15 Weeks ESE 2005

*task Takis for Lab Manual

Week 1: Digital Hardware Review
- pages 1-5
- early release!

Week 2: Digital Hardware Review, page 6
Digital Systems Review
→ "combinational logic"
→ pages 1-5

LAB:
- VHDL Tutorial, pages 1-7
- early release → emphasize structural implementation
- introduce software (Web Edition Xilinx Simulator for Ubuntu) → get details from Takis
- ask them to start Lab 1: Full Adder

Week 3:
Digital Logic Design Simplifying Strategies
- (A+A)B
- multi-level combinational logic
- bubble pushing
- pages 6, 7
- begin (officially) Lab 1

Week 4
Digital Systems Cont'd
- floating value (tri-state logic)
- K-Maps
- multiplexors, multiplexor logic
→ pages 8-11

Won't be accepted past this time → Lab 1 turned in, Lab 2 begin

Week 5 — Timing Issues and Sequential Logic

- t_{pd}, t_{cd}
- glitches
- SR-Latch
- D-Latch
- D-Flipflop
- pages 12–14

LAB: Lab 2 continues...

give weeks 2 →

- cover implementation of registers in VHDL

Week 6 — Sequential/Synchronous Logic Design.

- race conditions, critical paths
- key attributes
  - pages 15–16
- intro to Finite State Machines
  - pages 1–4

LAB: Lab 3 begins, Lab 2 turned in

Week 7: Finite State Machines Cont'd

- implementation of FSM (page 5)
- Moore vs. Mealy
- Robot Snail Example, pages 6-8

LAB: Lab 3 continues

Week 8: • Introduce embedded hardware kits
→ Beaglebone Kit
- device tree
- GPIOs
- device tree overlays
- ask Takis for GPIO Lab

LAB: Lab 3 is turned in, start GPIO Lab on Beaglebone

Week 9: • Continue with Embedded Linux
Hardware
- ADC on Beaglebone accessing using sysfs

LAB: Finish up GPIO, start triple axis accelerometer lab, ask Takis for ADC Lab

Week 10: • Stored Program Computer Architectures
page 9 10
→ TEST #1 in Lecture (10%)
- Methods of Integration: pages 10, 11

Lab: finish ADC Lab, start "Lab 4"

Week 11: Memories
- Overview: arrays, bit cells, DRAM, SRAM
- → pages 1-4 (page 4)! so they can do Lab 5
- ALU

Lab: turn in Lab 4, begin Lab 5

Week 12: Memories Cont'd
- register file, ROM; pages 5-8
- Programmed I/O
- LAB
  - turn in 5, start Lab 6 - pages 1-3

Week 13: ARM 32-bit Micro Architecture & Architecture
- Intro
  - pages 5, 6
- Architecture (Assembly Language)
  - pages 7-9
- LAB: turn in Lab 6, start Lab 7

Week 14: ARM Architecture Cont'd
- pages 9-
- programming 11-
- LAB: finish up

Week 15: LAB/Review Session
- final test 15%

SUBMISSIONS

- 4 page max (single sided) — two-column
- hardcopy

PROCEDURES

- ask them to tidy up

Lambton College

School of Technology, Energy & Apprenticeship

International

Course Outline – Winter 2017

Course Code: ESE 2005 **Course Title:** Embedded Systems Architecture I

Prepared By:	Jay Nadeau	*Date:*	April 2016
Revised By:	*Click here to enter name.*	*Date:*	*Click here to enter a date.*
Approved By:	*Click here to enter name.*		

Prerequisite:	ESE 1005, ESE 1014
Corequisite:	*Click here to enter corequisites or type None.*
Prerequisite for:	ESE 3005

1. Course Description

Students are introduced to the Von Neumann and Harvard processor architectures. Students will recognize the differences between Microprocessors, Microcontrollers and System-on-Chip/System-on-Module, and the differences between Intel and ARM processors. Students are introduced to the building blocks of microcomputers and microprocessors such as memory, I/O, registers, the ALU and the control unit. The design of computer instruction sets and CPUs are reviewed with students designing systems that use peripherals such as real-time clocks, analog-to-digital converters, digital-to-analog converters and interfacing with GPIO pins. Students will write applications that use the ARM and Thumb instruction sets.

2. General Education and Essential Employability Skills

This course provides the following provincial Essential Employability Skills:

#1: Communