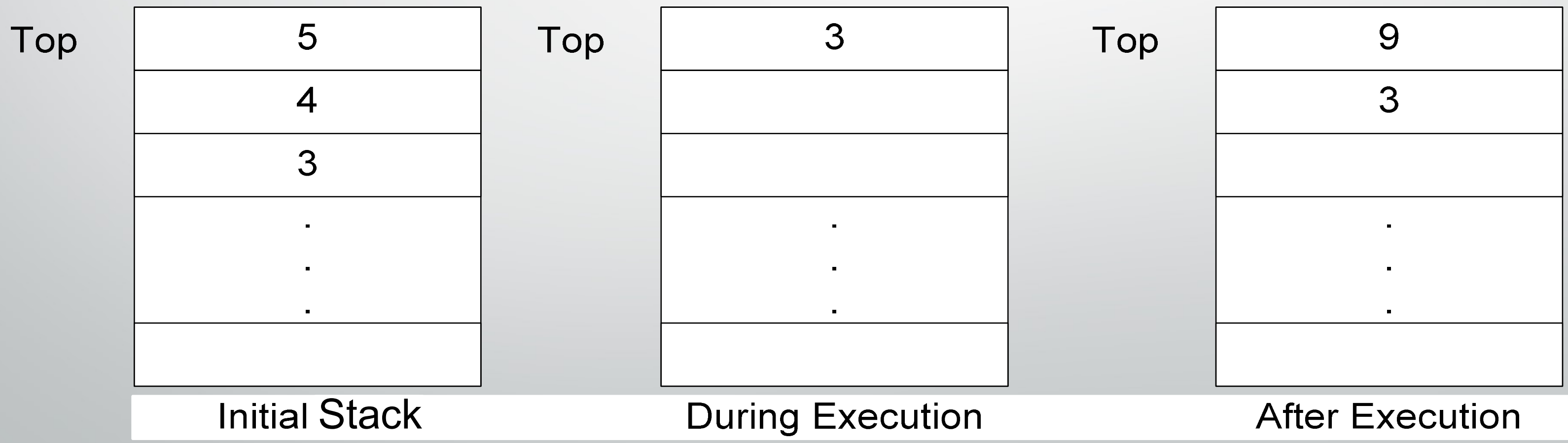
- Every push operation will increment the top of the stack pointer (with the word size of the machine); Every pop operation will decrement the top of the stack pointer

School of Computer Science

Instructions in a Stack Based Architecture

- Get their operands from the stack and write their results to the stack
- Advantage - Program code takes little memory (no need to specify the address of the operands in memory or registers)
  - Push is one exception, because it needs to specify the operand (either as constant or address)

ADD Instruction Execution



Simple Stack Based Instruction Set

PUSH #a	**Stack <-a**
POP	**a<-Stack (the value popped is discarded)**
ST	**a <-Stack** **(a) <-Stack**
LD	**a <-Stack** **Stack <- (a)**
ADD	**a <- Stack** **b <- Stack** **Stack <- a + b**
SUB	**a <- Stack** **b <- Stack** **Stack <- b – a**
AND	**a <- Stack** **b <- Stack** **Stack <- a & b (bit wise computation)**
OR	**a <- Stack** **b <- Stack** **Stack <- a | b (bit wise computation)**

Programs in Stack Based Architecture (1)

- Writing programs for stack-based architectures is not easy, since stack-based processors are better suited for postfix notation rather than infix notation
  - **Infix** notation is the traditional way of representing math expressions, with operation placed between operands
  - **Postfix** notation – the operation is placed after the operands

- Once the expression has been converted into postfix notation, implementing it in programs is easy

- Create a stack-based program that computes:
  - 2 + (7&3)

Programs in Stack Based Architecture (2)

- First, we need to convert the expression into postfix notation:
  - 2 + (7&3) = 2 + (7 3 &) = (2 (7 3 &)+)
- Convert the postfix notation into a series of instructions, using the instructions from the instruction set presented earlier
    - PUSH #2
    - PUSH #7
    - PUSH #3
    - AND
    - ADD
- To verify the result, we need to hand simulate the execution

- CT101 -
Computing Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

General Purpose Register Architecture

- Instructions in a GPR architecture
- A GPR instruction set
- Programs in GPR architecture

General Purpose Register Architecture (1)

- The instructions read their operands and write their results to random access register file.
- The general-purpose register file allows the access of any register in any order by specifying the number (register ID) of the register
- The main difference between a general-purpose register and the stack is that reading repeatedly a register will produce the same result and will not modify the state of the register file.
  - Popping an item from a LIFO structure (stack) will modify the contents of the stack

General Purpose Register Architecture (2)

Register File

Register 0	data
Register 1	data
Register 2	data
	. . .
Register n	data

- Many GPR architectures assign special values to some registers in the register file to make programming easier
  - i.e. sometimes, register 0 is hardwired with value 0 to generate this most common constant

Instructions in GPR Architecture (1)

- GPR instructions need to specify the register that hold their input operands and the register that will hold the result
- The most common format is the three operands instruction format.
  - ADD r1, r2, r3 instructs the processor to read the contents of r2 and r3, add them together and write the result in r1
- Instructions having two or one input are also present in GPR architecture

Instructions in GPR Architecture (2)

- A significant difference between GPR architecture and stack-based architecture is that programs can choose which values should be stored in the register file at any given time, allowing them to cache most accessed data
  - In stack-based architectures, once the data has been used, it is gone.
- GP architectures have better performance from this point of view, at the expense of needing more storing space for the program (since the instructions are larger, needing to encode also addresses of the operands)

Simple GPR Instruction Set

ST (ra), rb	**(ra) <- rb**
LD ra, (rb)	**ra <- (rb)**
ADD ra, rb, rc	**ra <- rb +rc**
SUB ra, rb, rc	**ra <- rb -rc**
AND ra, rb, rc	**ra <- rb & rc**
OR ra, rb, rc	**ra <- rb | rc**
MOV ra, rb	**ra <- rb**
MOV ra, #constant	**ra <- constant**

Sample instruction set, similar with the one presented for the Stack-based architecture.

Programs in a GPR Architecture (1)

OÉ Gaillimh
NUI Galway

- Programming a GPR architecture processor is less structured than programming a stack-based architecture one.

- There are fewer restrictions on the order in which the operations can be executed
  - On stack-based architectures, instructions should execute in the order that would leave the operands for the next instructions on the top of the stack
  - On GPR, any order that places the operands for the next instruction in the register file before that instruction executes is valid.
    - Operations that access different registers can be reordered without making the program invalid

Programs in GPR Architecture (2)

OÉ Gaillimh
NUI Galway

- Create a GPR based program that computes:
  - 2 + (7&3)
- GPR programming uses infix notation:
  - MOV R1, #7
  - MOV R2, #3
  - AND R3, R1, R2
  - MOV R4, #2
  - ADD R4, R3, R4
- The result will be placed in R4

Comparing Stack based and GPR Architectures

- Stack-based architectures
  - Instructions take fewer bits to encode
  - Reduced amount of memory taken up by programs
  - Manages the use of register automatically (no need for programmer intervention)
  - Instruction set does not change if size of register file has changed

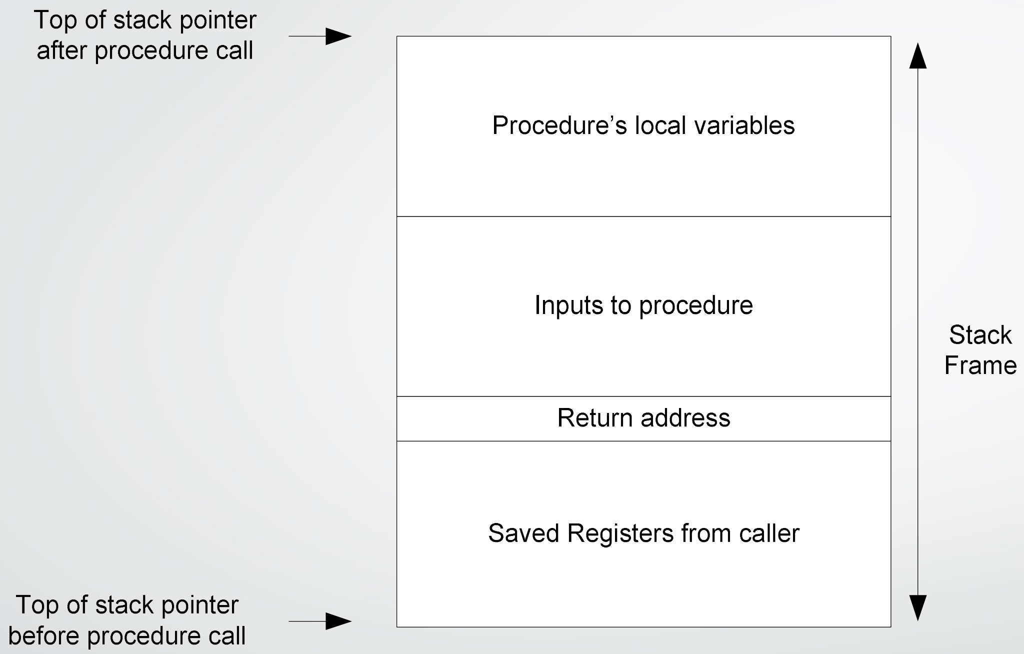
- GPR architectures
  - With evolution of technology, the amount of space taken up by a program is less important
  - Compilers for GPR architectures achieve better performance with a given number of general purpose registers than they are on stack-based architectures with same number of registers
    - The compiler can choose which values to keep (cache) in register file at any time

- GPR architectures are used by modern computers (workstations, PCs, etc..)

Using Stacks to Implement Procedure Calls (1)

- Programs need a way to pass inputs to the procedures that they call and to receive outputs back from them
- Procedures need to be able to allocate space in memory for local variables, without overriding any data used by their calling program
- It is impossible to determine which registers may be safely used by the procedure (especially if the procedure is located in a library), so a mechanism to save/restore registers of calling program has to be in place
- Procedures need a way to figure out where they were called from, so the execution can return to the calling program when the procedure completes (they need to restore the program counter)

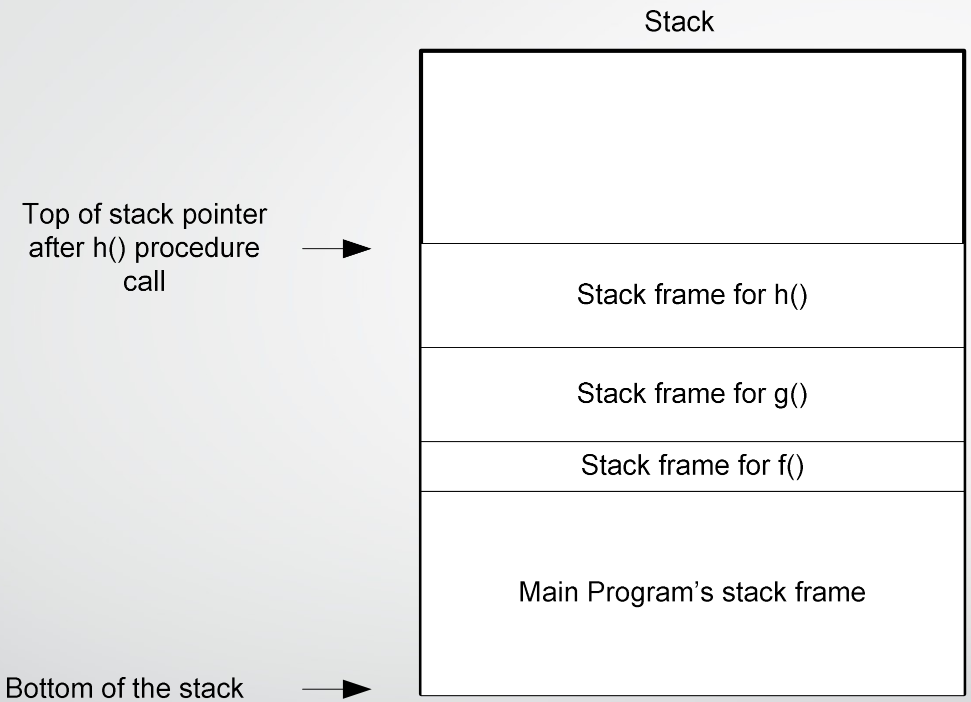
Using Stacks to Implement Procedure Calls (2)



- When a procedure is called, a block of memory in the stack is allocated. This is called a stack frame
  - The top of the stack pointer is incremented by the number of locations in the stack frame

Using Stacks to Implement Procedure Calls (3)

OÉ Gaillimh
NUI Galway



- Nested procedure calls – main program calls function f(), function f() calls function g(), function g() calls function h()

References

- "Computer Systems Organization & Architecture", John D. Carpinelli, ISBN: 0-201-61253-4

OÉ Gaillimh
NUI Galway

- CT101 -
Computing Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Instruction Set Architecture

- Instruction Set Architecture
- Programming languages
- Instruction types
- Data types
- Instruction formats
- Addressing modes
- Instruction set design

Instruction Set Architecture

- Includes the microprocessor's instruction set, the set of all the assembly language instructions that the microprocessor can execute

- Specifies:
  - The registers accessible to the programmer, their size and the instructions in the instructions set that can use each register
  - Information necessary to interact with the memory (e.g. alignment)
  - How microprocessors react to interrupts (e.g. interrupt routines)

- Before getting into the details, we need to describe programming languages

Programing Languages

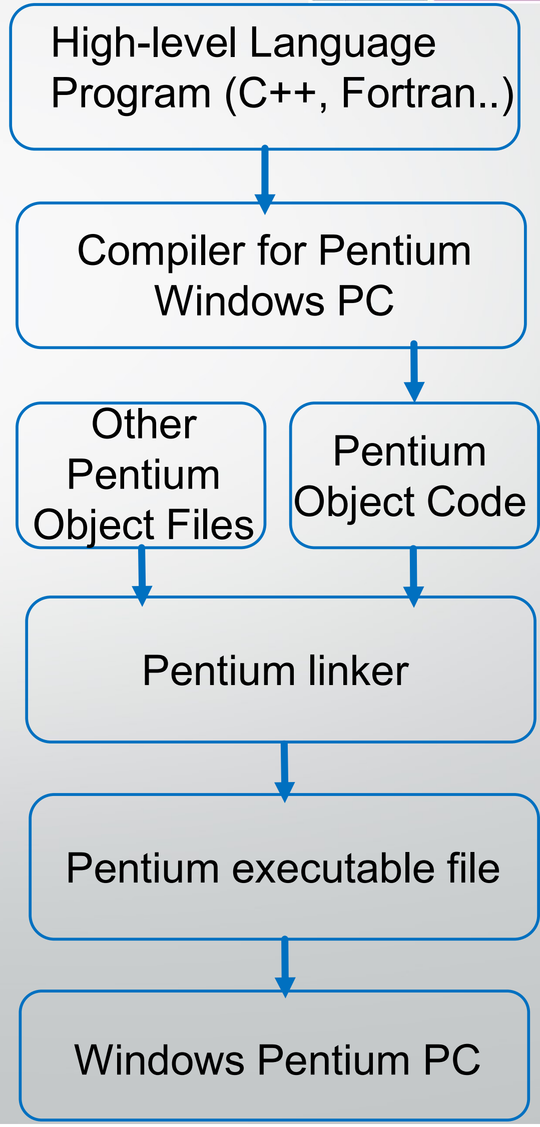
OÉ Gaillimh
NUI Galway

- High level languages
  - Hide all of the details about the computer and the operating system
  - Platform independent
- Assembly language
  - Platform dependent
  - Processors are made usually backwards compatible
- Machine languages
  - Contain the binary values that cause the processor to perform certain operations
  - Platform specific

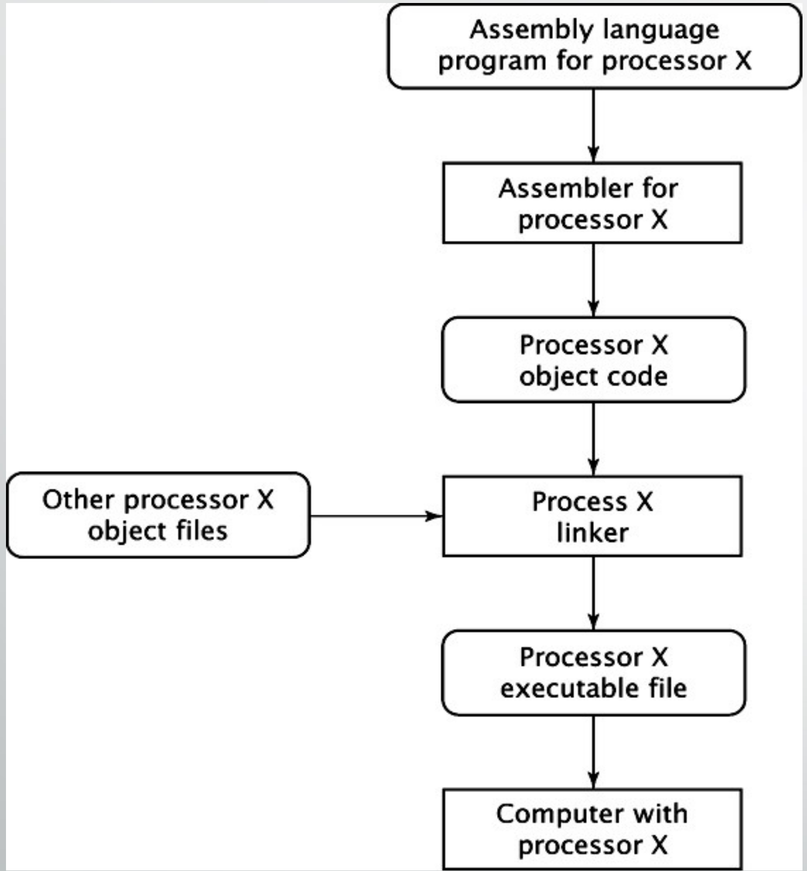
Compiling Native Code

Compiler:
- Checks to make sure every line in the code is valid
- Once the program is syntax error free, it will finish compiling the **source code** and generates **object code**
- Object code is the machine language equivalent to the source code
- At this point, the program has been complied successfully, but is not ready to execute.

Linking:
- Some programs need object code from other programs or libraries (other object files)
- A linker combines your object code with any other required object code
- The combined code is stored as executable file, the code that the computer runs.
- A loader copies the executable file into the memory, and then the microprocessor runs the code contained in that file.

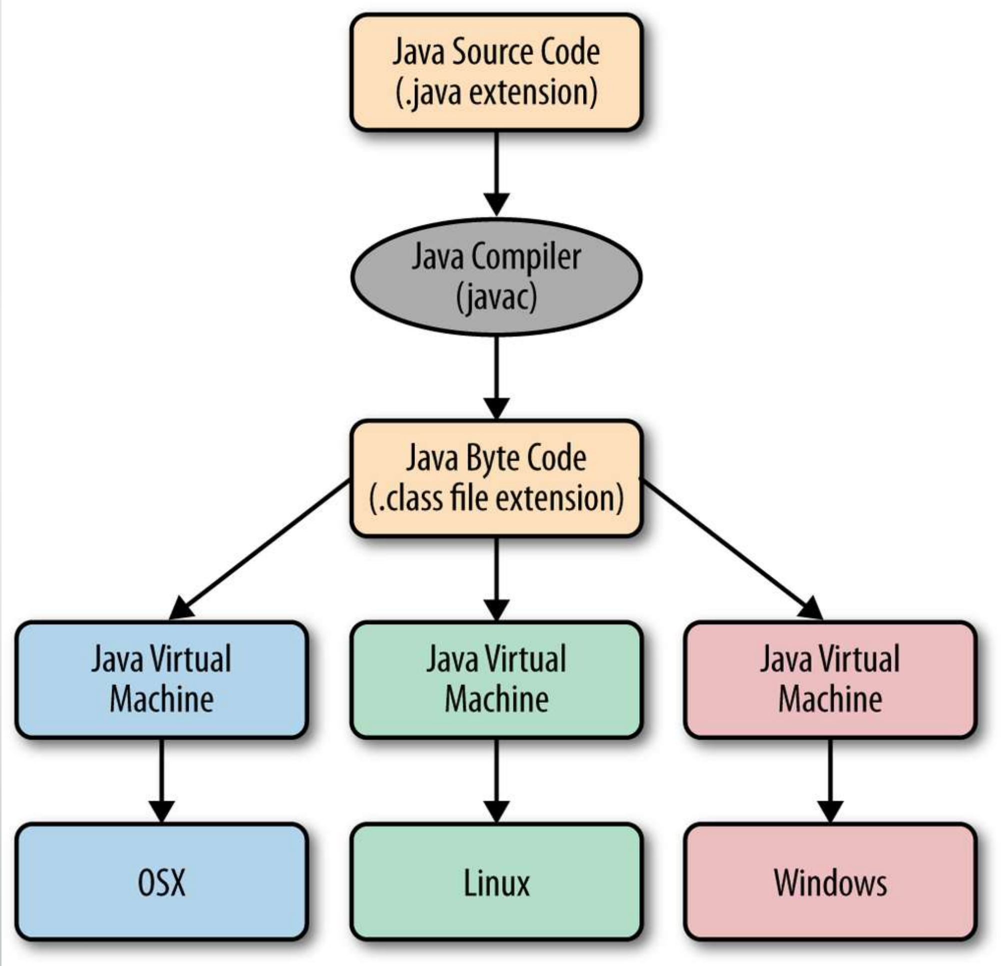


Assembling Programs



- Assembly language is specific to one microcontroller
- Converts the source code into object code
- The linker will combine the object code of your program with any other required object code to produce executable code
- Loader will load the executable code into memory, for execution

Java – Different way of programming



https://medium.com/@PrayagBhakar/lesson-2-behind-the-scenes-4df6a461f31f

Instruction Set Architecture

- Defines any aspects of the processor that an assembly language programmer needs to know, in order to write a correct program
- Specifies:
  - The registers accessible to the programmer, their size and the instructions in the instructions set that can use each register
  - Information necessary to interact with the memory
    - Certain microprocessors require instructions to start only at specific memory locations; this alignment of the instructions will be part of the instruction architecture
  - How microprocessor reacts to interrupts
    - Some microprocessors have interrupts, that cause the processor to stop what is doing and perform some other preprogrammed functions (interrupt routines)

RISC vs. CISC (1)

OÉ Gaillimh
NUI Galway

- The belief that better performance would be obtained by reducing the number of instruction required to implement a program, led to design of processors with very complex instructions (CISC)
  - CISC – ***C***omplex ***I***nstruction ***S***et ***C***omputers

- As compiler technologies improved, researchers started to wonder if CISC architectures really delivered better performances than architectures with simpler instruction set
  - RISC – ***R***educed ***I***nstruction ***S***et ***C***omputers

RISC vs. CISC (2)

- CISC
  - Fewer instructions to execute a given task than RISC
  - Programs for CISC take less storage space than programs for RISC
  - Arithmetic or other instructions may read their operand from memory and could write the result in memory

- RISC
  - Simpler instructions, faster execution speeds per instruction, more instructions executed in same amount of time than CISC
  - Cheaper to implement (simple instruction set results in simple implementation internal micro-architecture)
  - Load/Store architecture – only load and store are used to access the external memory

RISC vs. CISC (3)

OÉ Gaillimh
NUI Galway

RISC	CISC

LD R4, (R1)
LD R5, (R2)
ADD R6, R4, R5
ST (R3), R6
	ADD (R3), (R2), (R1)

- Addition of two operands from memory, with result written in memory, in RISC and CISC architectures
- Having an operation broken into small instructions (RISC) allows the compiler to optimize the code
  - i.e. between the two LD instructions (memory is slow) the compiler can add some instructions that don't need memory access
- The CISC instruction has no option but to wait for its operands to come from the memory, potentially delaying other instructions

Instruction types

- Data Transfer Instructions
  - Operations that move data from one place to another
  - These instructions don't actually modify the data, they just copy it to the destination
- Data Operation Instructions
  - Unlike the data transfer instructions, the data operation instructions do modify their data values
  - They typically perform some operation using one or two data values (operands) and store the result
- Program Control Instructions
  - Jump or branch instructions used to go in another part of the program; the jumps can be absolute (always taken) or conditional (taken only if some condition is met)
  - Specific instructions that can generate interrupts (software interrupts)

Data Transfer Instructions (1)

- Load data from memory into the microprocessor
  - These instructions copy data from memory into microprocessor registers (i.e. LD)

- Store data from the microprocessor into the memory
  - Similar to load data, except that the data is copied in the opposite direction (i.e. ST)
  - Data is saved from internal microprocessor registers into the memory

- Move data within the microprocessor
  - These instructions move data from one microprocessor register to another (i.e. MOV)

Data Transfer Instructions (2)

- Input data to the microprocessor
  - A microprocessor may need to input data from the outside world, these are the instructions that do input data from the input device into the microprocessor
  - In example: microprocessor needs to know which key was pressed (i.e. IORD)
- Output data from the microprocessor
  - The microprocessor copies data from one of its internal registers to an output device
  - In example: microprocessor may want to show on a display the content of an internal register (the key that have been pressed) (i.e. IOWR)

Data Operation Instructions

- Arithmetic instructions
  - add, subtract, multiply or divide: ADD, SUB, MUL, DIV, etc..
  - Instructions that increment or decrement one from a value: INC, DEC
  - Floating point instructions that operate on floating point values (as opposed to integer values): FADD, FSUB, FMUL, FDIV
- Logic Instructions: AND, OR, XOR, NOT, etc …
- Shift Instructions: SR, SL, RR, RL, etc…

Program Control Instructions (1)

OÉ Gaillimh
NUI Galway

- Conditional or unconditional jump and branch instructions
  - JZ, JNZ, JMP, etc…
- Comparison instructions
  - TEST
- Instructions to call a (and return from) routine; they can be as well, conditional
  - CALL, RET, IRET etc..

Program Control Instructions (2)

- Specific instructions to generate software interrupts; there are also interrupts that are not part of the instruction set, called hardware interrupts, generated by devices outside of a microprocessor
  - INT

- Exceptions and traps – are triggered when valid instructions perform invalid operations, such as dividing by zero

- Halt instructions - causes the processor to stop executions, such as at the end of a program
  - HALT

Data Types

- A microprocessor must operate with multiple data types; this has a direct implication in the instruction architecture set, because the designer has to include different instructions to perform the same operation on different data types
- Numeric data representation:
  - Integer representation
    - Unsigned representation: n bit value range from 2^n -1 to 0
    - Signed representation: n bit value range from -2^{n-1} to 2^{n-1}-1
  - Floating point representation
    - A processor may have special registers and instructions for floating point data
- Boolean data:
  - Nonzero value is used to represent TRUE and zero value is used to represent FALSE
- Character data
  - Stored as binary value, encoded using ASCII, UNICODE, etc…
  - A microprocessor may concatenate strings of characters, replace certain characters in a string, etc..
  - Some instruction sets do include instructions to directly manipulate character data

Instruction Formats

- An instruction is represented as a binary value with specific format, called the **instruction code**
- It is made out of different groups of bits, with different significations:
  - **Opcode** – represents the operation to be performed (it is the instruction identifier)
  - **Operands** – one, two or three represent the operands of the operation to be performed
- A microprocessor can have one format for all the instructions or can have several different formats
- An instruction is represented by a single instruction code

Instruction Formats

4 bits	2 bits	2 bits	2 bits
opcode	operand #1	operand #2	operand #3

ADD A,B,C	$(A=B+C)$	1010 00 01 10

(a)

4 bits	2 bits	2 bits
opcode	operand #1	operand #2

MOVE A,B	$(A=B)$	1000 00 01
ADD A,C	$(A=A+C)$	1010 00 10

(b)

4 bits	2 bits
opcode	operand

LOAD B	$(Acc=B)$	0000 01
ADD C	$(Acc=Acc+C)$	1010 10
STORE A	$(A=Acc)$	0001 00

(c)

4 bits
opcode

PUSH B	(Stack$=B$)	0101
PUSH C	(Stack$=C,B$)	0110
ADD	(Stack$=B+C$)	1010
POP A	($A=$stack)	1100

(d)

Instruction Formats

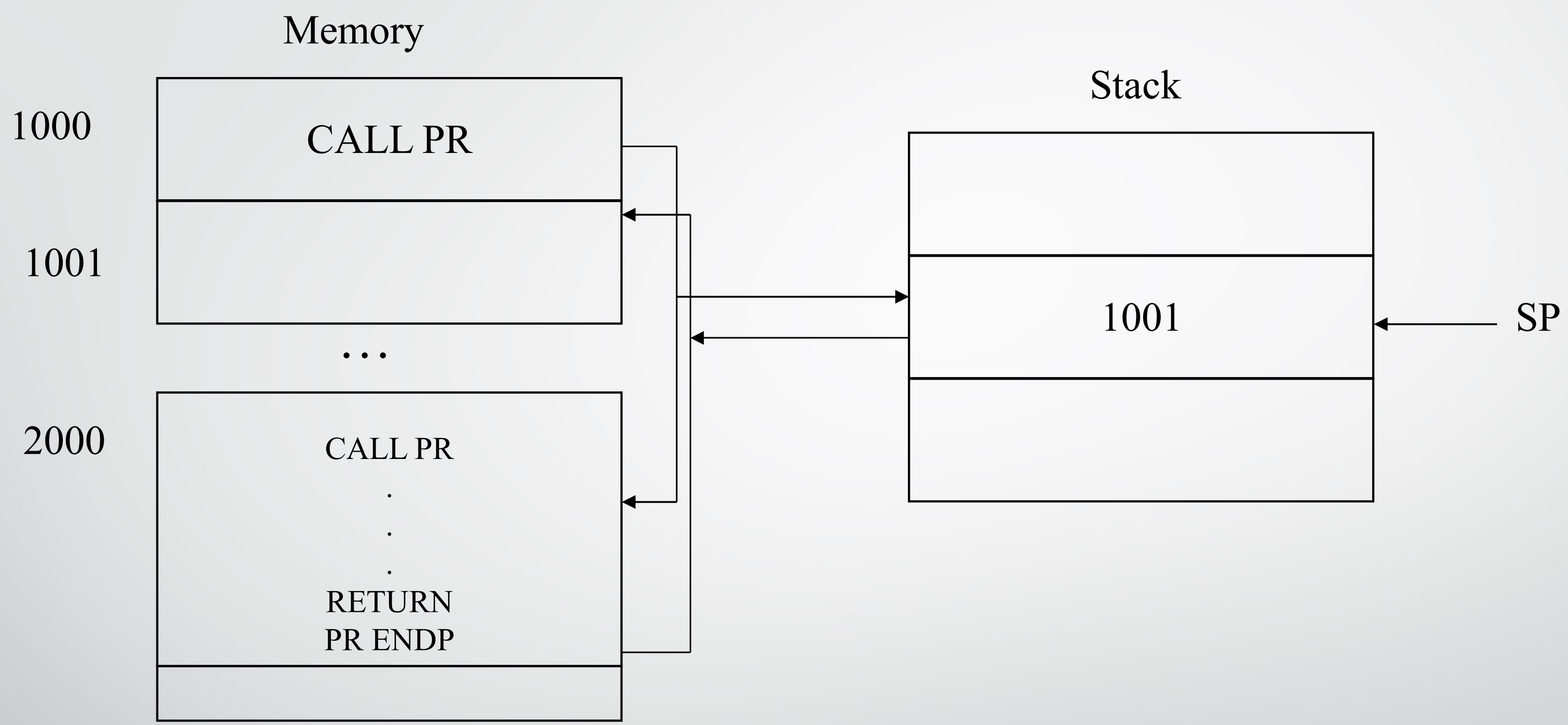
OÉ Gaillimh
NUI Galway

- Fewer operands translates into more instructions to accomplish the same task
- The hardware required to implement the microprocessor becomes less complex with fewer operands; microprocessors whose instructions specify a fewer number of operands can execute instructions more quickly than those that specify more operands
- The example was simplified to show the difference between three, two, one and zero operands instructions; in practice, the instructions require many more bits than those used in these examples; an operand field may specify an arbitrary memory address, rather than one of the four registers; this could require 16, 32 or even more bits per operand

CPU Elements

- **Program Counter** or PC contains the address of the instruction that will be executed next
- **Stack** – a data structure of last in first out type
  - Dedicated hardware stack – it has a hardware limitation
  - Memory implemented stack –limited by the physical memory of the system
    - A stack is described by a special register – **stack pointer** – that holds the address of the last
  - It can be used explicitly to save/restore data
  - It is used implicitly by procedure call instructions (if available in the instruction set)
- **IR** – Instruction Register that holds the current instruction being processed by the microprocessor; it is not exposed through the instruction set architecture; just an organization element

Implicit Stack Usage



- CALL – before the jump to the PR address, the call instruction will save the PC (program counter) in the stack
- Return – will extract the address of the next instruction before jump and restore the PC (program counter) value

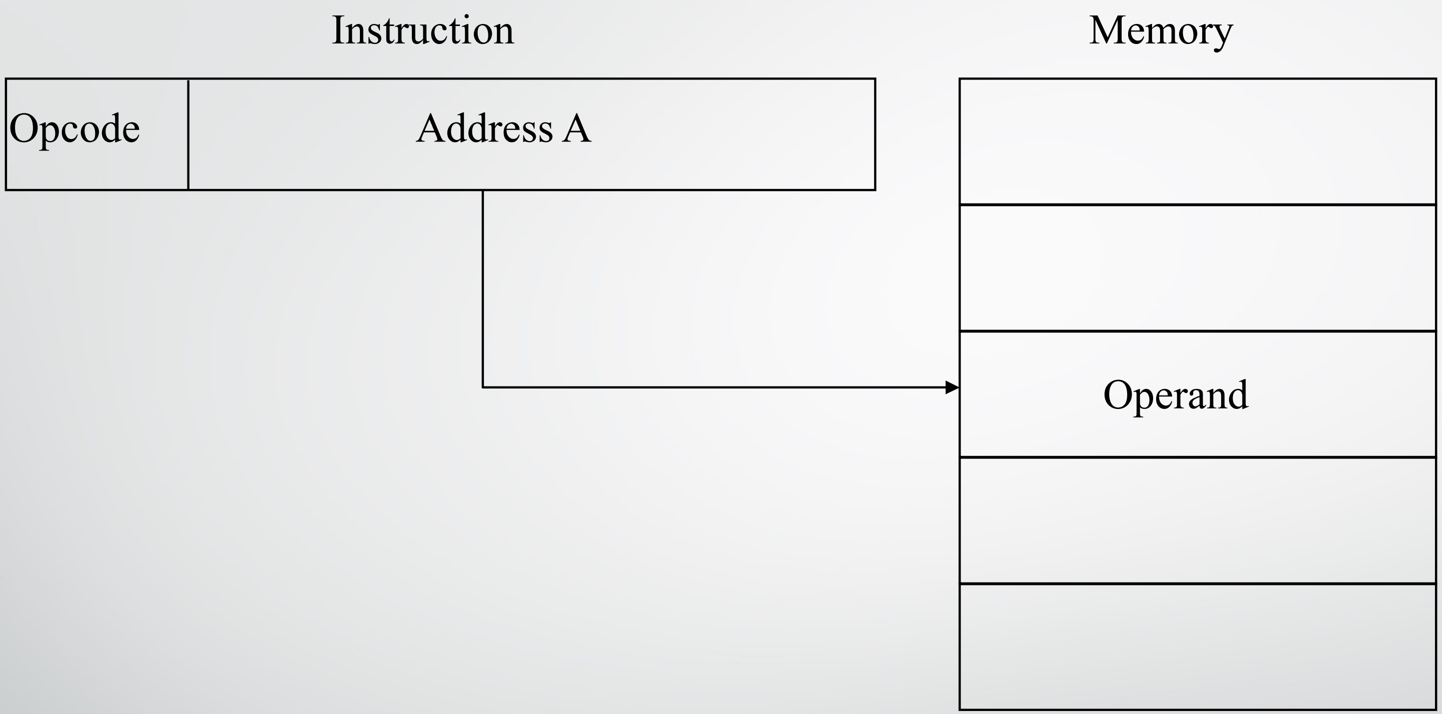
Explicit stack usage

- Typical Stack operations (assuming that the stack grows from higher addresses towards lower addresses):
  - PUSH (X):
    - (SP)= (SP)-1
    - ((SP)) = X
  - POP (X)
    - X = ((SP))
    - (SP) = (SP)+1

Addressing Modes

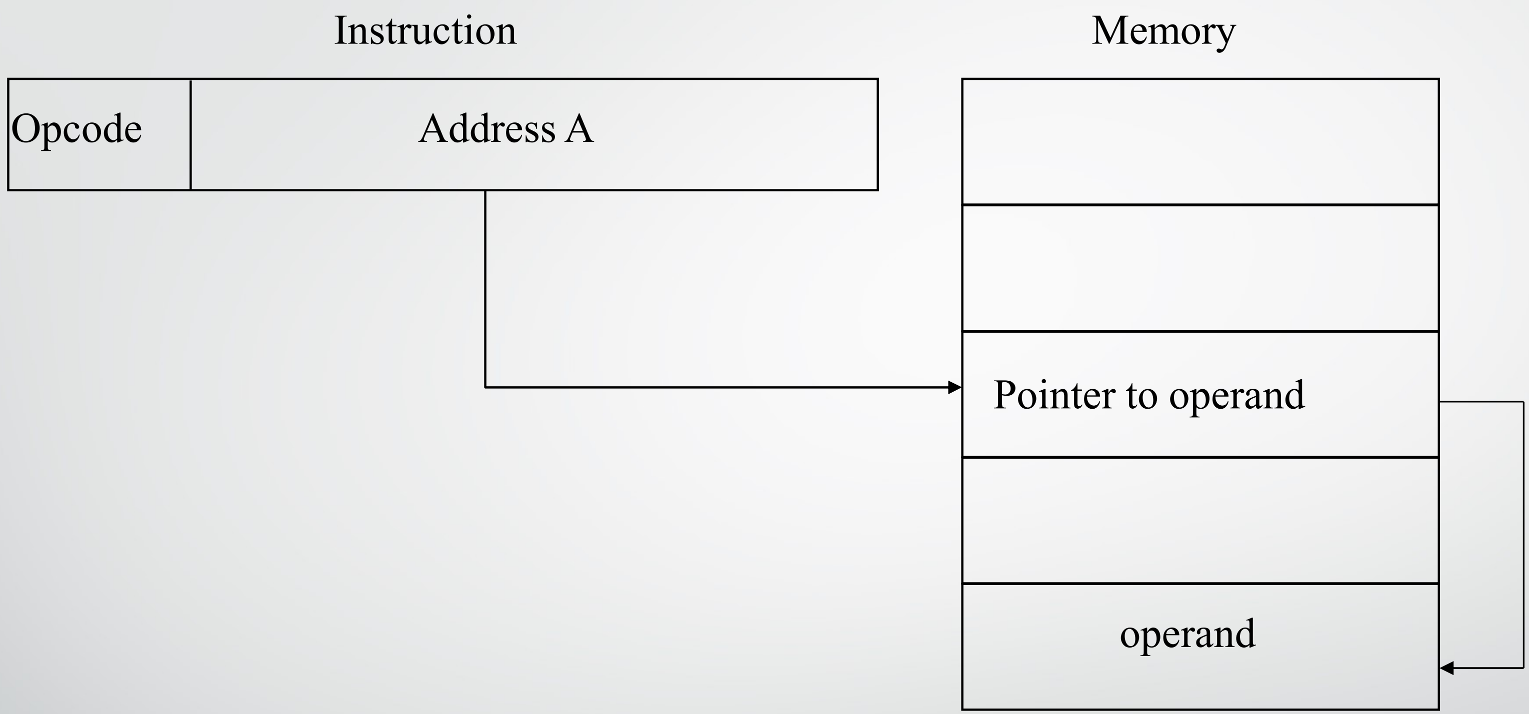
- When a microprocessor accesses memory, to either read or write data, it must specify the memory address it needs to access
- Several addressing modes to generate this address are known, a microprocessor instruction set architecture may contain some or all those modes, depending on its design
- In the following examples we will use the LDAC instruction (loads data from a memory location into the AC (accumulator) microprocessor register)

Direct mode



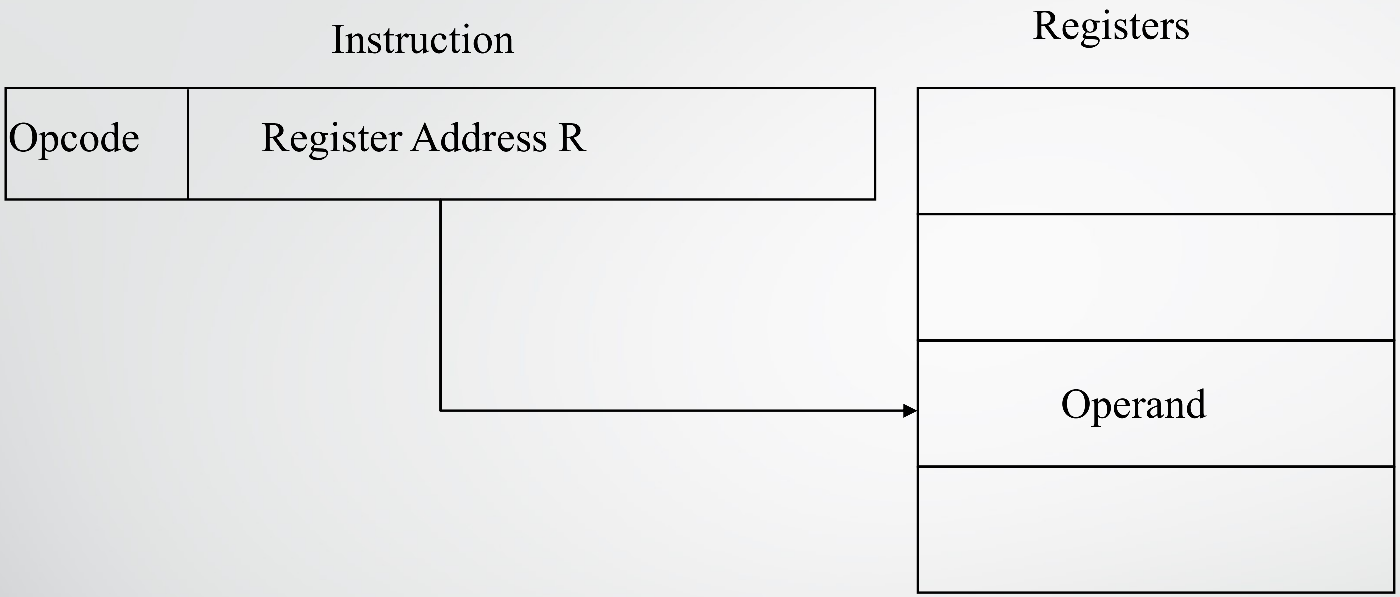
- Instruction includes the A memory address
- LDAC 5 – accesses memory location 5, reads the data (10) and stores the data in the microprocessor's accumulator
- This mode is usually used to load variables and operands into the CPU

Indirect mode



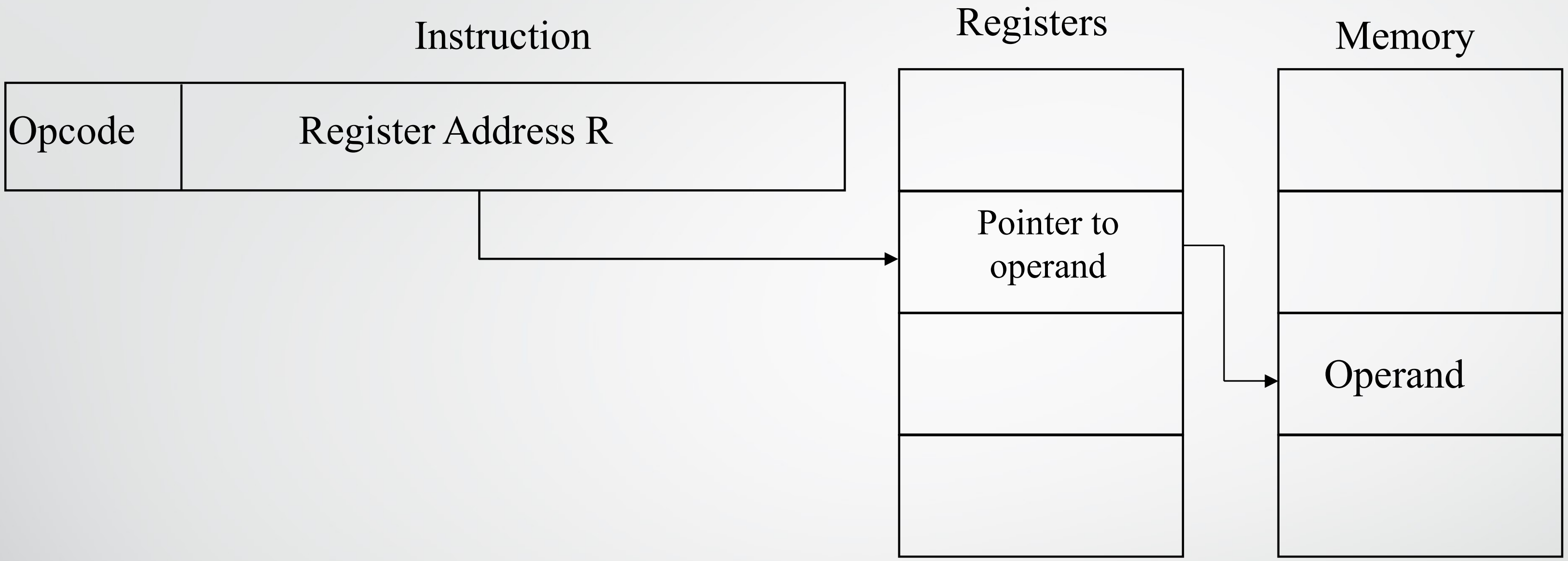
- Starts like the direct mode, but it makes an extra memory access. The address specified in the instruction is not the address of the operand, it is the address of a memory location that contains the address of the operand.
- LDAC @5 or LDAC (5), first retrieves the content of memory location 5, say 10, and then CPU goes to location 10, reads the content (20) of that location and loads the data into the CPU (used for relocatable code and data by operating systems)

Register direct mode



- It specifies a register instead a memory address
- LDAC R – if register R contains a value 5, then the value 5 is copied into the CPU’s accumulator
- No memory access
- Very fast execution
- Very limited address space

Register indirect mode



- LDAC @R or LDAC (R) – the register contains the address of the operand in the memory
- Register R (selected by the operand), contains value 5 which represents the address of the operand in the memory (10)
- One fewer memory access than indirect addressing

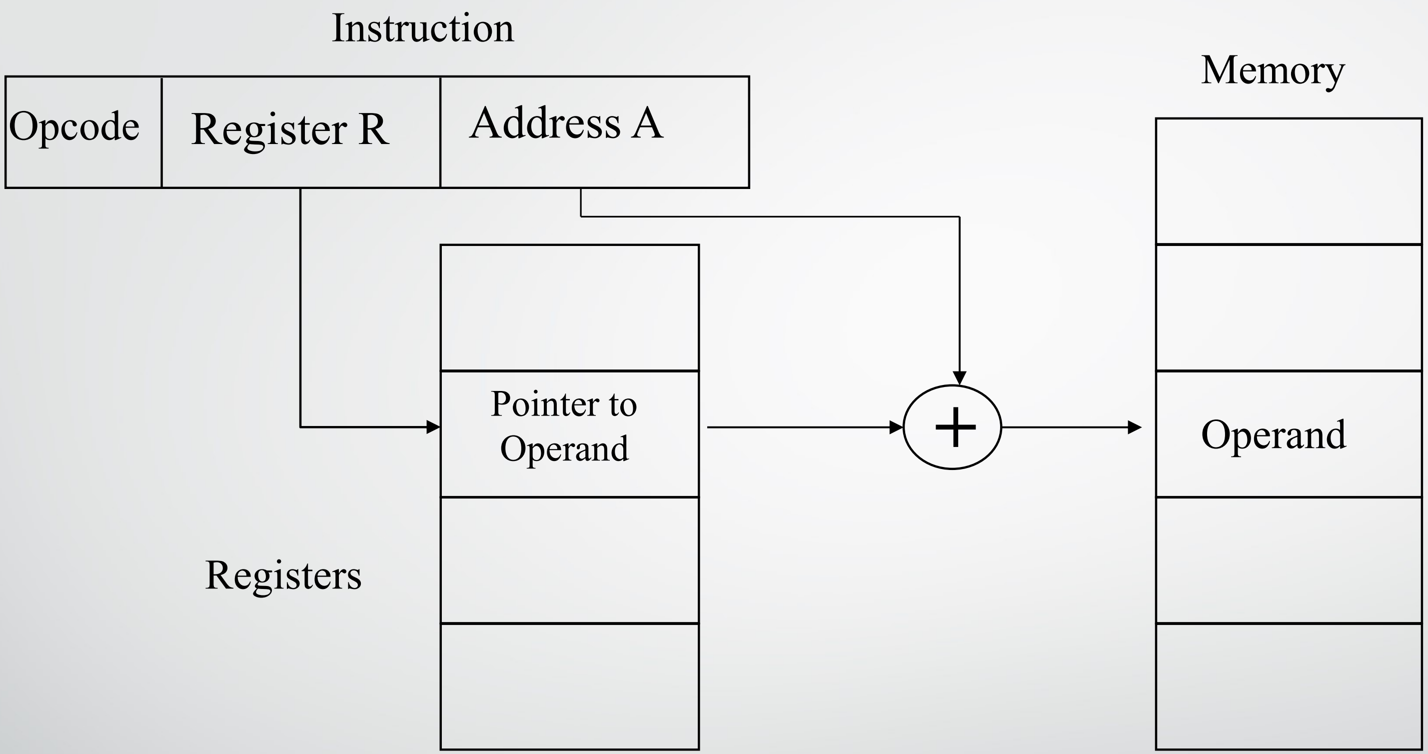
Immediate mode

- The operand is not specifying an address, it is the actual data to be used
- LDAC #5 loads the actual value 5 into the CPU's accumulator
- No memory reference to fetch data
- Fast, no memory access to bring the operand

Implicit addressing mode

- Doesn’t explicitly specify an operand
- The instruction implicitly specifies the operand because it always applies to a specific register
- This is not used for load instructions
- As an example, consider an instruction CLAC, that is clearing the content of the accumulator in a processor and it is always referring to the accumulator
- This mode is used also in CPUs that use a stack to store data; they don’t specify an operand because it is implicit that the operand must come from the stack

Displacement addressing mode



- Effective Address = A + (content of R)
- Address field hold two values
  - A = base value
  - R = register that holds displacement
  - or vice versa

Relative addressing mode

- It is a particular case of displacement addressing, where the register is the program counter; the supplied operand is an offset; Effective Address = A + (PC)
- The offset is added to the content of the CPU's program counter register to generate the required address
- The program counter contains the address of next instruction to be executed, so the same relative instruction will produce *different* addresses at *different* locations in the program
- Consider that the relative instruction LDAC $5 is located at memory address 10 and it takes two memory locations; the next instruction is at location 12, so the operand is actually located at (12 +5) 17; the instruction loads the operand at address 17 and stores it in the CPU's accumulator
- This mode is useful for short jumps and relocatable code

Indexed addressing mode

- Works like relative addressing mode; instead of adding the A to the content of program counter (PC), the A is added to the content of an index register
- If the index register contains value 10, then the instruction LDAC 5(X) reads data from memory at location (5+10) 15 and stores it in the accumulator
- Good for accessing arrays
  - Effective Address = A + R
  - R++

Based addressing mode

- Works the same with indexed addressing mode, except that the index register is replaced by a base address register
- A holds displacement
- R holds pointer to base address
  - R may be explicit or implicit

Addressing modes

a) 0: LDAC 5
instruction gets data from location 5
5: 10 → stores value in CPU

b) 0: LDAC @ 5
instruction gets address from location 5
5: 10
then gets data from location 10
10: 20 → stores value in CPU

c) 0: LDAC R
instruction gets data from register R
R: 5 → stores value in CPU

d) 0: LDAC R
instruction gets address from register R
R: 5
then gets data from location 5
5: 10 → stores value in CPU

e) 0: LDAC #5 → stores value from instruction in CPU

f) 0: LDAC (implicit)
instruction gets value from stack
stack → stores value in CPU

g) 0: LDAC $5
1: instruction adds address of next instruction (1) to
5: offset (5) to get address (6)
6: 12 → stories value in CPU

h) 0: LDAC 5(X)
instruction gets value from index register
X: 10
then adds contents of X (10) to offset (5) to get address (15)
15: 30 → stores value in CPU

Instruction Set Architecture Design

OÉ Gaillimh
NUI Galway

- What is the processor able to do
  - If it will be general purpose, then the ISA should be pretty rich to perform a variety of tasks
  - If it will be a specialized processor, then the ISA should perform a specific set of tasks, well known in advance
- The instruction set has to have all the instructions to perform its required tasks – **completeness**
- The instruction set has to be orthogonal – to minimize the overlap between instructions
- The register set:
  - Too few registers will cause too many accesses to the memory, thus reducing performance
  - Too many registers would be overkill for specialized microcontrollers

Instruction Set Architecture Design

- Does the microprocessor have to be backwards compatible with other microprocessors?
- What type of data and sizes of data will the microprocessor deal with?
  - If floating point operation is needed, then the design must include instructions that will work on floating point data; also registers to store floating point data are needed;
- Are interrupts needed?
  - If needed, the design should include the registers and instructions to deal with interrupts
- Are conditional instructions needed?
  - Usually, the conditions are stored in 1-bit registers that store the value of various conditions; typical flags include the zero flag (set 1 when an operation produces a result of zero), sign flag (set to one when an instruction produces a negative result), etc…

References

OÉ Gaillimh
NUI Galway

- "Computer Systems Organization & Architecture", John D. Carpinelli, ISBN: 0-201-61253-4

OÉ Gaillimh
NUI Galway

- CT101 -
Computer Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

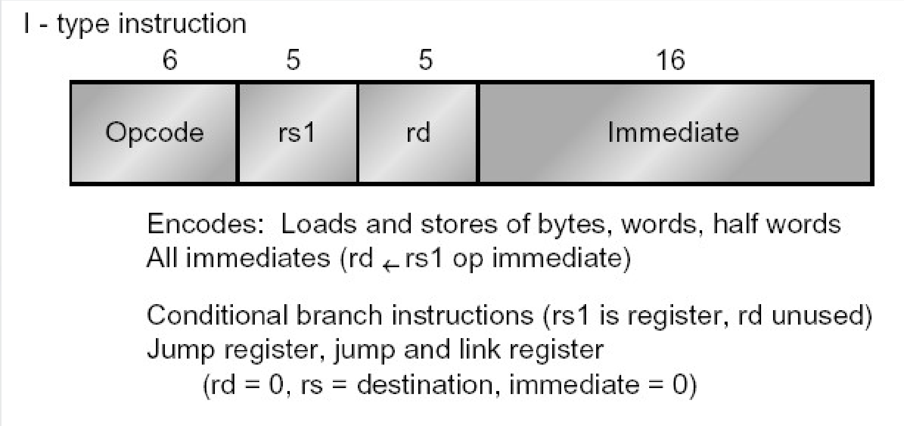
Processor Design

- GPR processor – non-pipeline implementation
- Pipeline
- GPR processor – pipeline implementation
- Performance issues in pipeline

GPR Example processor (1)

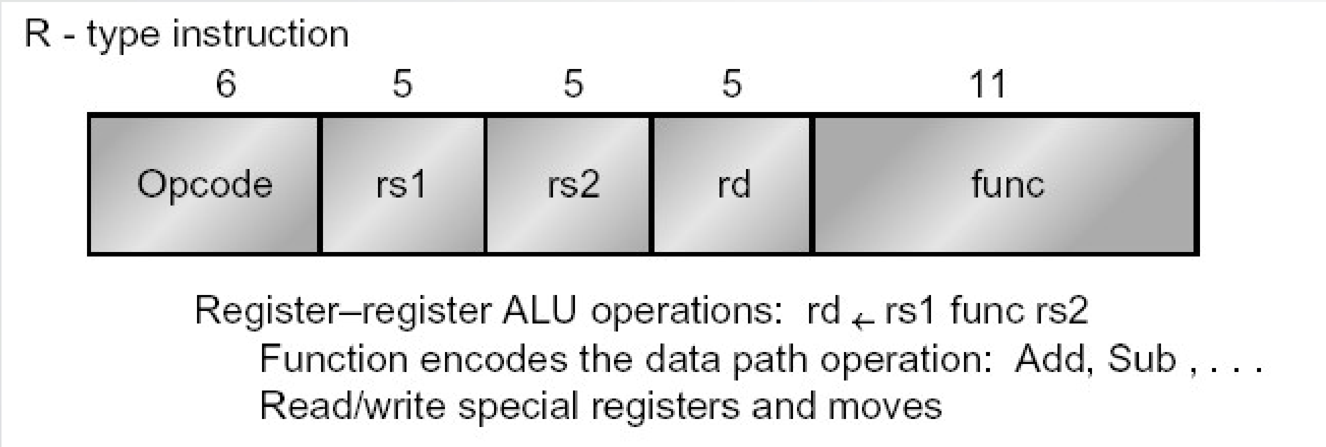
- Consider a simple GPR architecture
  - 32 GPR registers, R0 to R31
  - Value of R0 is always 0
- Data types:
  - 8-bit bytes, 16-bit half words and 32-bit words (integer data)
  - Operations work on 32-bit integers
  - 8 bit and 16-bit operands are loaded into the 32-bit registers with sign bit duplicated
- Addressing modes:
  - Immediate (16-bit field)
  - Displacement mode (contents of register added to 16-bit address field)

GPR Example Processor (2)



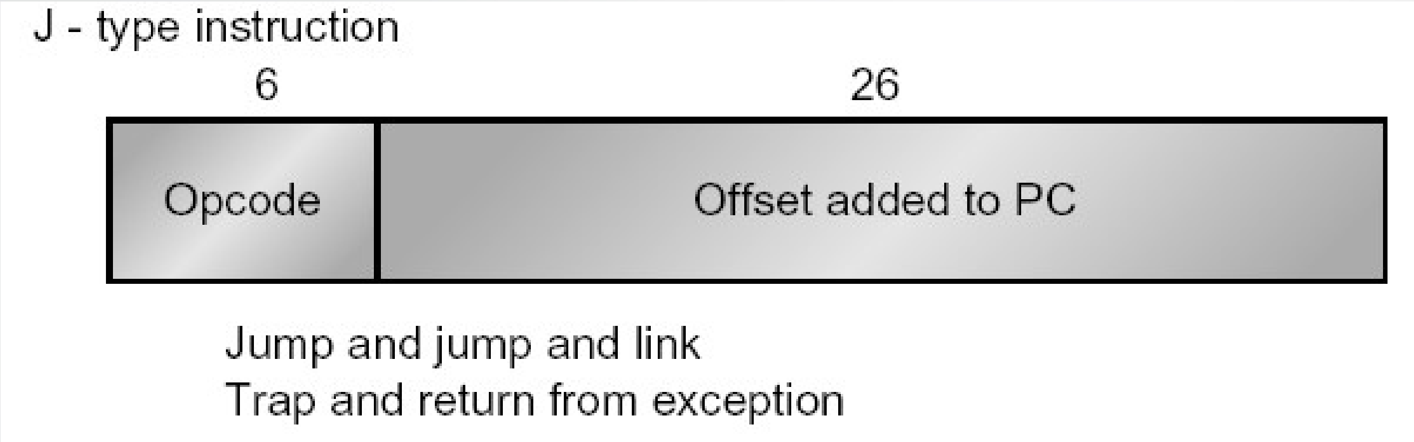
- Examples of I Instructions
  - LW R2, 50 (R3) – Regs[R2] <- Mem{50 + Regs[R3]}
  - LW R2, 50 (R0) – Regs[R2] <- Mem{50 + 0}
  - SW R3, 500 (R4) – Mem{500+Regs[R4]}<-Regs[R3]
  - BNEZ R4, name – if (Regs[R4]){PC<-name}
  - JR R3 – PC<- Regs[R3] (jump register)
  - JALR R2 – Regs[R31] <-PC+4; PC<-Regs[R2] (jump and link register)

GPR Example Processor (3)



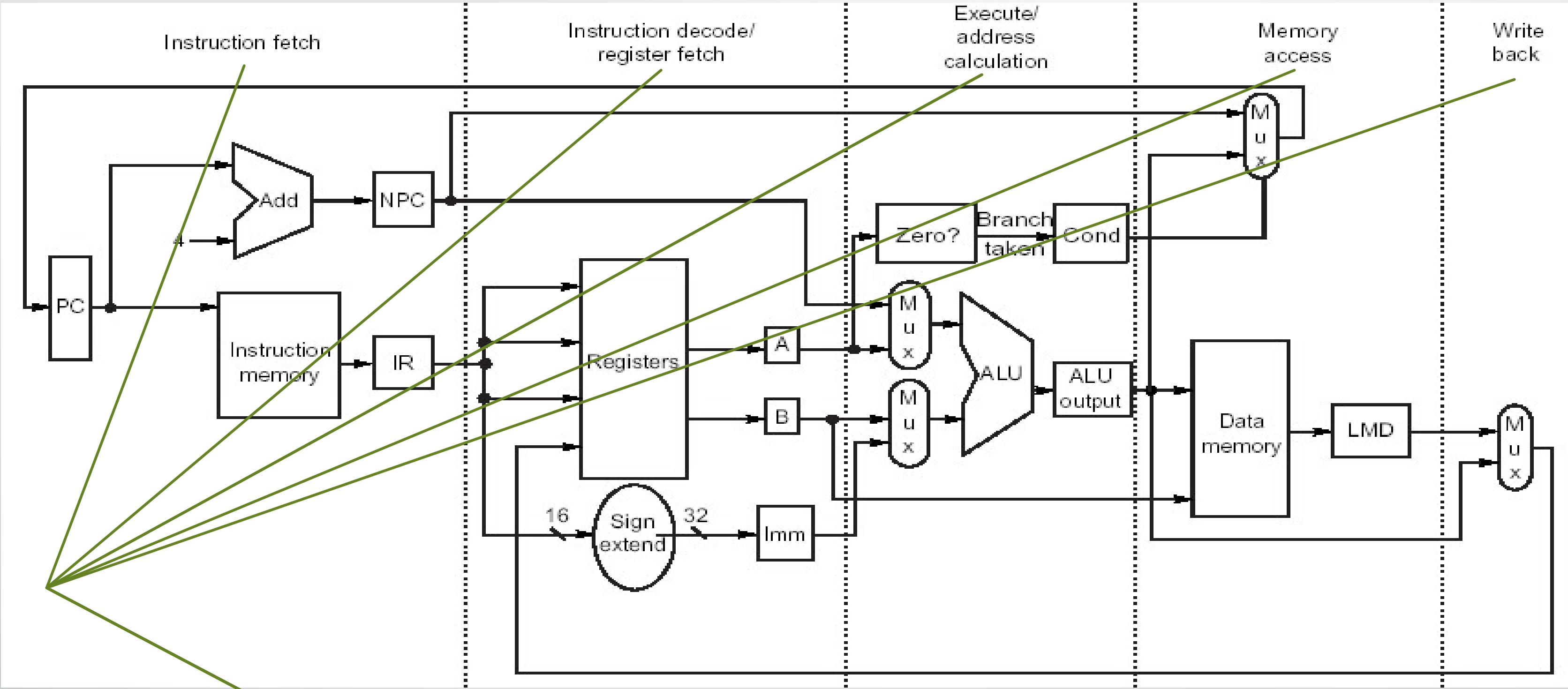
- Example of R – type instructions
  - ADD R1, R2, R3 – Regs[R1]<- Regs[R2]+Regs[R3]
  - SLT R1, R2, R3 – if (Regs[R2]<Regs[R3]{Regs[R1]<-1}else{Regs[R1]<-0} (set if less than)

GPR example processor (4)



- J name – PC<-name
- JAL name – Regs[31]<-PC+4; PC<-name (jump and link)

Example processor implementation



Write-Back Cycle (WB)

- Register – Register ALU Instruction
  - $Regs[IR_{16...20}] \leftarrow ALUOutput$
- Register –Immediate ALU Instruction
  - $Regs[IR_{11...15}] \leftarrow ALUOutput$
- Load Instruction
  - $Regs[IR_{11..15}] \leftarrow LMD$

Instruction Fetch

- Instruction Fetch Cycle (IF):
  - IR← Mem[PC]
  - NPC← PC+4
- Operation:
  - Send out the PC and fetch the instruction from memory
  - Increment the PC by 4 to address the next instruction and save it in NPC (**N**ext **P**rogram **C**ounter) register

Instruction Decode

- Instruction Decode/Register Fetch Cycle (ID)
  - $A \leftarrow Regs[IR_{6...10}]$
  - $B \leftarrow Regs[IR_{11...15}]$
  - $Imm \leftarrow ((IR_{16})^{16}\#\#IR_{16...31}$
- Operation
  - Decode the instruction and access the register files to access the registers; the output of the general-purpose registers are read into two temporary register (A and B) for use in later clock cycles.
  - The lower 16 bits of IR are also *sign extended* and stored into temporary register Imm, for later use

Instruction Execution

Instruction Execution/Effective Address Cycle (EX)

- Memory Reference Instruction
  - ALUOutput ← A + Imm
  - The ALU adds the operands to form the effective address and places the result into the register ALUOutput
- Register – Register ALU Instruction
  - ALUOutput ← A func B
  - The ALU performs the function specified by the instruction and places the result into the ALUOutput register
- Register – Immediate ALU Instruction
  - ALUOutput ← A op Imm
  - The ALU performs the operation indicated by the opcode on the value from register A and the value from Imm. Result is placed in ALUOutput register
- Branch Instruction
  - ALUOutput ←NPC + Imm
  - Cond ← (A op 0)
  - The ALU adds the contents of NPC with the sign extended value of Imm to compute the address of the branch target. Register A is checked to see if the branch is taken. The comparison operation op is determined by the branch opcode (i.e. op is "==" for instruction BEQZ)

Instruction Memory Access

OÉ Gaillimh
NUI Galway

- Memory Access/Branch Completion Cycle (MEM)
  - Memory Reference Instruction
    - Load
      - LMD $\leftarrow$ Mem[ALUOutput]
    - Store
      - Mem [ALUOutput] $\leftarrow$B
    - Access memory if needed.
      - If instruction is a load, then data returns from memory and is placed in LMD register (Load Memory Data)
      - If instruction is a store, then the data from B register is written back into the memory, at location stored in the previous cycle in ALUOutput
  - Branch Instruction
    - If (cond) {PC$\leftarrow$ALUOutput} else {PC$\leftarrow$NPC}
    - If the instruction branches, then the PC is replaced with branch destination address. Otherwise it is replaced with incremented PC in the register NPC

Instruction Write-Back

- Write-Back Cycle (WB)
  - Register – Register ALU Instruction
    - Regs[$IR_{16\ldots20}$]← ALUOutput
  - Register –Immediate ALU Instruction
    - Regs[$IR_{11\ldots15}$] ← ALUOutput
  - Load Instruction
    - Regs[$IR_{11..15}$] ← LMD
  - Write the results back into the register file, whether the data comes from the main memory or as a result of an operation (from ALU); the register destination can be in two positions which is up to the instruction type

Pipeline

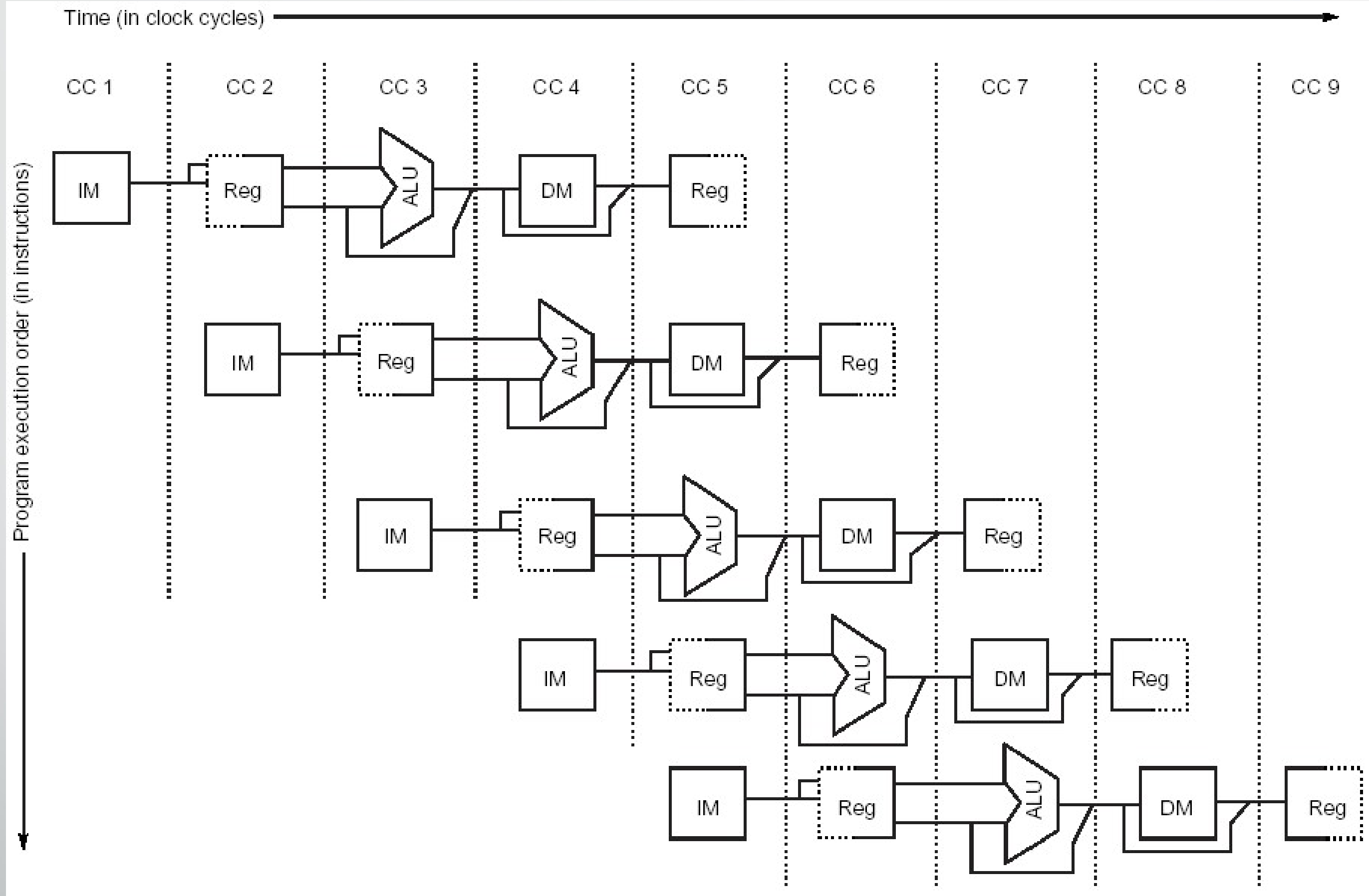
- ***Pipelining*** is an implementation technique whereby multiple instructions are overlapped in execution
  - The goal of the pipeline is to reduce the execution time for a set of instructions
  - Today, pipelining is the key implementation technique for modern processors
- Each stage in the pipeline completes a part of the instruction
- ***Throughput*** is determined by how often an instruction exits the pipeline (gets completed)

Basic Pipeline (1)

Instruction Number	Clock number									
	1	2	3	4	5	6	7	8	9	10
Instruction i	IF	ID	EX	MEM	WB					
Instruction i+1		IF	ID	EX	MEM	WB				
Instruction i+2			IF	ID	EX	MEM	WB			
Instruction i+3				IF	ID	EX	MEM	WB		
Instruction i+4					IF	ID	EX	MEM	WB	
Instruction i+5						IF	ID	EX	MEM	WB

- We can pipeline the presented datapath with no changes by starting a new instruction on each clock cycle
- While each instruction will take 5 clock cycles to complete, at each clock cycle, the hardware will initiate the execution of a new instruction

Basic Pipeline (2)



- Example processor datapath, drawn in pipeline fashion

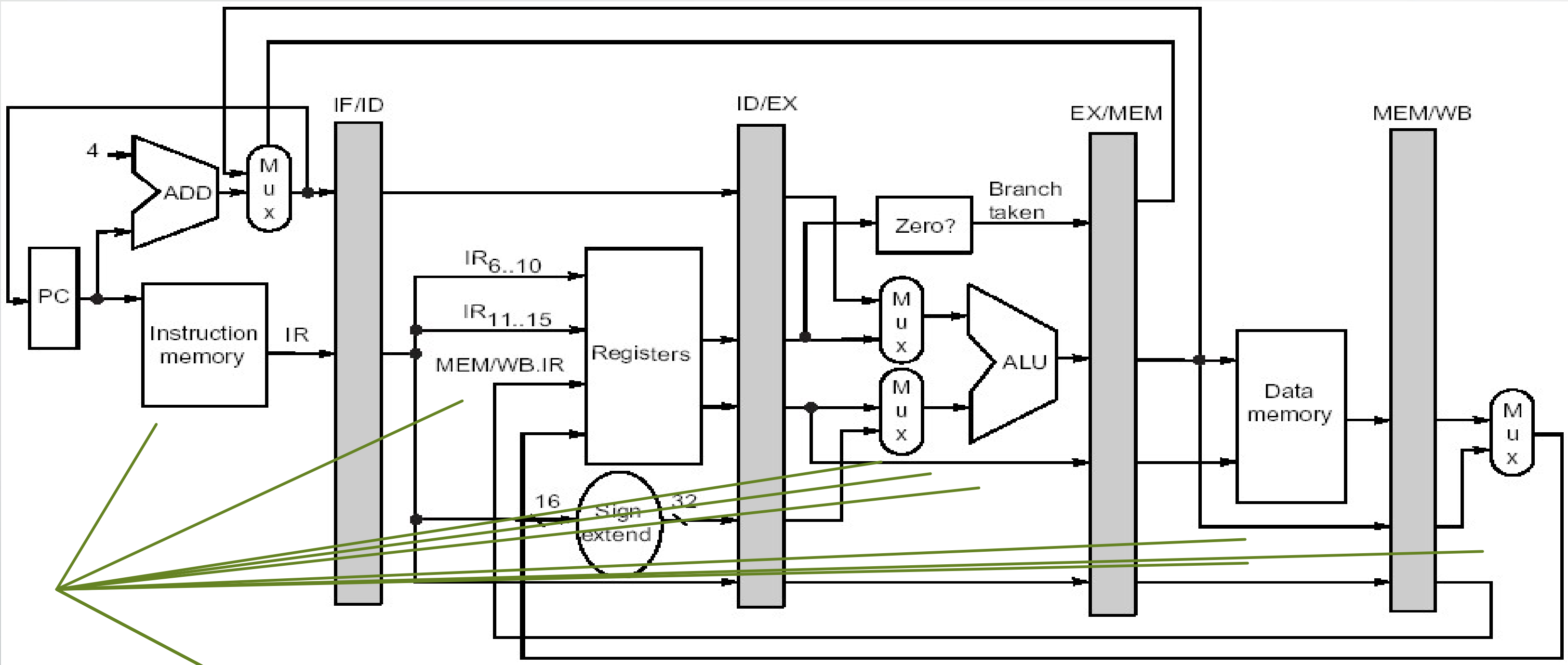
Basic Pipeline (3)

- The use of pipeline forces us to think about:
  - Datapath should use separate instructions and data memories
    - The memory system must deliver five times the bandwidth
  - The register file is used in two stages: for reading in ID stage and for writing in WB stage
    - This means that we need to be able to perform two reads and a write every clock cycle
    - What if a read and a write target the same register?
  - PC – to start a new instruction every clock, PC has to be incremented and stored every clock cycle and this should be done during IF in preparation for next instruction
    - The problem occurs when we consider the effect of taken branches, that change the PC as well, but not until the MEM stage
    - We will deal with this problem by reorganizing the way PC gets written

Basic Pipeline (4)

- Pipelining the datapath requires that values passed from one pipe stage to the next pipe stage must be placed in registers. Those registers, placed between each pipe stage, are called **Pipeline Registers**.
- The pipeline registers serve to convey data and control information from one stage to the next.
- PC (Program Counter) can also be thought as a pipeline register that sits before the IF phase of an instruction, leading to one pipeline register for each stage.
- Most of the data flows from left to right, which is from earlier in time to later in time. The paths that flow from right to left (which carry the PC and the values for WB stage) introduce complications into our pipeline.

Basic Pipeline (5)



one

Write-Back Cycle (WB)

ALU Instructions

For Register – Register ALU Instruction

Regs[MEM/WB.IR$_{16..20}$]← MEM/WB.ALUOutput

For Register –Immediate ALU Instruction

Regs[MEM/WB.IR$_{11...15}$] ← MEM/WB.ALUOutput

Memory Access (Load) Instruction

Regs[MEM/WB.IR$_{11..15}$] ← MEM/WB.LMD

Pipelined Instruction Fetch

- Instruction Fetch
  - IF/ID.IR ← mem[PC]
  - IF/ID.NPC, PC ← If (EX/MEM.cond) {EX/MEM.ALUOutput}else{PC+4}
- Operation:
  - Send out the PC and fetch the instruction from memory
  - Increment the PC by 4 to address the next instruction or save the address generated by a taken branch of a previous instruction in execution stage

Pipelined Instruction Decode

- Instruction Decode Cycle/Register Fetch
  - ID/EX.A ← Regs[$IR_{6\ldots10}$]; ID/EX.B ← Regs[$IR_{11\ldots15}$]
  - ID/EX.NPC ← IF/EX.NPC;
  - ID/EX.IR ← IF/EX.IR
  - ID/EX.Imm← $(IF/ID.IR_{16})^{16}$##$IF/ID.IR_{16\ldots31}$
- Operation
  - Decode the instruction and access the register files to access the registers; the output of the general purpose registers are read into two temporary register (A and B, part of the pipeline registers ID/EX stage) for use in later clock cycles
  - The lower 16 bits of IR, stored in pipeline registers from IF/ID stage are also sign extended and stored into temporary register Imm (part of ID/EX pipeline registers), for later use
  - Values for NPC and IR are passed to the next stage of pipeline registers (from IF/ID to ID/EX)

Pipelined Instruction Execution (1)

OÉ Gaillimh
NUI Galway

- Instruction Execution/Effective Address Cycle (EX)
  - Memory Reference Instruction
    - EX/MEM.IR ← ID/EX.IR
    - EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm
    - EX/MEM.Cond ← 0
    - EX/MEM.B ← ID/EX.B
  - The value of the IR from previous stage of pipeline registers (from ID/EX) is passed onto the next stage of pipeline registers (to EX/MEM)
  - ALU adds the operands (stored in the previous stage pipeline registers – ID/EX to form the effective address and places the result into the register EX/MEM.ALUOutput, part of the next stage pipeline registers.
  - The Cond register (of EX/MEM pipeline registers) is set to 0 (no branch)
  - The value of B register from previous stage (ID/EX) is saved into the next stage pipeline registers (EX/MEM) for usage in next cycle (contains the value to be saved by a store operation).

Pipelined Instruction Execution (2)

OÉ Gaillimh
NUI Galway

- Instruction Execution/Effective Address Cycle (EX)
  - ALU Instruction
    - Register – Register ALU Instruction
      - EX/MEM.IR ← ID/EX.IR
      - EX/MEM.ALUOutput ← ID/EX.A func ID/EX.B
      - EX/Mem.Cond ← 0
      - The ALU performs the function specified by the instruction and places the result into the ALUOutput register (of the next stage pipeline registers)
    - Register – Immediate ALU Instruction
      - EX/MEM.IR ← ID/EX.IR
      - EX/Mem.ALUOutput ← ID/EX.A op ID/EX.Imm
      - EX/Mem.Cond ← 0
      - The ALU performs the operation indicated by the opcode on the value from register A and the value from Imm (both retreived from ID/EX pipeline registers). Result is placed in ALUOutput register of the EX/MEM pipeline registers

Pipelined Instruction Execution (3)

OÉ Gaillimh
NUI Galway

- Instruction Execution/Effective Address Cycle (EX)
  - Branch Instruction
    - EX/MEM.ALUOutput ←ID/EX.NPC + ID/EX.Imm
    - EX/MEM.Cond ← (ID/EX.A op 0)
    - The ALU adds the contents of NPC with the sign extended value of Imm to compute the address of the branch target. Register A is checked (from the pipeline registers of ID/EX stage) to see if the branch is taken. The comparison operation op is determined by the branch opcode (i.e. op is "==" for instruction BEQZ)

Pipelined Instruction Memory Access (1)

- Memory Access (MEM)
  - Memory Reference Instruction
    - MEM/WB.IR ← EX/MEM.IR
    - For Load
      - MEM/WB.LMD ← Mem[EX/MEM.ALUOutput]
    - For Store
      - Mem [EX/MEM.ALUOutput] ←EX/MEM.B
    - Access memory:
      - If instruction is a load, then data returns from memory and is placed in LMD register (Load Memory Data) of MEM/WB pipeline registers
      - If instruction is a store, then the data from B register of EX/MEM pipeline registers is written back into the memory, at location stored in previous cycle in ALUOutput (of EX/MEM pipeline registers)

Pipelined Instruction Memory Access (2)

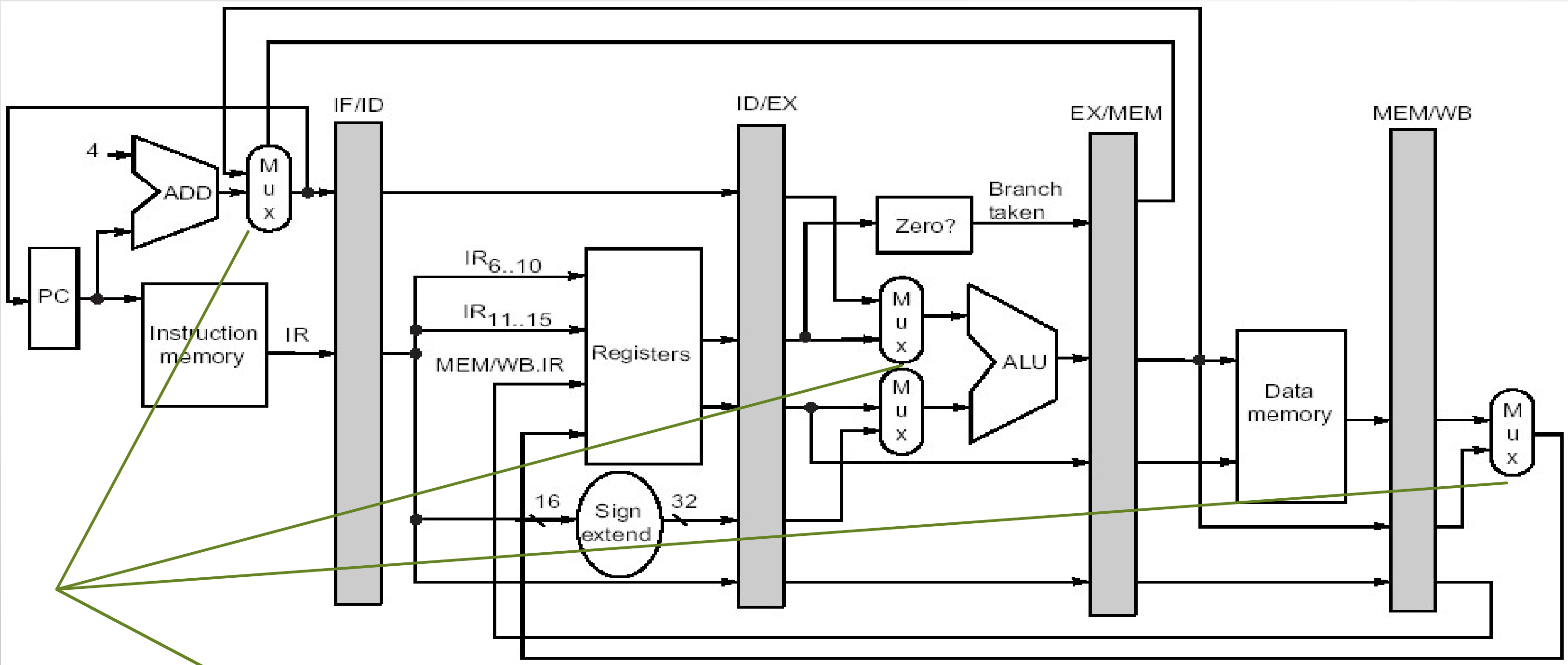
- Memory Access (MEM)
  - ALU Instruction
    - MEM/WB.IR ← EX/MEM.IR
    - MEM/WB.ALUOutput ← EX/MEM.ALUOutput
    - Save the contents of the ALU output to the next stage pipeline registers, for usage in WB stage.
    - Propagate the contents of IR to the next stage, for usage in the next clock cycle

Pipelined Instruction Write-Back

- Write-Back Cycle (WB)
  - ALU Instructions
    - For Register – Register ALU Instruction
      - Regs[MEM/WB.$IR_{16...20}$]← MEM/WB.ALUOutput
    - For Register –Immediate ALU Instruction
      - Regs[MEM/WB.$IR_{11...15}$] ← MEM/WB.ALUOutput
  - Memory Access (Load) Instruction
    - Regs[MEM/WB.$IR_{11..15}$] ← MEM/WB.LMD
  - Write the results back into the register file, whether the data comes from the main memory or as a result of an operation (from ALU); the register destination can be in two positions up to the instruction type

Control Path for Pipeline Processor

OÉ Gaillimh
NUI Galway



one

The forth multiplexer is controlled by whether the instruction in WB stage is a load or an ALU operation.

Performance Issues in Pipeline (1)

OÉ Gaillimh
NUI Galway

- Pipelining increases the processor throughput
  - Number of instructions completed per unit of time
- Pipelining does NOT increase the execution speed of individual instruction
  - In fact, it actually decreases the execution speed per individual instruction, due to the overhead introduced in the data path and control of pipeline
- The increase in the throughput means that a program runs faster and has lower total execution time, even if no single instruction runs faster

Performance Issues in Pipeline (2)

OÉ Gaillimh
NUI Galway

- There are limits on the physical limit on the pipeline, caused by:
  - Execution time of each instruction doesn’t decrease
  - Imbalance between pipeline stages
    - Reduces performance, since the clock can not run any faster than the time needed for the slowest pipeline stage
  - Pipeline overhead
    - Arises from the combination of pipeline register delay and clock skew

Performance Computation (1)

- Consider our example un-pipelined processor
  - The ALU operations and branches uses four cycles. The relative frequency of ALU operations is 40% and 20% for branches
  - The memory operations use five cycles. The relative frequency is 40 %
  - Clock cycle is 10ns
- Consider a 1ns overhead to the clock introduced by the pipeline
- How much speedup in the instruction execution rate will we gain from pipeline?

Performance Computation (2)

- The average instruction execution time for the un-pipelined machine is:
  - Clock Cycle * Average CPI (Clock cycles Per Instruction)
  - = 10 ns * [(40% + 20%) * 4 + 40% * 5] = 10 ns * 4.4 = 44 ns
- In pipeline implementation, the clock must run at the speed of the lowest pipeline segment plus the clock overhead, which would be 11ns

$$Speedup = \frac{AverageInstructionTimeUnpipelined}{AverageInstructionTimePipelined} = \frac{44ns}{11ns} = 4times$$

References

- “Computer Architecture – A Quantitative Approach”, John L Hennessy & David A Patterson, ISBN 1-55860-329-8
- “Computer Architecture”, Nicholas Charter, ISBN – 0-07-136207

- CT101 -
Computing Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Pipeline Hazard

- Introduction to pipeline hazard
- Structural Hazard
- Data Hazard
- Control Hazard

Pipeline Hazards (1)

- ***Pipeline Hazards*** are situations that prevent the next instruction in the instruction stream from executing in its designated clock cycle
- Hazards reduce the performance from the ideal speedup gained by pipelining
- Three types of hazards:
  - ***Structural hazards***
    - Arise from resource conflicts when the hardware can't support all possible combinations of overlapping instructions
  - ***Data hazards***
    - Arise when an instruction depends on the results of a previous instruction in a way that is exposed by overlapping of instruction in pipeline
  - ***Control hazards***
    - Arise from the pipelining of branches and other instructions that change the PC (Program Counter)

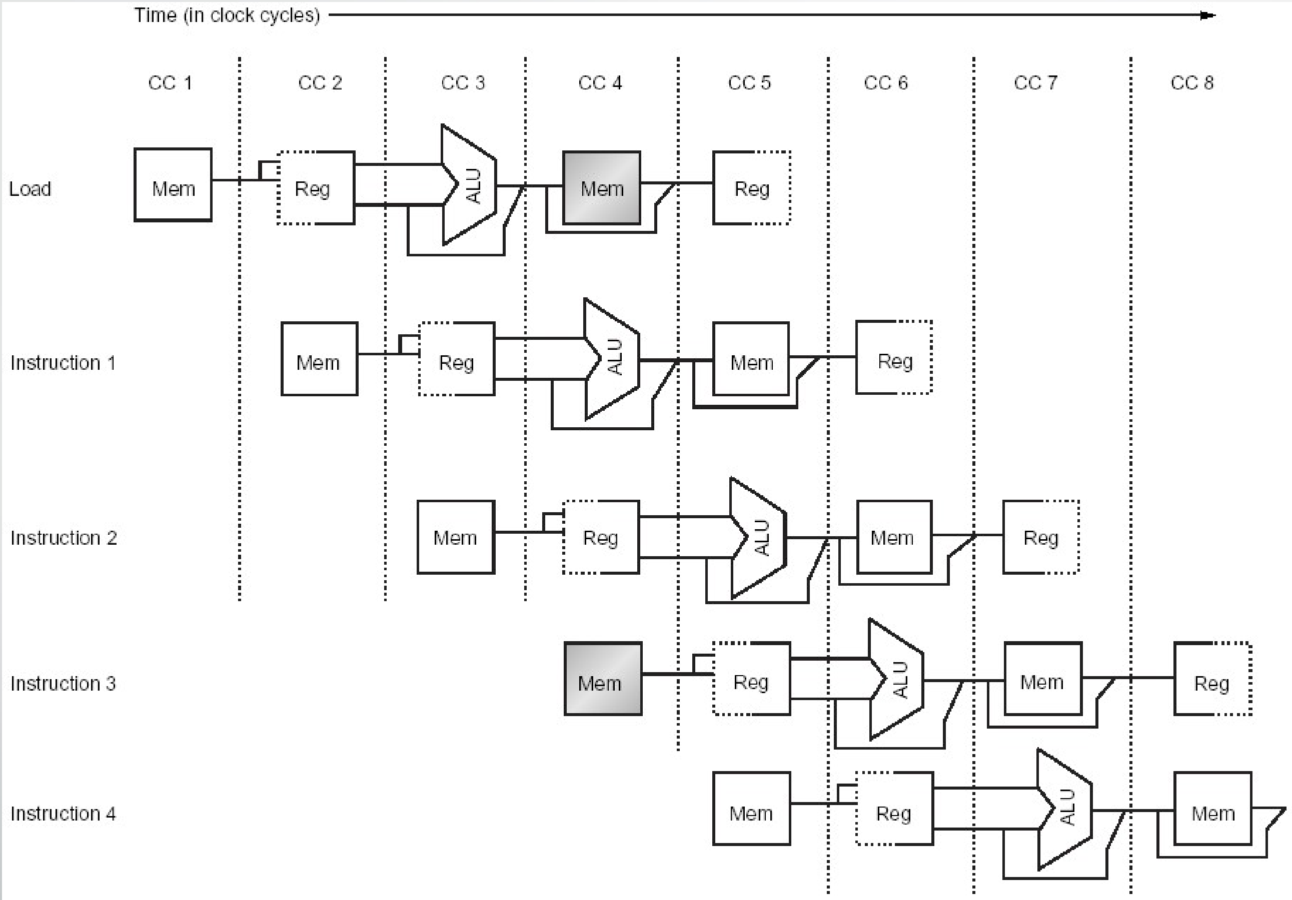
Pipeline Hazards (2)

- Hazards in pipeline can make the pipeline ***stall***
- Eliminating a hazard often requires that some instructions in the pipeline can proceed while others are delayed
  - When an instruction is stalled, instructions issued *later* than the stalled instruction are stopped, while the ones issued *earlier* must continue
- No new instructions are fetched during the stall

Structural Hazards (1)

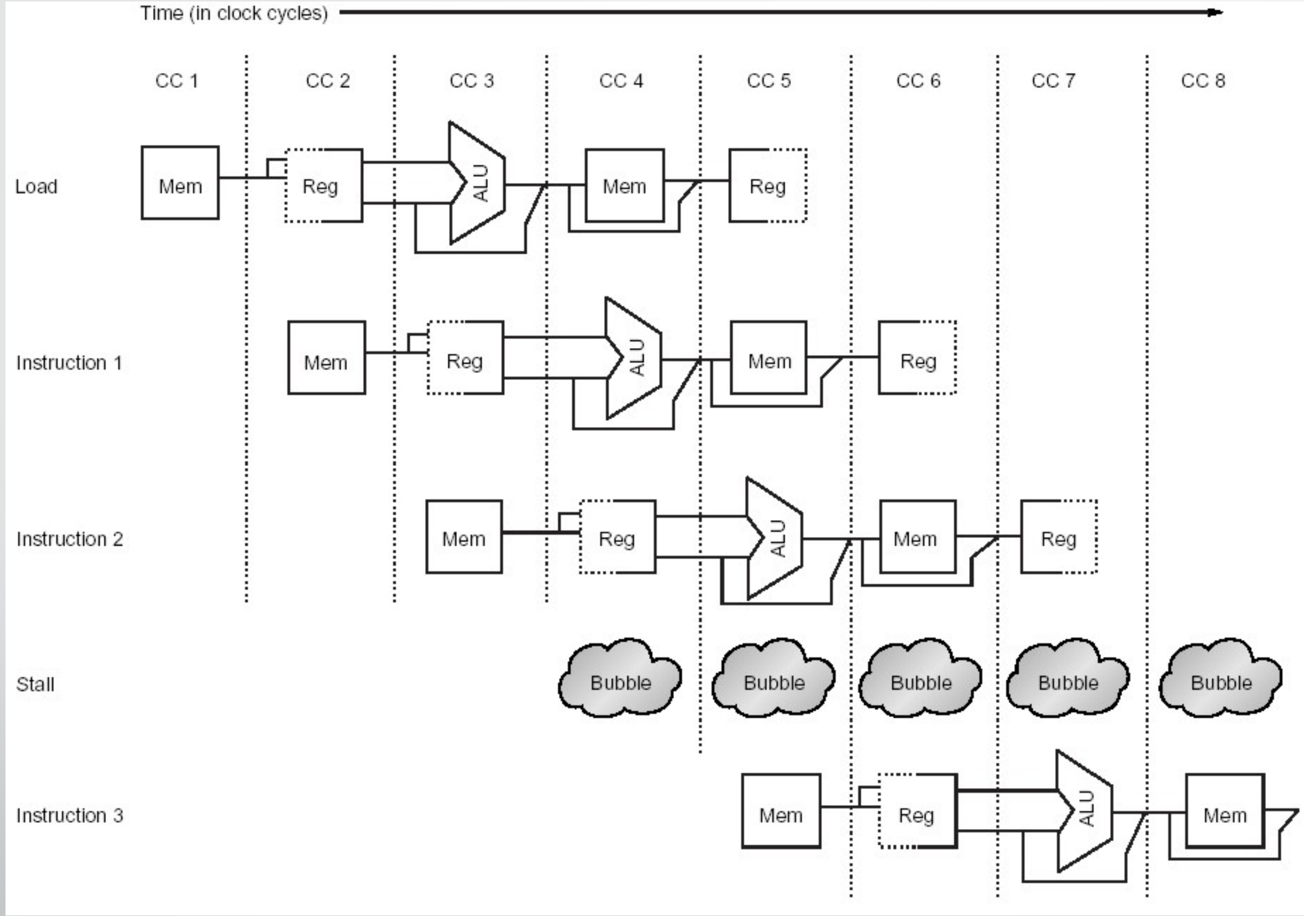
- If certain combination of instructions can't be accommodated because of **resource conflicts**, the machine is said to have a *structural hazard*
- It can be generated by:
  - Some functional unit is not fully pipelined
  - Some resources have not been duplicated enough to allow all the combinations in the pipeline to execute
  - For example: a machine may have only one register file write port, but under certain conditions, the pipeline might want to perform two writes in one clock cycle – this will generate structural hazard
    - When a sequence of instructions encounter this hazard, the pipeline will stall one of the instructions until the required unit is available
    - Such stalls will increase the Clock cycle Per Instruction from its ideal 1 for pipelined machines

Structural Hazards (2)



- Consider a Von Neumann architecture (same memory for instructions and data)

Structural Hazards (3)



- Stall cycle added (commonly called pipeline *bubble*)

Structural Hazards (4)

Instruction Number	Clock number									
	1	2	3	4	5	6	7	8	9	10
load	IF	ID	EX	MEM	WB					
Instruction i+1		IF	ID	EX	MEM	WB				
Instruction i+2			IF	ID	EX	MEM	WB			
Instruction i+3				stall	IF	ID	EX	MEM	WB	
Instruction i+4						IF	ID	EX	MEM	WB
Instruction i+5							IF	ID	EX	MEM

- Another way to *represent* the stall – no instruction is initiated in clock cycle 4

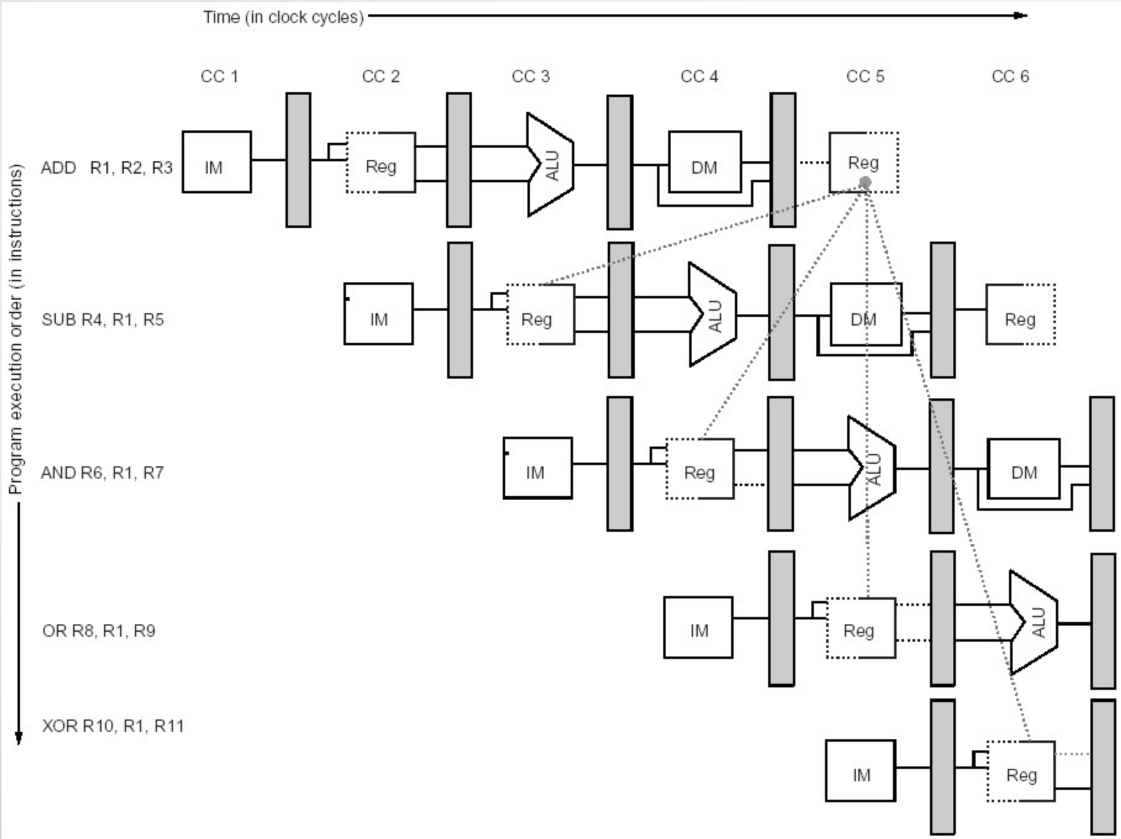
Structural Hazards (5)

- A machine with structural hazard will have lower CPI
- Why would a designer allow structural hazard?
  - To reduce cost
    - Pipelining all the functional units or duplicating them may be too costly
  - To reduce latency
    - Introducing too many pipeline stages may cause latency issues

Data Hazards (1)

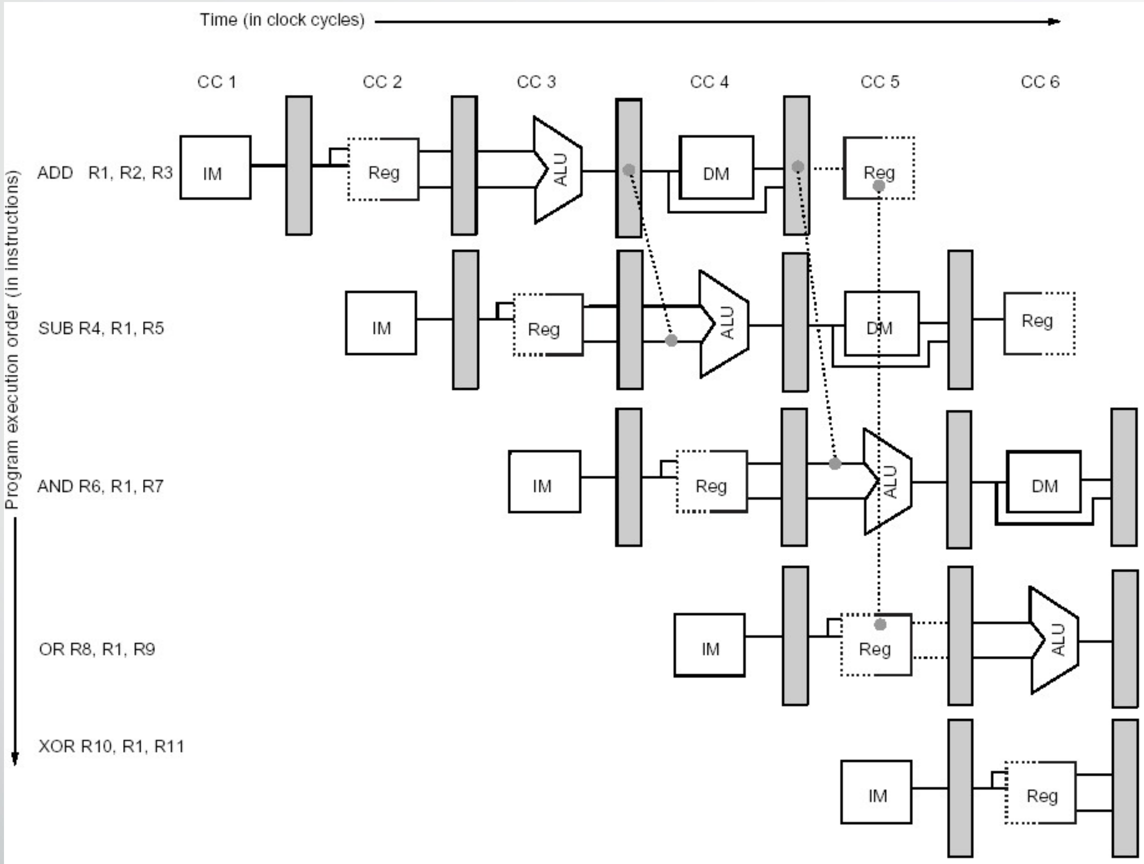
- Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an un-pipelined machine
- Consider the execution of following instructions, on our pipelined example processor:
  - ADD R1, R2, R3
  - SUB R4, R1, R5
  - AND R6, R1, R7
  - OR R8, R1, R9
  - XOR R10, R1, R11

Data Hazards (2)



- The use of results from ADD instruction causes hazard since the register is not written until after those instructions read it.

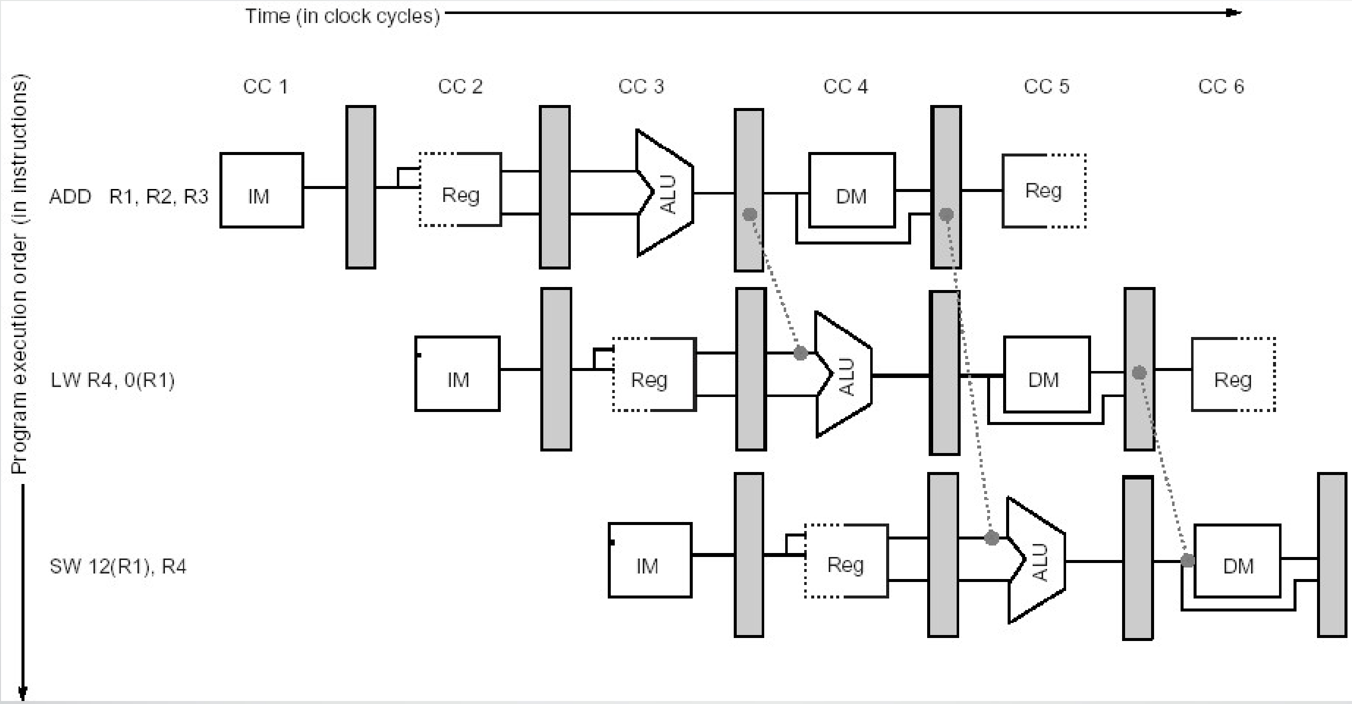
Data Hazards (3)



- Eliminate the stalls for the hazard involving SUB and AND instructions using a technique called ***forwarding***

School of Computer Science

Data Hazards (4)



- Store requires an operand during MEM and forwarding is shown here.
  - The result of the load is forwarded from the output in MEM/WB to the memory input to be stored
  - In addition the ALUOutput is forwarded to ALU input for address calculation for both Load and Store

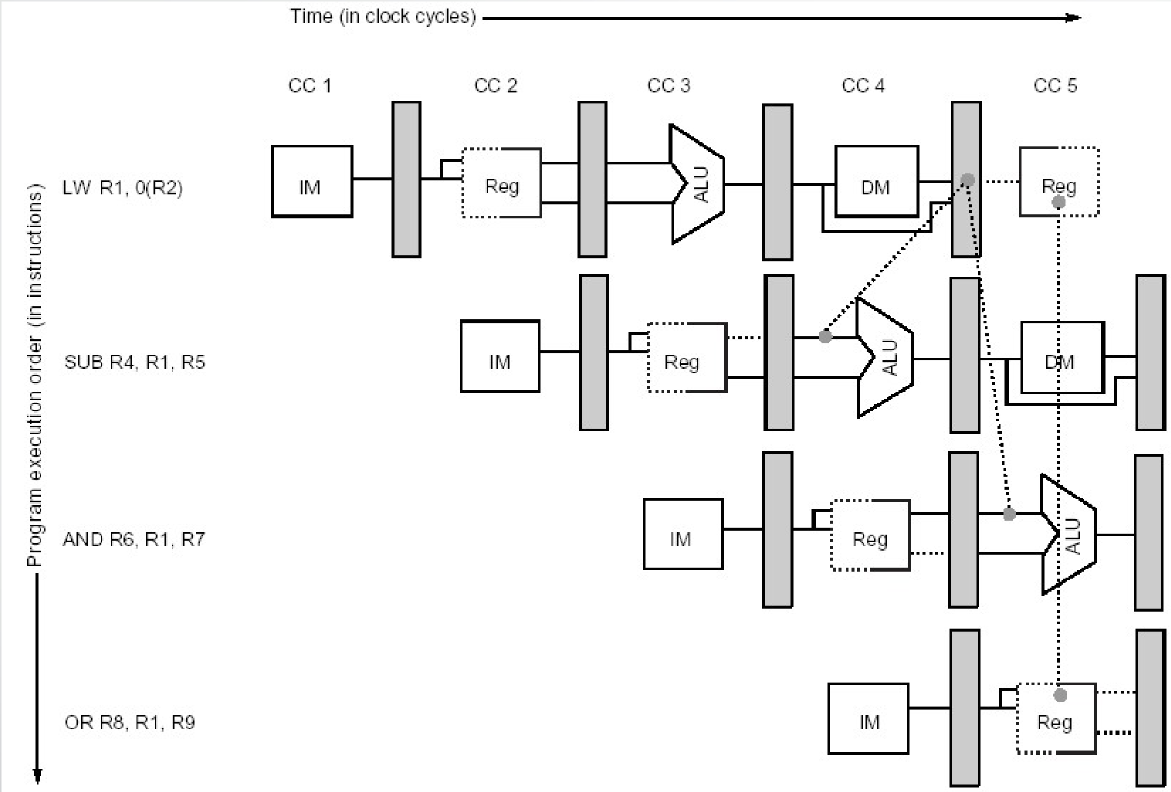
Data Hazards Classification

- Depending on the order of read and write access in the instructions, data hazards could be classified as three types.
- Consider two instructions i and j, with i occurring before j.
- Possible data hazards:
  - **RAW** (Read After Write)
    - j tries to read a source before i writes to it , so j incorrectly gets the old value;
    - most common type of hazard, that is what we tried to explain so far.
  - **WAW** (Write After Write)
    - j tries to write an operand before it is written by i. The write ends up being performed in wrong order, having i overwrite the operand written by j, the destination containing the operand written by i rather than the one written by j
    - Present in pipelines that write in more than one pipe stage
  - **WAR** (Write After Read)
    - j tries to write a destination before it is read by i, so the instruction i incorrectly gets the new value
    - This doesn't happen in our example, since all reads are early and writes late

Data Hazards Requiring Stalls (1)

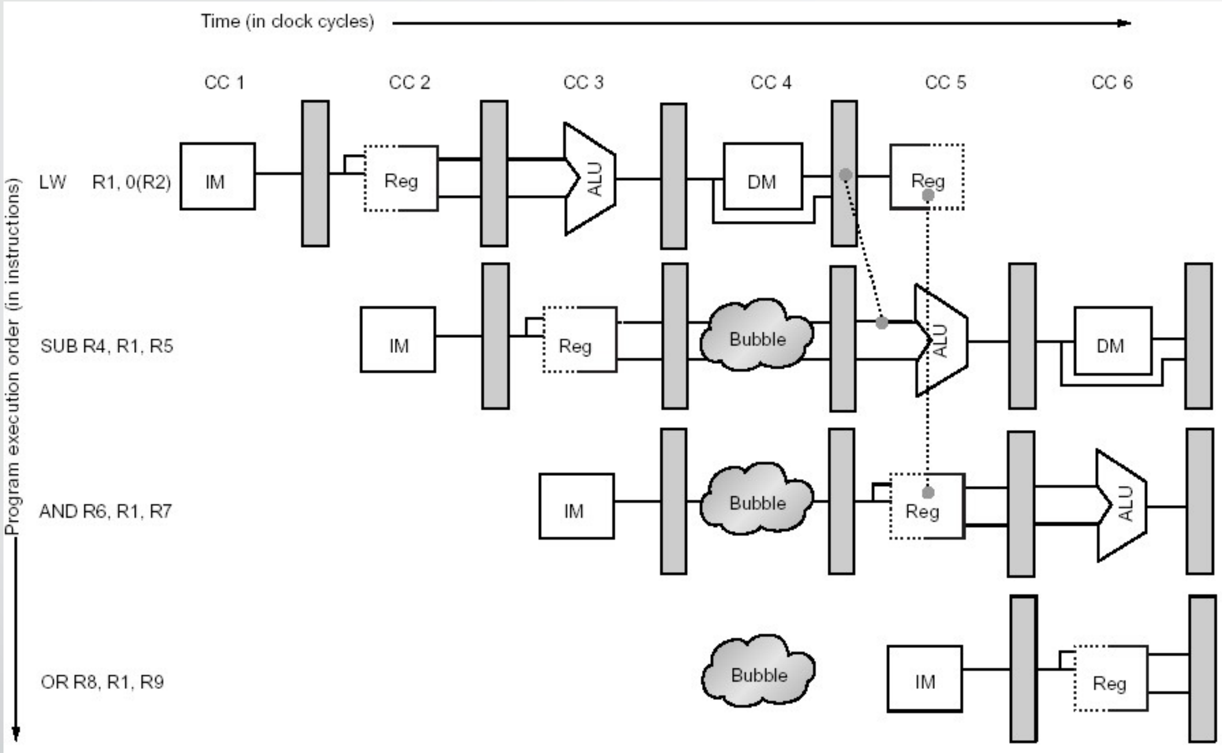
- Unfortunately not all data hazards can be handled by forwarding. Consider the following sequence:
  - LW R1, 0(R2)
  - SUB R4, R1, R5
  - AND R6, R1, R7
  - OR R8, R1, R9
- The problem with this sequence is that the Load operation will not have data until the end of MEM stage.

Data Hazards Requiring Stalls (2)



- The load instruction can forward the results to AND and OR instruction, but not to the SUB instruction since that would mean forwarding results in “negative” time

Data Hazards Requiring Stalls (3)



- The load interlock causes a stall to be inserted at clock cycle 4, delaying the SUB instruction and those that follow by one cycle.
  - This delay allows the value to be successfully forwarded onto the next clock cycle

Data Hazards Requiring Stalls (4)

LW R1, 0(R2)	IF	ID	EX	MEM	WB					
SUB R4, R1, R5		IF	ID	EX	MEM	WB				
AND R6, R1, R7			IF	ID	EX	MEM	WB			
OR R8, R1, R9				IF	ID	EX	MEM	WB		

- Before stall insertion

LW R1, 0(R2)	IF	ID	EX	MEM	WB					
SUB R4, R1, R5		IF	ID	stall	EX	MEM	WB			
AND R6, R1, R7			IF	stall	ID	EX	MEM	WB		
OR R8, R1, R9				stall	IF	ID	EX	MEM	WB	

- After stall insertion

Compiler Scheduling for Data Hazards (1)

OÉ Gaillimh
NUI Galway

- Consider typical code, such as A = B+C

LW R1, B	IF	ID	EX	MEM	WB					
LW R2, C		IF	ID	EX	MEM	WB				
ADD R3, R1, R2			IF	ID	stall	EX	MEM	WB		
SW A, R3				IF	stall	ID	EX	MEM	WB	

- The ADD instruction must be stalled to allow the load of C to complete
- The SW needs not be delayed because the forwarding hardware passes the result from MEM/WB directly to the data memory input for storing

Compiler Scheduling for Data Hazards (2)

- Rather than just allow the pipeline to stall, the compiler could try to schedule the pipeline to avoid the stalls, by rearranging the code
  - The compiler could try to avoid the generating the code with a load followed by an immediate use of the load destination register
  - This technique is called ***pipeline scheduling*** or ***instruction scheduling*** and it is a very often used technique in modern compilers

Instruction scheduling example

- Generate code for our example processor that avoids pipeline stalls from the following sequence:
  - A = B +C
  - D = E - F
- Solution
  - LW Rb, B
  - LW Rc, C
  - LW Re, E ; swap instructions to avoid stall
  - ADD Ra, Rb, Rc
  - LW Rf, f
  - SW a, Ra ; store/load exchanged to avoid stall
  - SUB Rd, Re, Rf
  - SW d, Rd

Control Hazards (1)

- Can cause a greater performance loss than that of data hazards
- When a branch is executed it may or it may not change the PC (to other value than its value + 4)
  - If a branch is changing the PC to its target address, then it is a ***taken*** branch
  - If a branch doesn't change the PC to its target address, then it is a ***not taken*** branch
- If instruction i is a taken branch, then the value of PC will not change until the end MEM stage of the instruction execution in the pipeline
  - A simple method to deal with branches is to stall the pipe as soon as we detect a branch until we know the result of the branch

Control Hazards (2)

Branch Instruction	IF	ID	EX	MEM	WB					
Branch Successor		IF	stall	stall	IF	ID	EX	MEM	WB	
Branch Successor +1						IF	ID	EX	MEM	WB
Branch Successor +2							IF	ID	EX	MEM

- A branch causes three cycle stall in our example processor pipeline
  - One cycle is a repeated IF – necessary if the branch would be taken. If the branch is not taken, this IF is redundant
  - Two idle cycles

Control Hazards (3)

- The three clock cycles lost for every branch is a significant loss
  - With a 30% branch frequency, the machine with branch stalls achieves only about half of the speedup from pipelining
  - ***Reducing the branch penalty becomes critical***
- The number of clock cycles in a branch stall can be reduced by two steps:
  - Find out if the branch is taken or not in early stage in the pipeline
  - Compute the taken PC (address of the branch target) earlier

References

- “Computer Architecture – A Quantitative Approach”, John L Hennessy & David A Patterson, ISBN 1-55860-329-8
- “Computer Architecture”, Nicholas Charter, ISBN – 0-07-136207

- CT101 -
Computing Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

Memory Subsystem

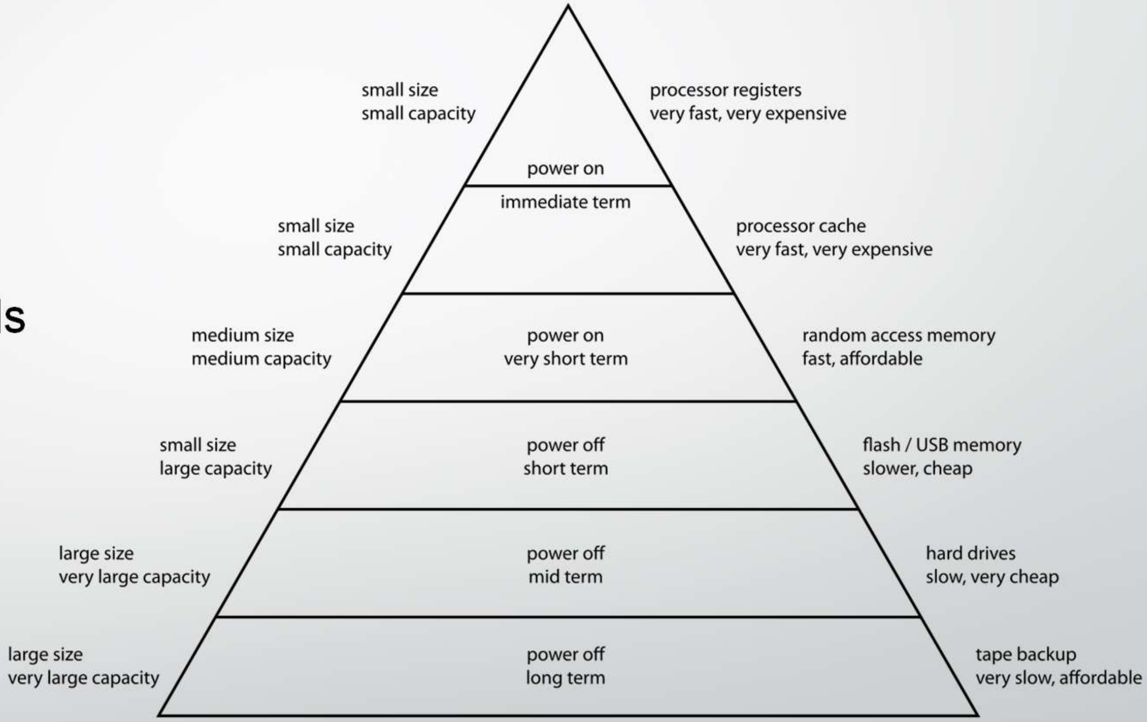
OÉ Gaillimh
NUI Galway

- Memory Hierarchy
- Types of memory
- Memory organization
- Memory Hierarchy Design
- Cache

Memory Hierarchy

OÉ Gaillimh
NUI Galway

- Registers
  - In CPU
- Internal or Main memory
  - May include one or more levels of cache
  - “RAM”
- External memory
  - Backing store



[wikipedia.org]

Internal Memory Types

<table>
<tr><th>Memory Type</th><th>Category</th><th>Erasure</th><th>Write Mechanism</th><th>Volatility</th></tr>
<tr><td>Random-access memory (RAM)</td><td>Read-write memory</td><td>Electrically, byte-level</td><td>Electrically</td><td>Volatile</td></tr>
<tr><td>Read-only memory (ROM)</td><td rowspan="2">Read-only memory</td><td rowspan="2">Not possible</td><td>Masks</td><td rowspan="5">Nonvolatile</td></tr>
<tr><td>Programmable ROM (PROM)</td><td rowspan="4">Electrically</td></tr>
<tr><td>Erasable PROM (EPROM)</td><td rowspan="3">Read-mostly memory</td><td>UV light, chip-level</td></tr>
<tr><td>Electrically Erasable PROM (EEPROM)</td><td>Electrically, byte-level</td></tr>
<tr><td>Flash memory</td><td>Electrically, block-level</td></tr>
</table>

External Memory Types

- HDD
  - Magnetic Disk(s)
- SSD (Solid State Drive(s))
- Optical
  - CD-ROM
  - CD-Recordable (CD-R)
  - CD-R/W
  - DVD
- Magnetic Tape

Random Access Memory (RAM)

- Read/Write
- Volatile
- Temporary storage
- Static or dynamic

Types of RAM

- Dynamic RAM (**DRAM**) – are like leaky capacitors; initially data is stored in the DRAM chip, charging its memory cells to maximum values. The charge slowly leaks out and eventually would go too low to represent valid data; before this happens, a **refresh** circuitry reads the contents of the DRAM and rewrites the data to its original locations, thus restoring the memory cells to their maximum charges

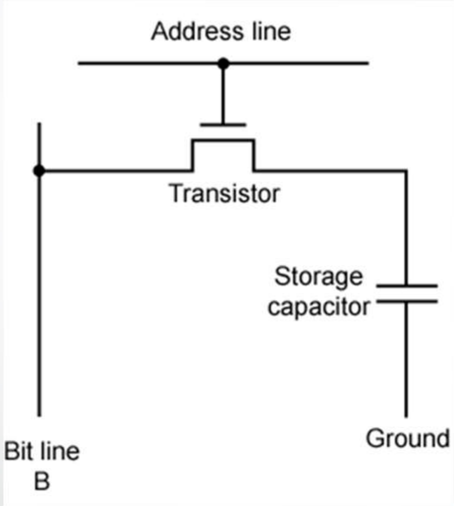
- Static RAM (**SRAM**) – is more like a register; once the data has been written, it will stay valid, it doesn't have to be refreshed. Static RAM is faster than DRAM, also more expensive. Cache memory in PCs is constructed from SRAM memory.

Dynamic RAM

- Bits stored as charge in capacitors
  - Charges leak
  - Need refreshing even when powered
- Simpler construction
- Smaller per bit than SRAM
  - Less expensive
- Need refresh circuits
- Slower
- Used for **main** memory in computing systems
- Essentially analogue
  - Level of charge determines value

DRAM Structure & Operation

- Address line is active when bit read or written
  - Transistor switch closed (current flows)
- Write
  - Voltage to bit line
    - High for 1 low for 0
  - Then signal address line
    - Transfers charge to capacitor
- Read
  - Address line selected
    - Transistor turns on
  - Charge from capacitor fed via bit line to sense amplifier
    - Compares with reference value to determine 0 or 1
  - Capacitor charge must be restored



DRAM Refreshing

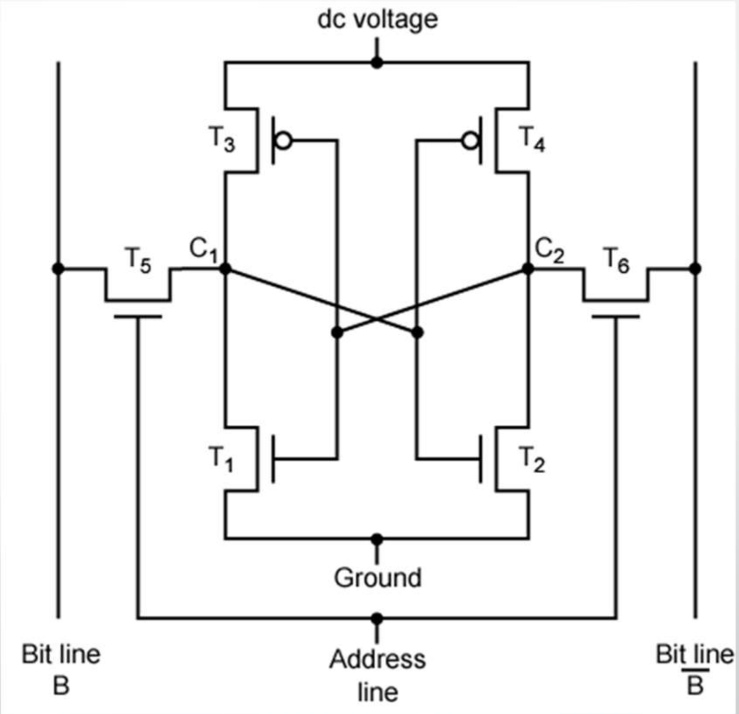
- Refresh circuit included on chip
  - Disable memory array chip
  - Count through rows and select each in turn
  - Read contents & write it back (restore)
- Takes time
- Slows down apparent performance

Static RAM

- Bits stored as on/off switches
- No charges to leak
- No refreshing needed when powered
- More complex construction
- Larger per bit
  - More expensive
- Faster
  - Cache

Static RAM Structure & Operation

- Transistor arrangement gives stable logic state
- State 1
  - C_1 high, C_2 low
  - T_1 T_4 off, T_2 T_3 on
- State 0
  - C_2 high, C_1 low
  - T_2 T_3 off, T_1 T_4 on
- Address line transistors T_5 T_6 is switch
- Write – apply value to B & compliment to $\overline{B}$
- Read – value is on line B



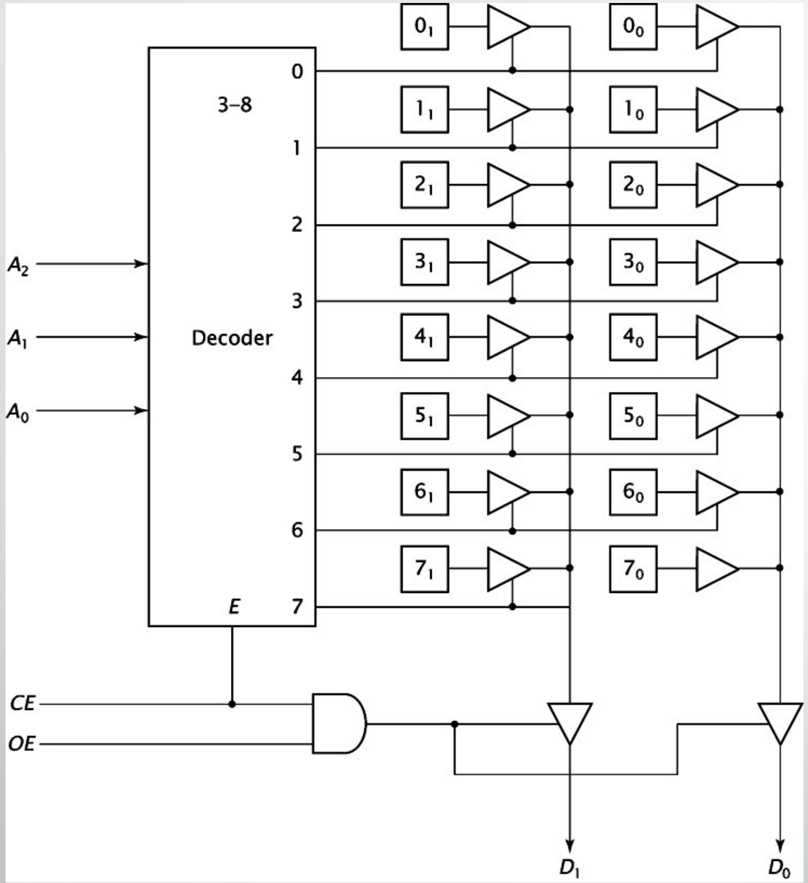
SRAM v DRAM

- Both volatile
  - Power needed to preserve data
- Dynamic cell
  - Simpler to build, smaller
  - More dense
  - Less expensive
  - Needs refresh
  - Larger memory units
- Static
  - Faster
  - Cache

Read Only Memory (ROM)

- Provides permanent storage (non-volatile)
- Used for: microprogramming, library subroutines (code) and constant data, systems programs (BIOS for PC or entire application + OS for certain embedded systems)
- Types
  - Written during manufacture (very expensive for small runs)
  - Programmable (once) PROM (needs special equipment to program)
  - Read “mostly”
    - Erasable Programmable (EPROM) - Erased by UV
    - Electrically Erasable (EEPROM) - Takes much longer to write than read
    - Flash memory - Erase whole memory electrically

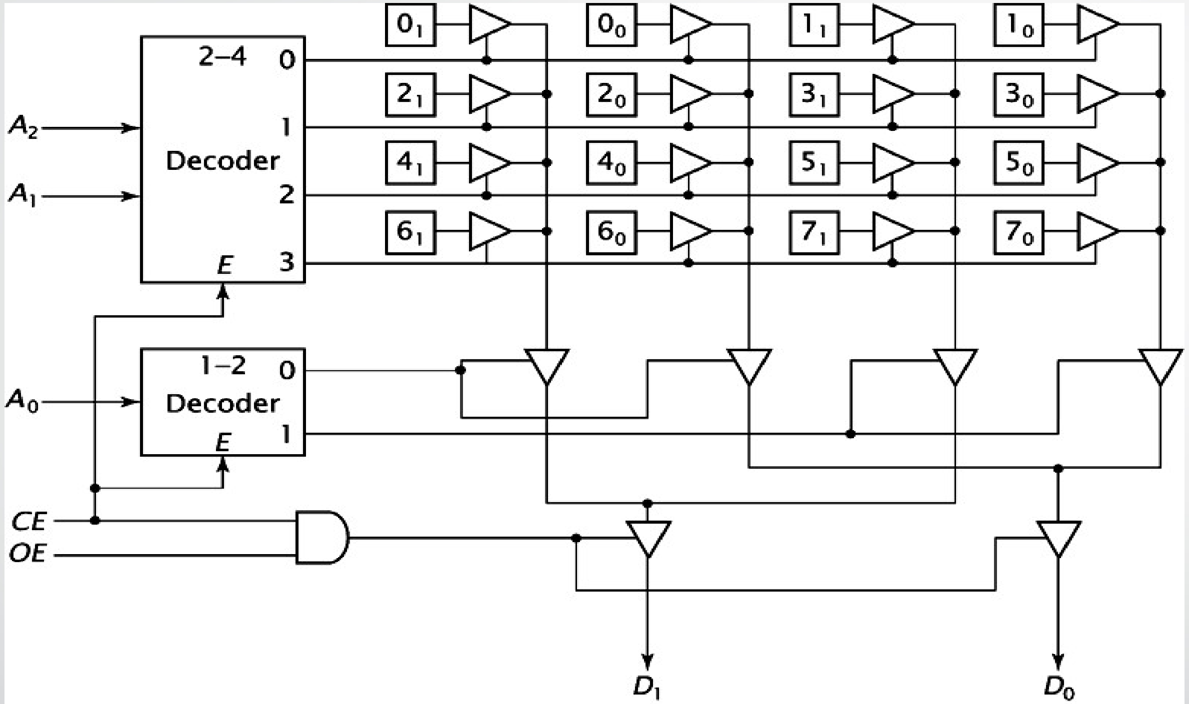
Internal Linear Organization



- 8X2 ROM chip
- As the number of locations increases, the size of the address decoder needed, becomes very large
- Multiple dimensions of decoding can be used to overcome this problem

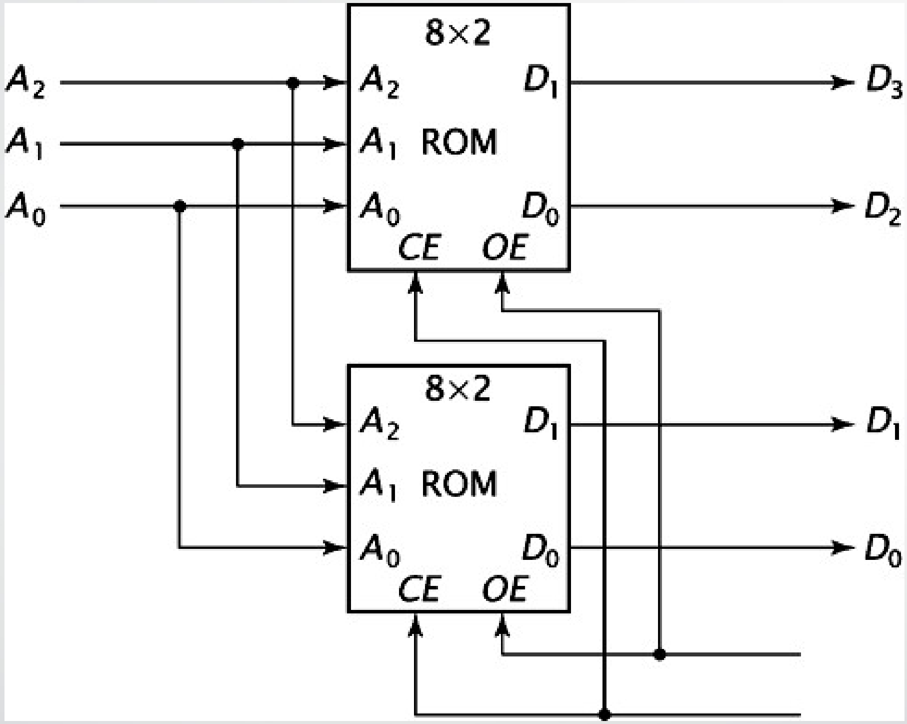
Internal Two-dimensional Organization

OÉ Gaillimh
NUI Galway



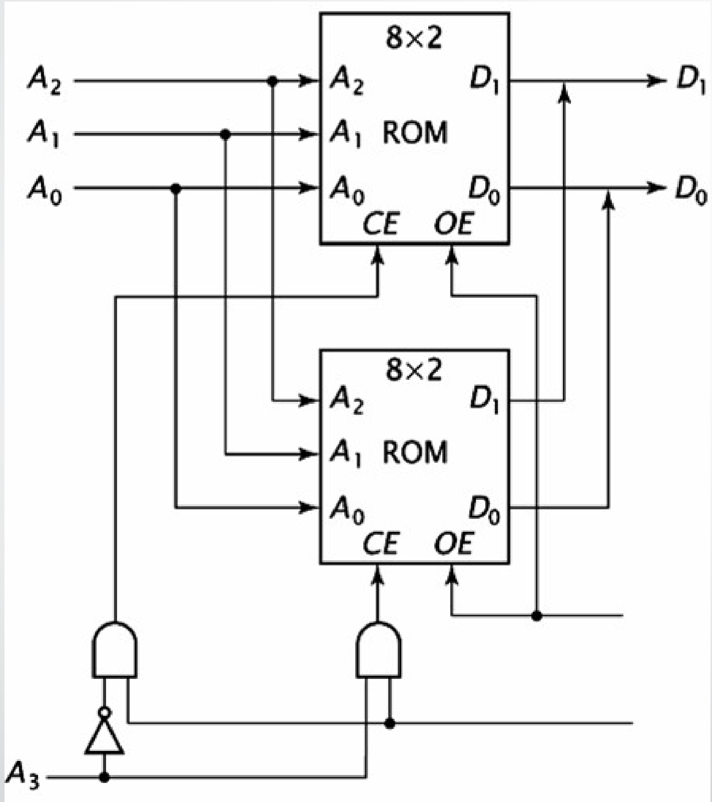
- High order address bits (A2A1) select one of the rows
- The low order address bit selects one of the two locations in the row

Memory Subsystems Organization (1)



- Two or more memory chips can be combined to create memory with **more bits per location** (two 8X2 chips can create a 8X4 memory)

Memory Subsystems Organization (2)



- Two or more memory chips can be combined to create **more locations** (two 8X2 chips can create 16X2 memory)

Memory Hierarchy Design (1)

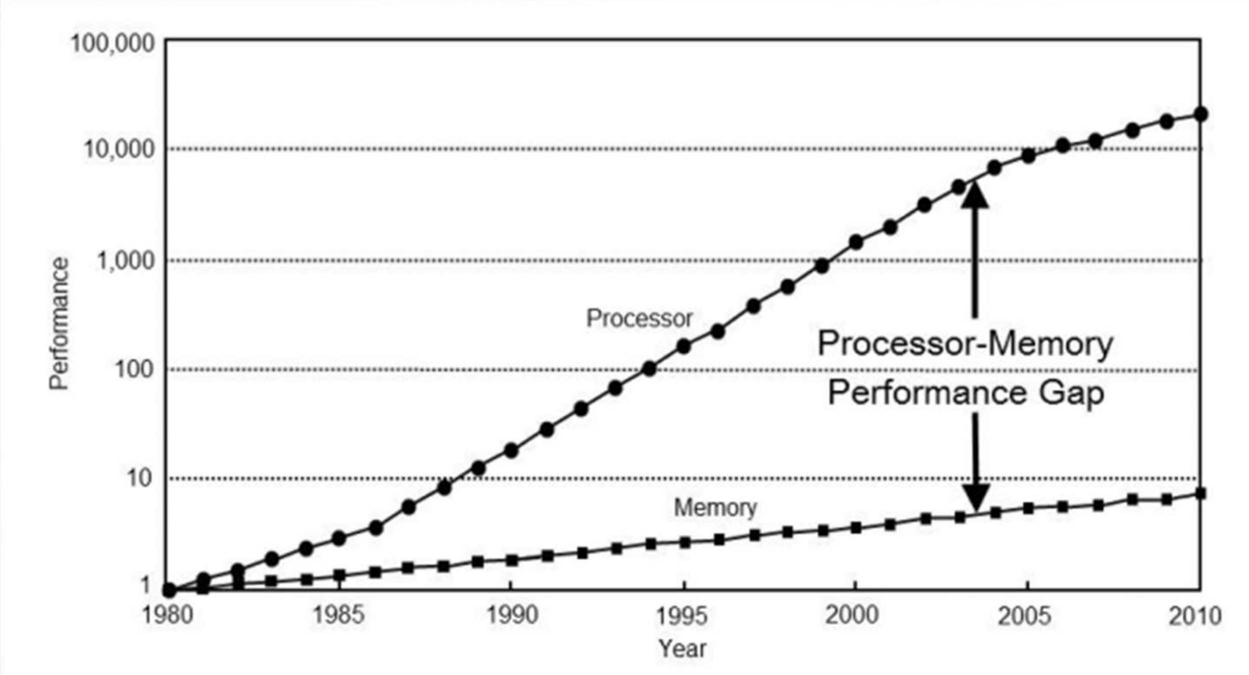
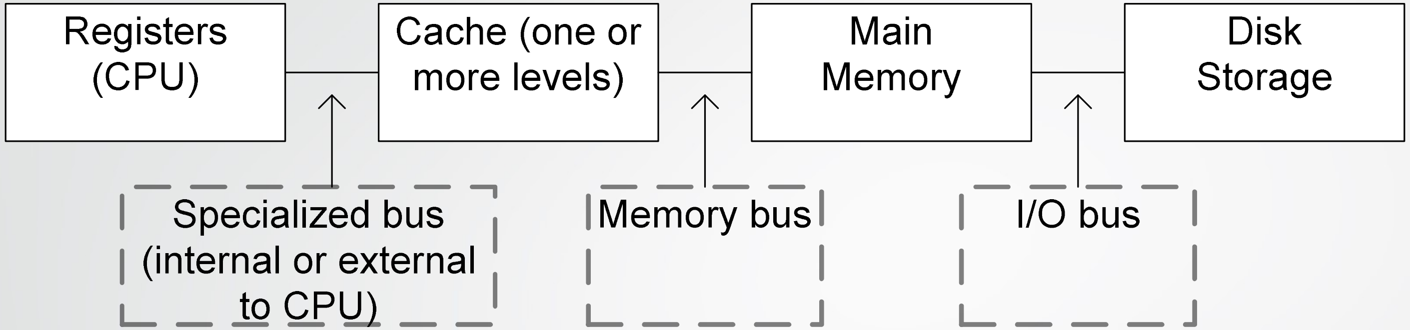


Image: https://www.extremetech.com/computing/261792-what-is-speculative-execution

- This picture shows the CPU performance against memory access time improvements over the years
  - Clearly there is a processor-memory performance gap that computer architects must take care of

Memory Hierarchy Design (2)



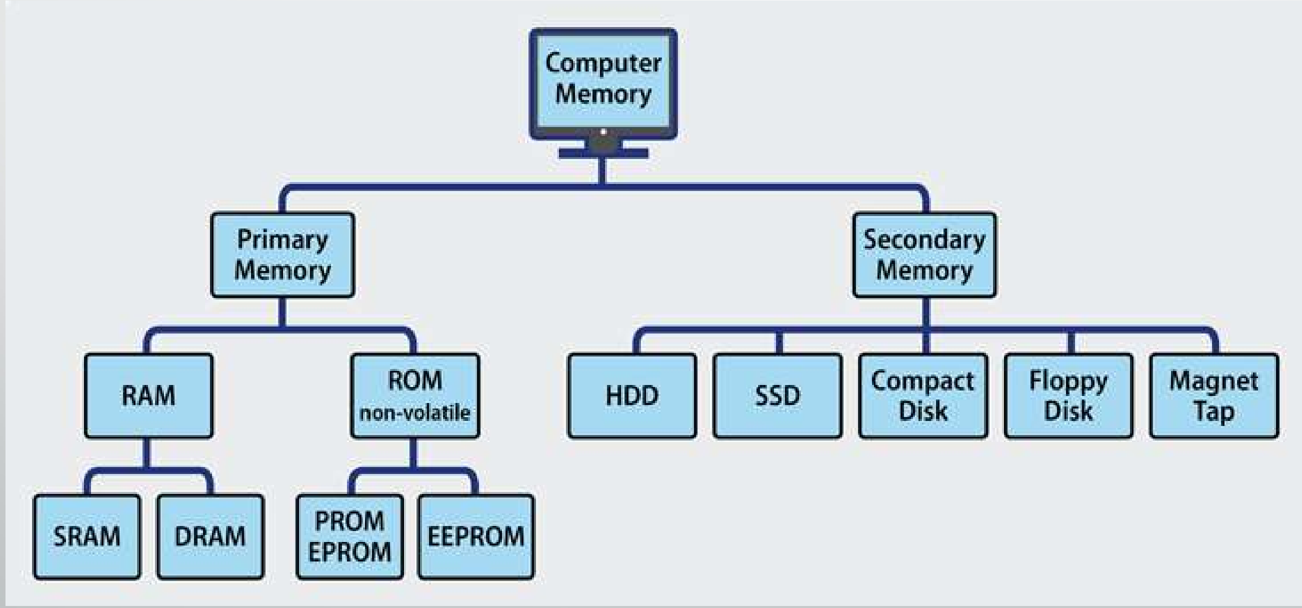
- It is a tradeoff between size, speed and cost and exploits the principle of locality.
- Register
  - Fastest memory element; but small storage; very expensive
- Cache
  - Fast and small compared to main memory; acts as a buffer between the CPU and main memory: it contains the most recent used memory locations (address and contents are recorded here)
- Main memory is the RAM of the system
- Disk storage - HDD

Memory Hierarchy Design (3)

- Comparison between different types of memory

Register Cache Memory HDD

larger, slower, cheaper



https://www.enterprisestorageforum.com/storage-hardware/types-of-computer-memory.html

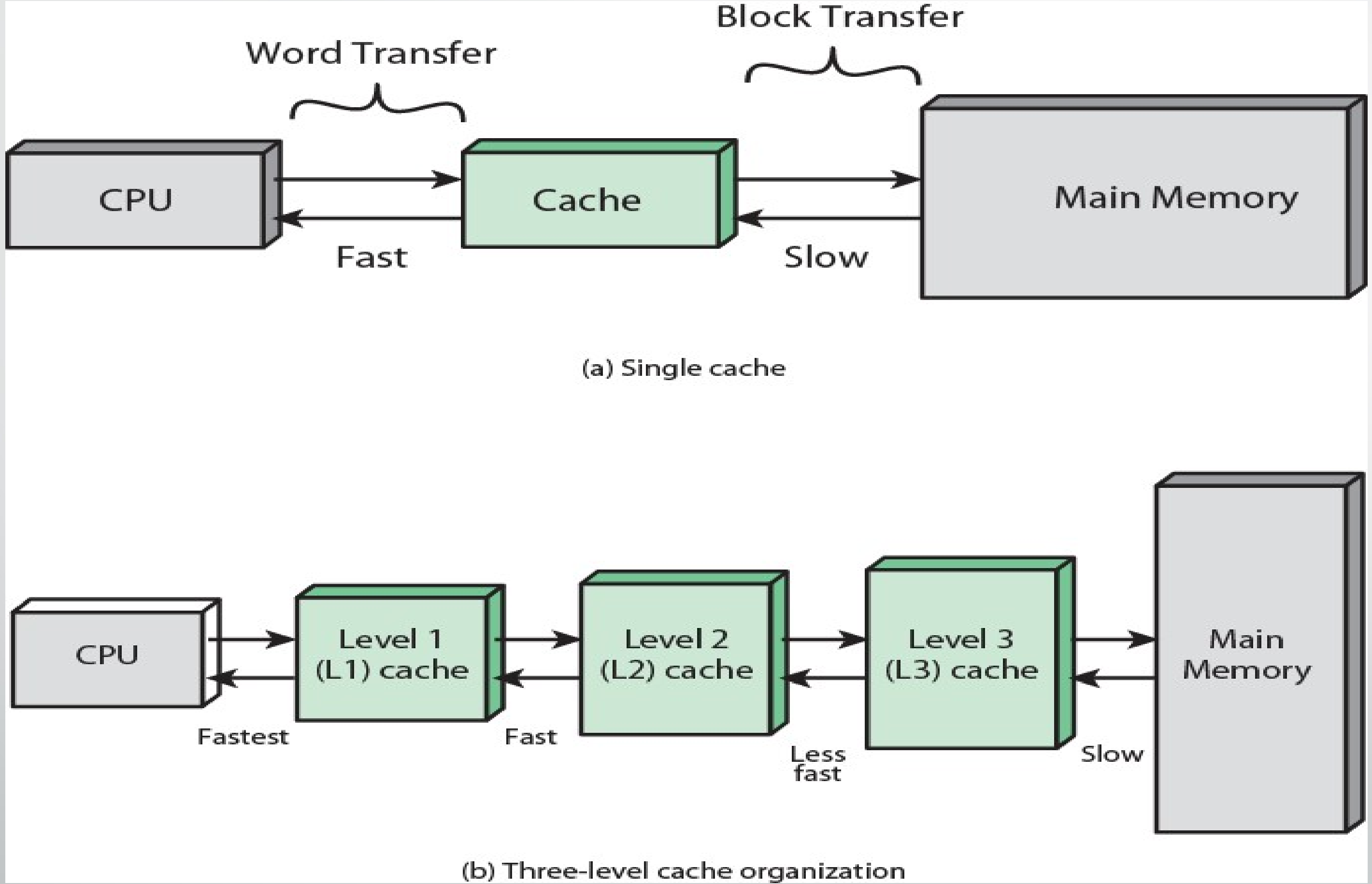
Cache (1)

- Is the first level of memory hierarchy encountered once the address leaves the CPU
  - Since the principle of locality applies, and taking advantage of locality to improve performance is so popular, the term ***cache*** is now applied whenever buffering is employed to reuse commonly occurring items
- We will study caches by trying to answer the four questions for the first level of the memory hierarchy

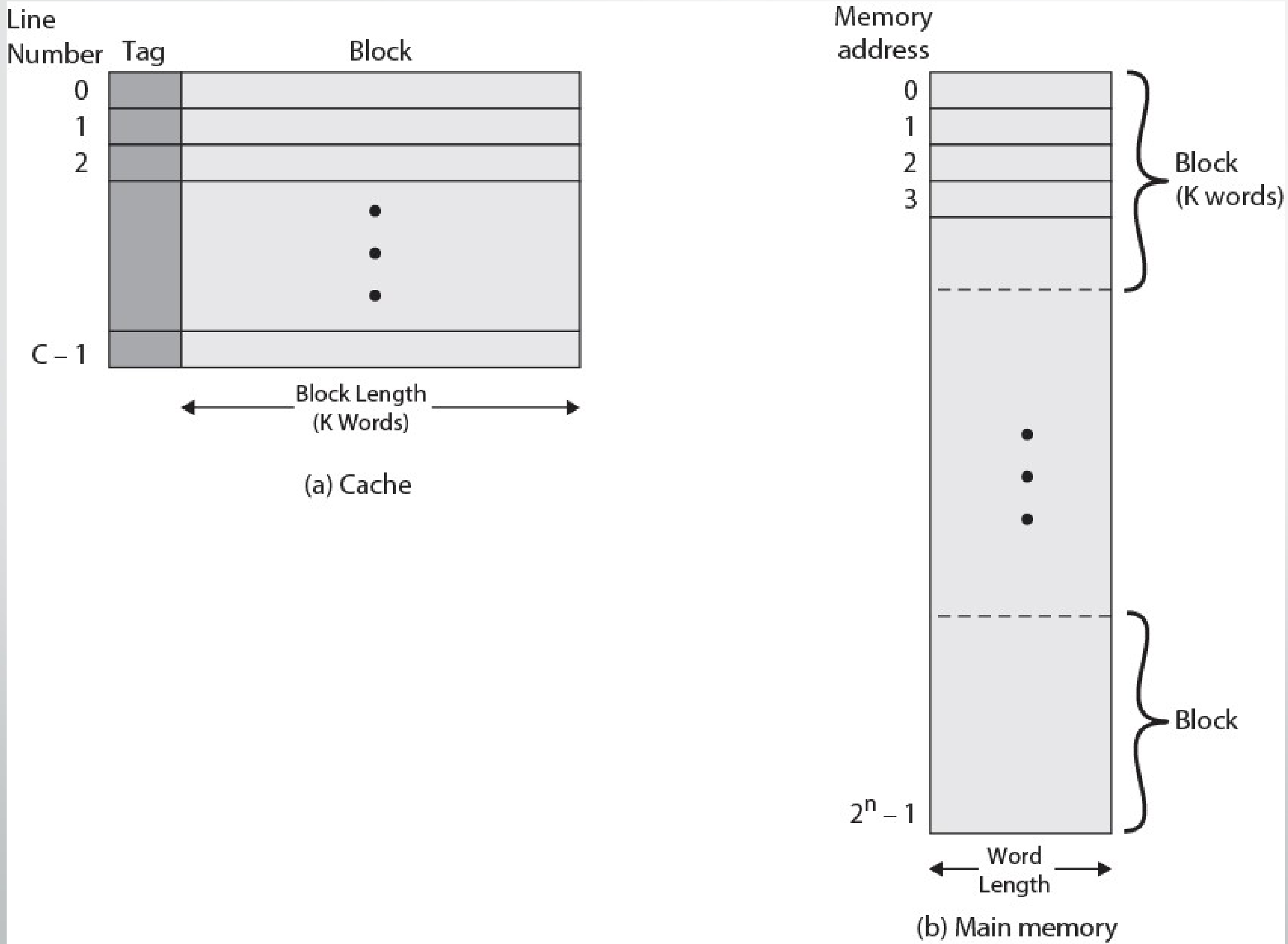
Cache (2)

- Every address reference goes first to the cache;
  - If the desired address is not here, then we have a ***cache miss***; The contents are fetched from main memory into the indicated CPU register and the content is also saved into the cache memory
  - If the desired data is in the cache, then we have a ***cache hit;*** The desired data is brought from the cache, at very high speed (low access time)
- Most software exhibits ***temporal locality*** of access, meaning that it is likely that same address will be used again soon, and if so, the address will be found in the cache
- Transfers between main memory and cache occur at granularity of ***cache lines*** or ***cache blocks***, around 32 or 64 bytes (rather than bytes or processor words). Burst transfers of this kind receive hardware support and exploit ***spatial locality*** of access to the cache (future access are often to address near to the previous one)

Cache Organization



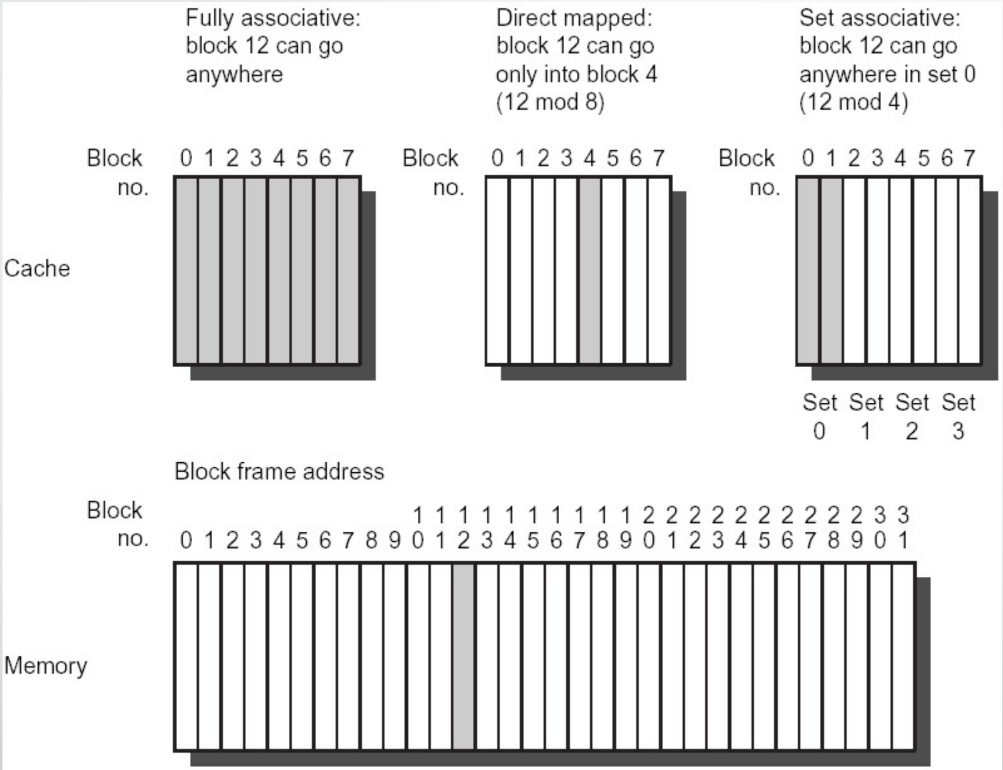
Cache/Main Memory Structure



Memory Hierarchy Design

- Where can a block be placed in the upper level?
  - BLOCK PLACEMENT
- How is a block found if it is in the upper level?
  - BLOCK IDENTIFICATION
- Which block should be replaced on a miss?
  - BLOCK REPLACEMENT
- What happens on a write?
  - WRITE STRATEGY

Where can a block be placed in Cache? (1)



- Our cache has 8 block frames and the main memory has 32 blocks

School of Computer Science

Where can a block be placed in Cache? (2)

- Direct mapped Cache
  - Each block has only one place where it can appear in the cache
  - *(Block Address) MOD (Number of blocks in cache)*

- Fully associative Cache
  - A block can be placed anywhere in the cache

- Set associative Cache
  - A block can be placed in a restricted set of places into the cache
  - A *set* is a group of blocks into the cache
  - *(Block Address) MOD (Number of sets in the cache)*
    - If there are n blocks in the cache, the placement is said to be *n-way set associative*

How is a Block Found in the Cache?

- Caches have an address tag on each block frame that gives the block address. The tag is checked against the address coming from CPU
  - All tags are searched in parallel since speed is critical
  - *Valid bit* is appended to every tag to say whether this entry contains valid addresses or not
- Address fields:
  - Block address
    - Tag – compared against for a hit
    - Index – selects the set
  - Block offset – selects the desired data from the block
- Set associative cache
  - Large index means large sets with few blocks per set
  - With smaller index, the associativity increases
- Full associative cache – index field does not exist

Block address		Block offset
Tag	Index	

Which Block should be Replaced on a Cache Miss?

OÉ Gaillimh
NUI Galway

- When a miss occurs, the cache controller must select a block to be replaced with the desired data
  - Benefit of direct mapping is that the hardware decision is much simplified

- Two primary strategies for full and set associative caches
  - ***Random*** – candidate blocks are randomly selected
    - Some systems generate pseudo random block numbers, to get reproducible behavior useful for debugging
  - ***LRU (Least Recently Used)*** – to reduce the chance that information that has been recently used will be needed again, the block replaced is the least-recently used one.
    - Accesses to blocks are recorded to be able to implement LRU

What Happens on a Write?

- Two basic options when writing to the cache:
  - Write through – the information is written to both, the block in the cache and the block in the lower-level memory
  - Write back – the information is written only to the cache
    - The modified block of cache is written back into the lower-level memory only when it is replaced

- To reduce the frequency of writing back blocks on replacement, an implementation feature called ***dirty bit*** is commonly used.
  - This bit indicates whether a block is ***dirty*** (has been modified since loaded) or ***clean*** (not modified). If clean, no write back is involved

References

- “Computer Architecture – A Quantitative Approach”, John L Hennessy & David A Patterson, ISBN 1-55860-329-8
- “Computer Systems Organization & Architecture”, John D. Carpinelli, ISBN: 0-201-61253-4
- “Computer Organization and Architecture”, William Stallings, 8th Edition

OÉ Gaillimh
NUI Galway

- CT101 -
Computing Systems

Dr. Frank Glavin

Room 404, IT Building

Frank.Glavin@NUIGalway.ie

School of Computer Science,

National University of Ireland, Galway

I/O Subsystem

OÉ Gaillimh
NUI Galway

- Overview
- Peripheral Devices and I/O Modules
- Programmed I/O
- Interrupt Driven I/O
- Direct Memory Access

Overview

- I/O devices are very different (i.e. keyboard and HDD performs totally different functions, yet they are both part of the I/O subsystem).
- The interfaces between the CPU and I/O devices are very similar.
- Each I/O device needs to be connected to:
  - Address bus – to pass address to peripheral
  - Data bus – to pass data to and from peripheral
  - Control bus – to pass control signals to peripherals

Problems

OÉ Gaillimh
NUI Galway

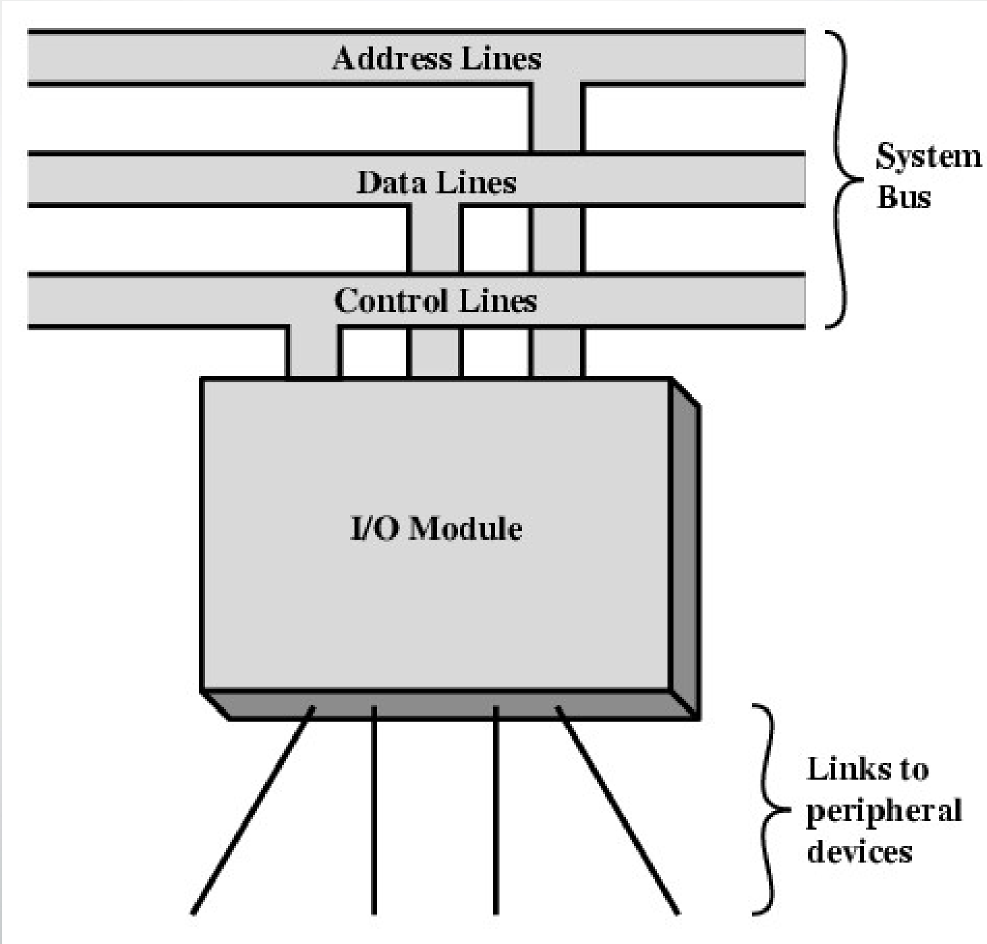
- Wide variety of peripherals
  - Delivering different amounts of data
  - At different speeds
  - In different formats

- All slower than CPU and RAM

- Need **I/O modules**
  - Interface with the processor and memory via system buses or central switch
  - Interface to one or more peripheral devices using specific data links/interfaces

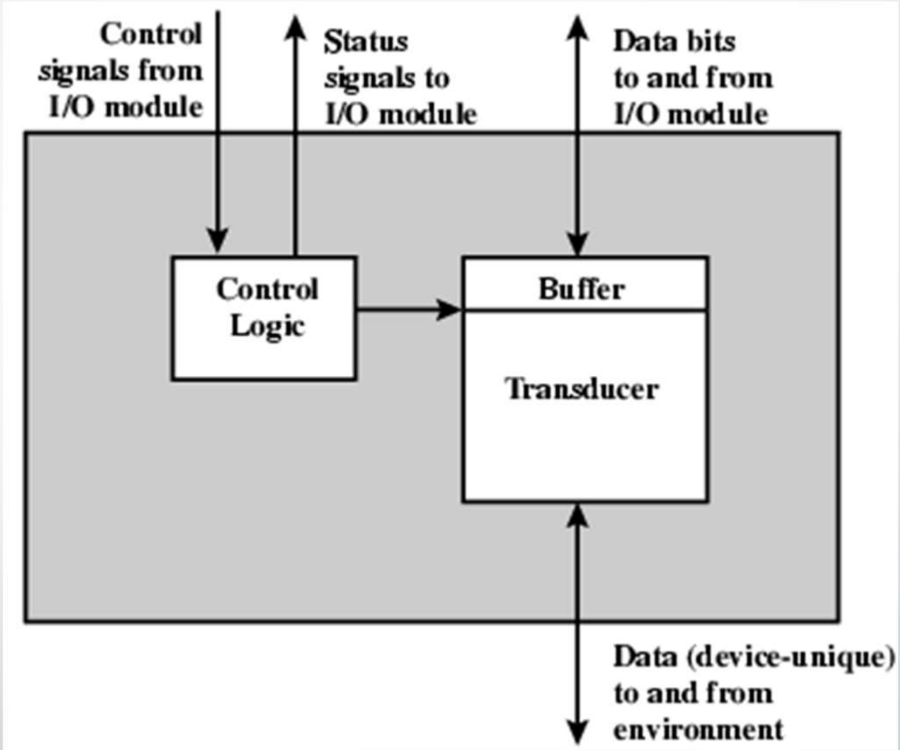
I/O Module

- Interface to CPU and Memory
- Interface to one or more peripherals



Peripheral Devices Types & Block Diagram

- Human readable
  - Screen, printer, keyboard
- Machine readable
  - Monitoring and control
- Communication
  - Modem
  - Network Interface Card (NIC)

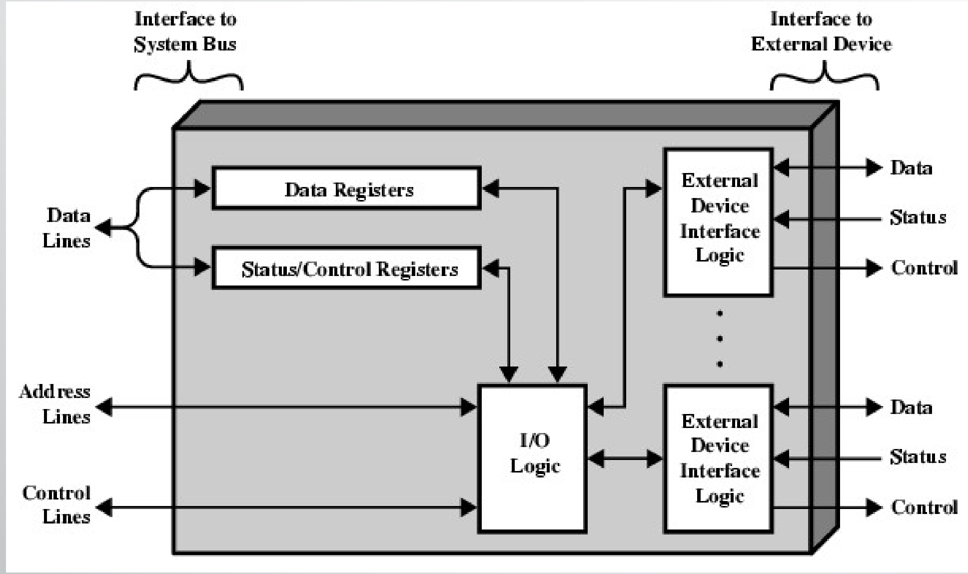


More about I/O Modules

OÉ Gaillimh
NUI Galway

- I/O Module Functions
  - Control & Timing
  - CPU Communication
  - Device Communication
  - Data Buffering
  - Error Detection
- I/O CPU Steps
  - CPU checks I/O module device status
  - I/O module returns status
  - If ready, CPU requests data transfer
  - I/O module gets data from device
  - I/O module transfers data to CPU
  - Variations for output, DMA, etc.

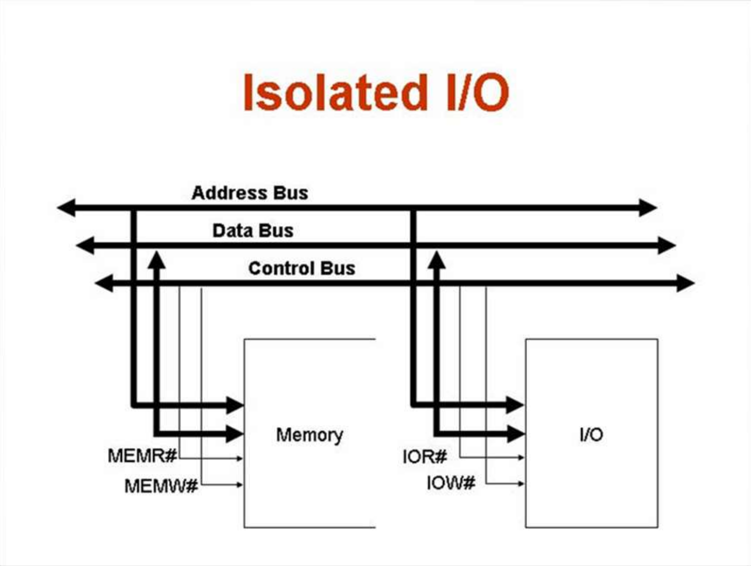
I/O Module Diagram & Design Decisions



- Hide or reveal device properties to CPU
- Support multiple or single device
- Control device functions or leave for CPU
- Also O/S decisions
  - e.g. Unix treats everything it can as a file

I/O Mapping

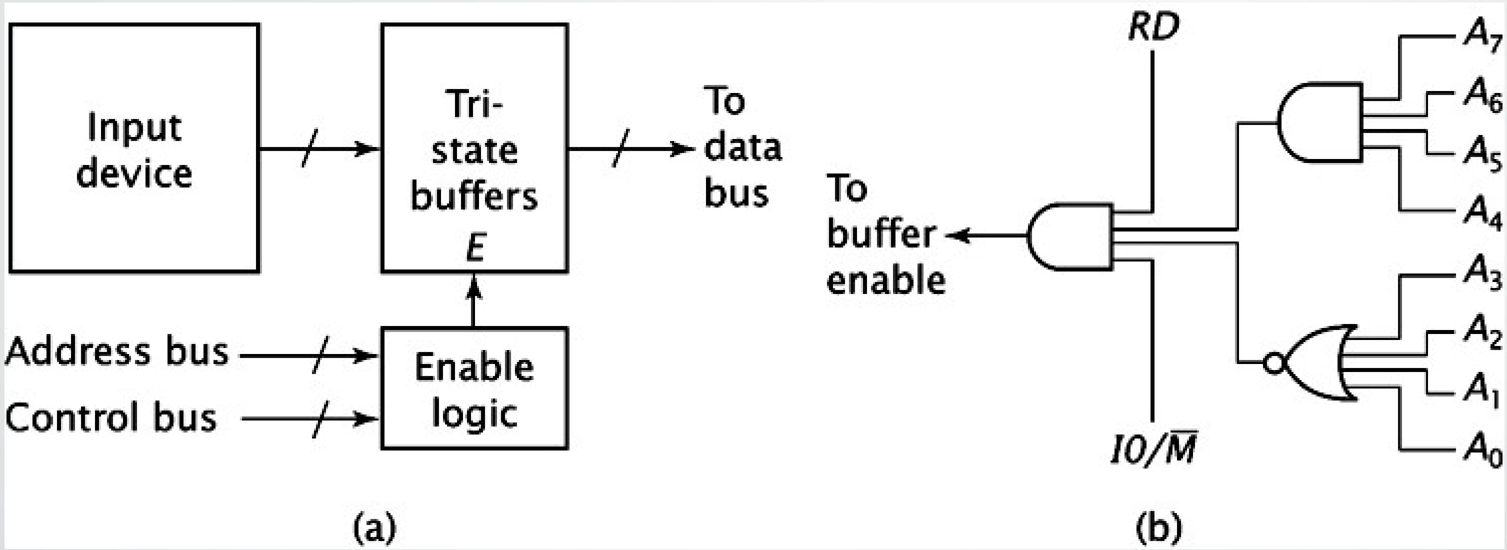
- Memory mapped I/O
  - Devices and memory share an address space
  - I/O looks just like memory read/write
  - No special commands for I/O
    - Large selection of memory access commands available
- Isolated I/O
  - Separate address spaces
  - Need I/O or memory select lines
  - Special commands for I/O
  - Special CPU control signals
  - Devices and Memory can have overlapping addresses



Addressing I/O Devices

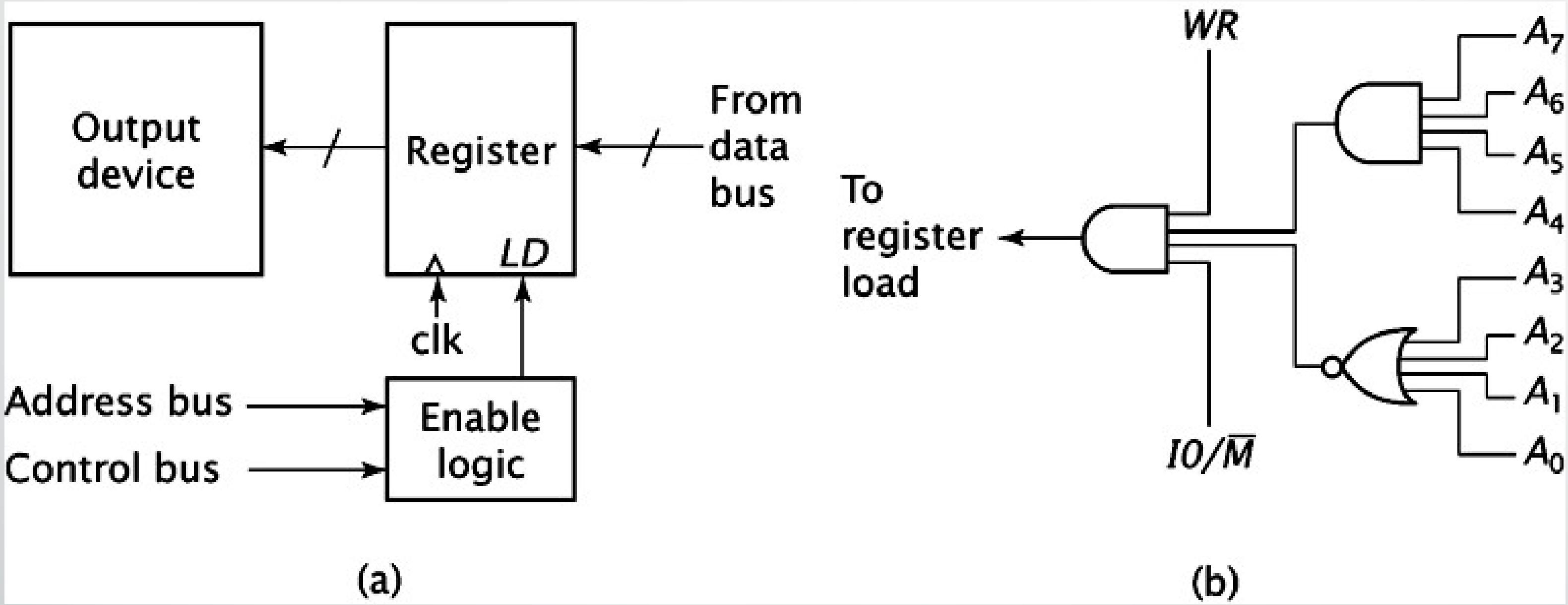
- I/O data transfer is very like memory access (CPU viewpoint)
- Each device given unique identifier
- CPU commands contain identifier (address)
- The I/O Module should contain address decoding logic

Input Devices



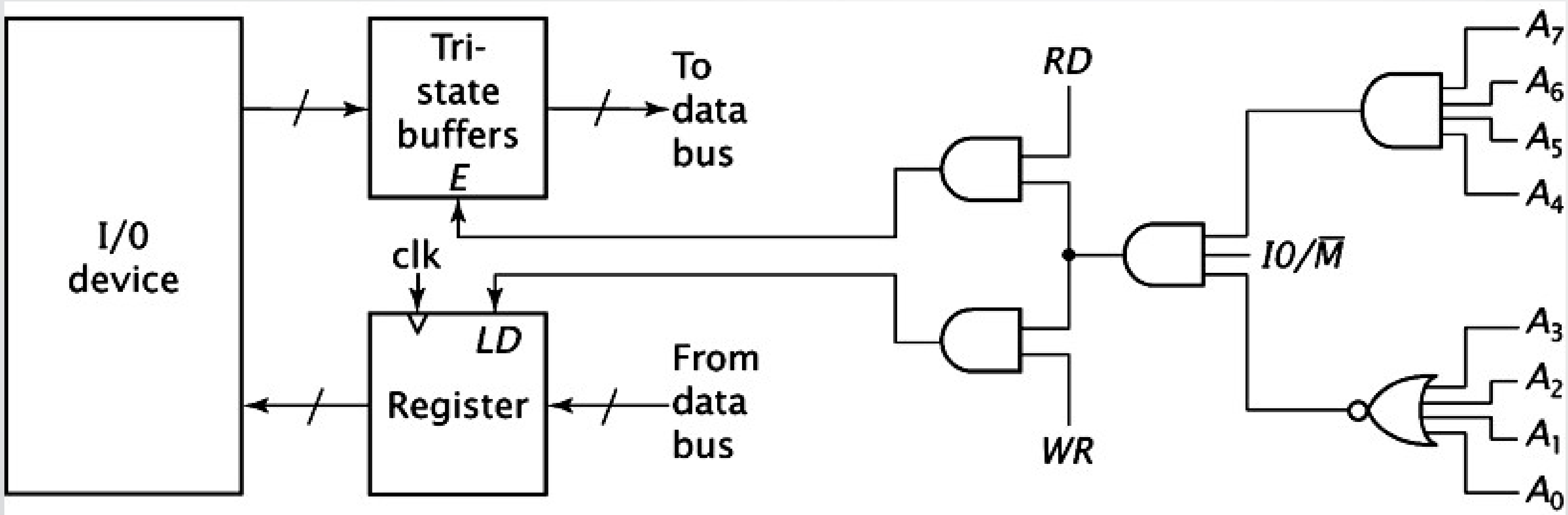
- When the values of the address/control buses are correct (the I/O device is addressed) the buffers are enabled, and the data passes on to the data bus; the CPU reads this data
- When the conditions are not right, the logic bloc (enable logic) will not enable the buffers; no data on the data bus
- The example shows an I/O device mapped at address 1111 0000 on a computer with 8-bit address bus and RD and IO/M' control signals

Output Devices



- Since the output devices read data from the data bus, they don't need the buffers; data will be made available to all the devices
- Only the correctly decoded one (addressed) will read in the data
- Example shows an output device mapped at 11110000 address in an 8-bit address bus computer, with WR and IO/M' signals

Bidirectional Devices (1)



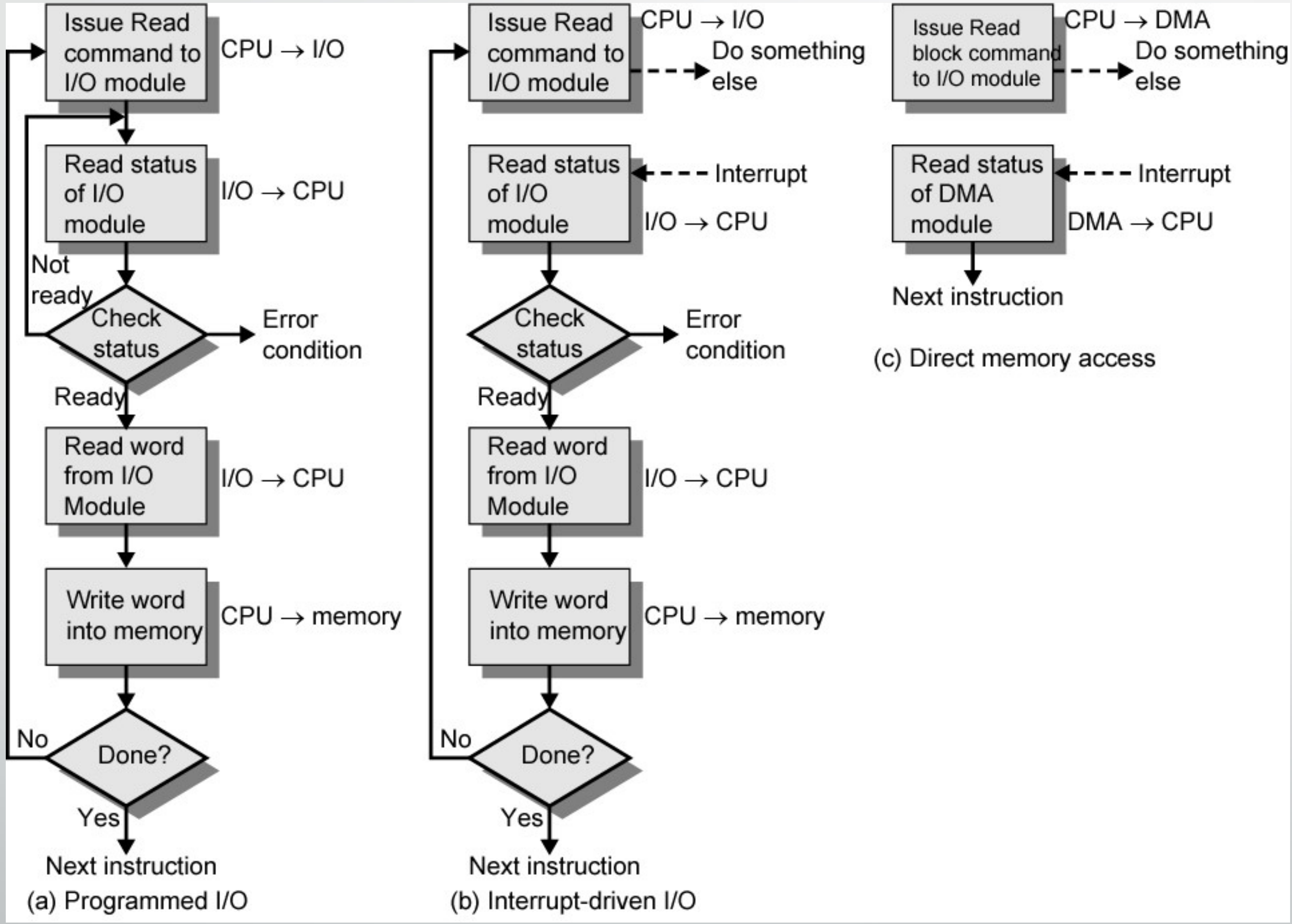
- Bidirectional devices require actually two interfaces, one for input and the other for output.
- Same gates could be used to generate the enable signal (for both the tri state buffers and the registers); the difference between read and write are made through the control signals (RD, WR)
- The example shows a combined interface for 1111 0000 address.

Bidirectional Devices (2)

OÉ Gaillimh
NUI Galway

- In real systems, we need to access more than just one output and one input data register
- Usually peripherals are issued with commands by the processor and they take some action and in response present data
- Up to how the processor knows if the peripheral device is ready after a command, we can have:
  - Programmed I/O (or also known as Polled I/O)
  - Interrupt driven I/O

Input / Output Techniques



Programmed I/O

- Overview
- CPU has direct control over I/O
  - Sensing status
  - Read/write commands
  - Transferring data
- CPU waits for I/O module to complete operation
- Wastes CPU time

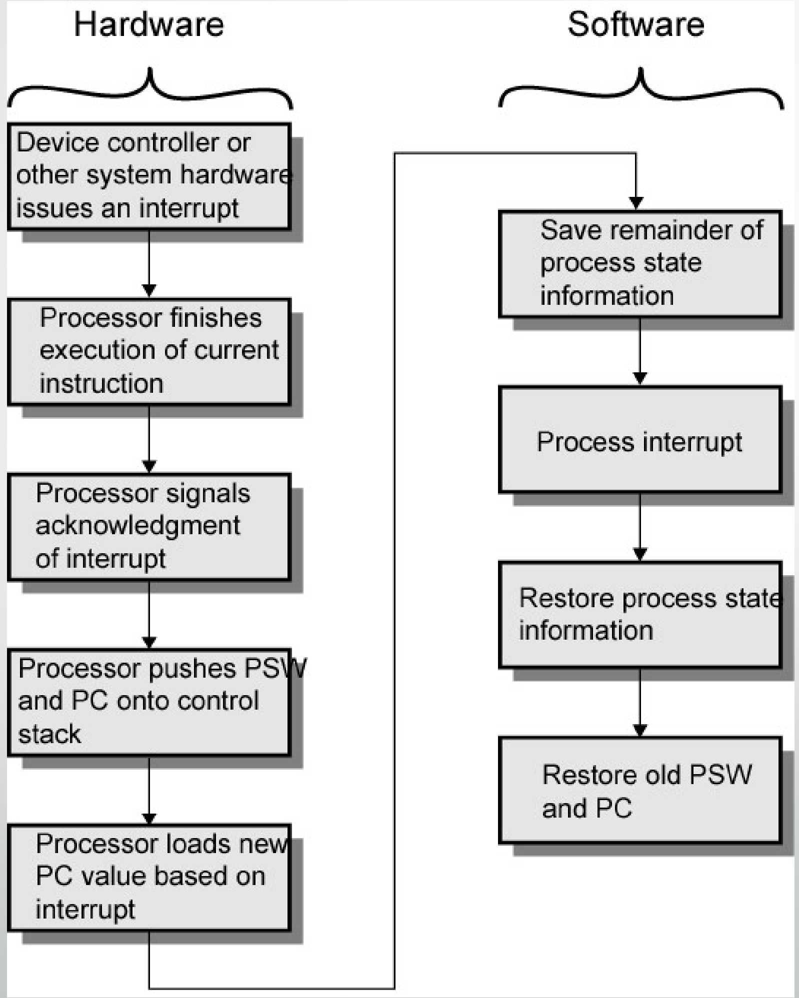
- Operations
  - CPU requests I/O operation
  - I/O module performs operation
  - I/O module sets status bits
  - CPU checks status bits periodically
  - I/O module does not inform CPU directly
  - I/O module does not interrupt CPU
  - CPU may wait or come back later

Interrupt Driven I/O

OÉ Gaillimh
NUI Galway

- Overview
  - Overcomes CPU waiting
  - No repeated CPU checking of device
  - I/O module interrupts when ready
- Operations
  - CPU issues read command
  - I/O module gets data from peripheral whilst CPU does other work
  - I/O module interrupts CPU
  - CPU requests data
  - I/O module transfers data

Simple Interrupt Processing



CPU Viewpoint

OÉ Gaillimh
NUI Galway

- Issue read command
- Do other work
- Check for interrupt at end of each instruction cycle
- If interrupted:
  - Save context (registers)
  - Process interrupt
    - Fetch data & store
  - Restore context (registers)

Design Issues

OÉ Gaillimh
NUI Galway

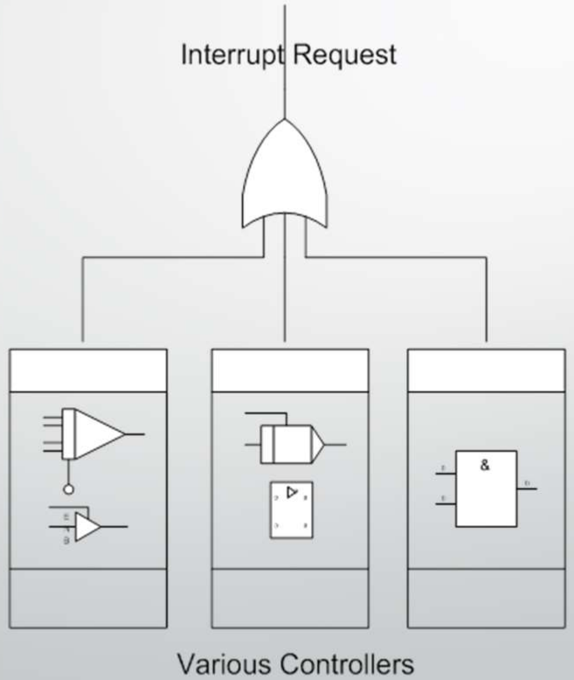
- How do you identify the module issuing the interrupt?
- How do you deal with multiple interrupts?
  - i.e. an interrupt handler being interrupted

Identifying Interrupting Module

- Different line for each module
  - Limits number of devices
- Software poll
  - CPU asks each module in turn
  - Slow
- Daisy Chain or Hardware poll
  - Interrupt Acknowledge sent down a chain
  - Module responsible places vector on bus
  - CPU uses vector to identify handler routine
- Bus Arbitration (e.g. PCI & SCSI)
  - Module must claim the bus before it can raise interrupt, thus only one module can rise the interrupt at a time
  - When processor detects interrupt, processor issues an interrupt acknowledge
  - Device places its vector on the data bus

Multiple Interrupts

- Each interrupt line has a priority
- Higher priority lines can interrupt lower priority lines



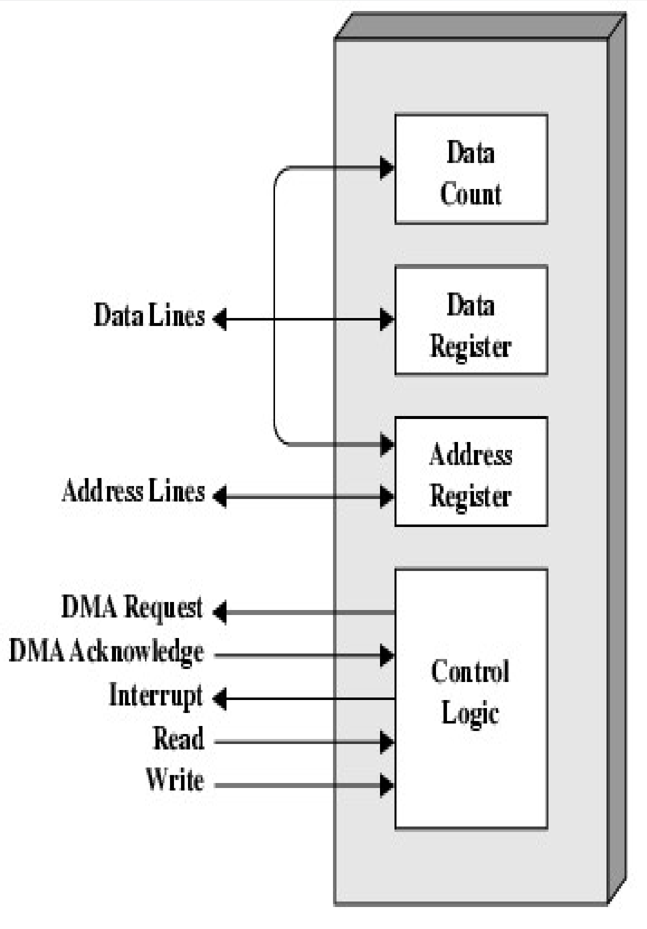
Direct Memory Access

OÉ Gaillimh
NUI Galway

- Interrupt driven and programmed I/O require *active CPU intervention*
  - Transfer rate is limited by the speed of processor testing and servicing a device
  - CPU is tied up in managing an I/O transfer. A number of instructions must be executed for each I/O transfer.
- DMA is the answer when large amounts of data need to be transferred.

DMA Function and Module

OÉ Gaillimh
NUI Galway

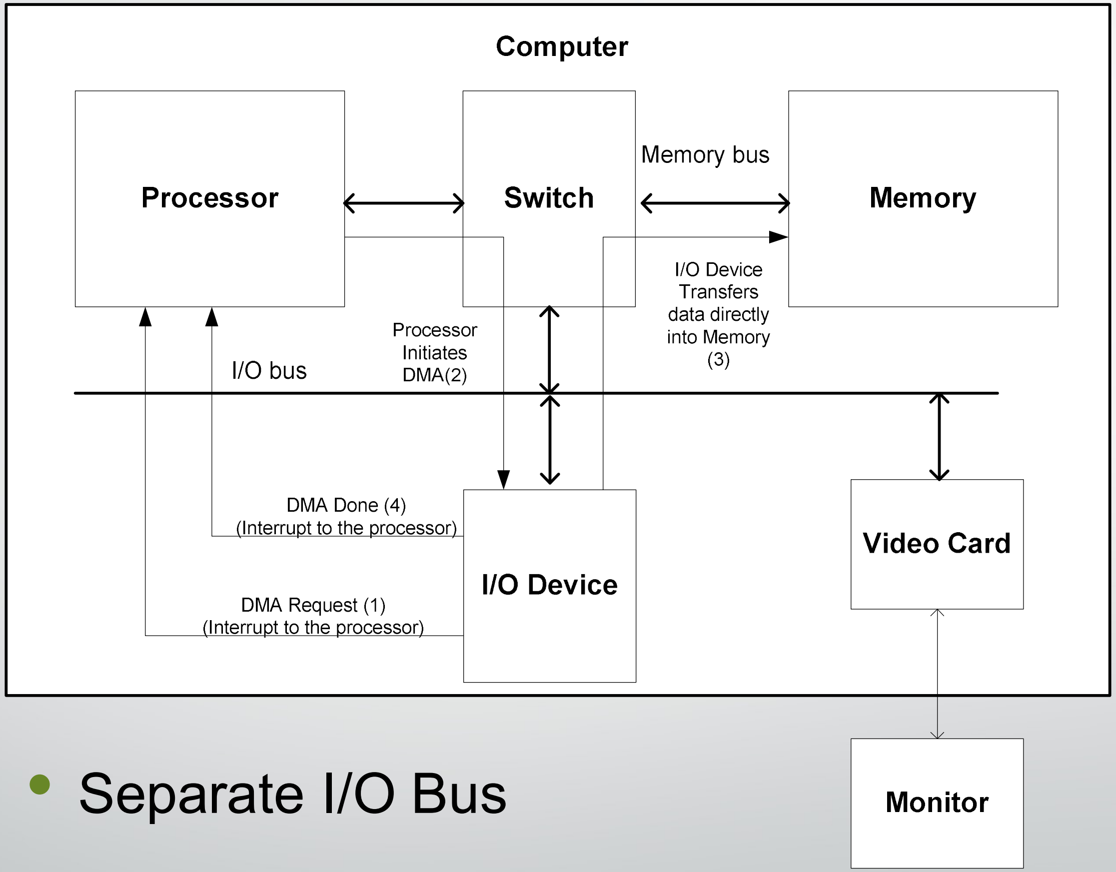


- DMA controller able to mimic the CPU and take over for I/O transfers
- CPU tells DMA controller:
  - Operation to execute
  - Device address involved in the I/O operation (sent on data lines)
  - Starting address of memory block for data (sent on data lines) and stored in the DMA address register
  - Amount of data to be transferred (sent on data lines) and stored into the data count
- CPU carries on with other work
- DMA controller deals with transfer
- DMA controller sends interrupt when finished

DMA Transfer Cycle Stealing

- DMA controller takes over bus for a cycle
- Transfer of one word of data
- Not an interrupt
  - CPU does not switch context
- CPU suspended just before it accesses bus
  - i.e. before an operand or data fetch or a data write
- Slows down CPU but not as much as CPU doing transfer

DMA Operation Example



- Separate I/O Bus

References

- "Computer Systems Organization & Architecture", John D. Carpinelli, ISBN: 0-201-61253-4
- "Computer Organization and Architecture", William Stallings, 8th Edition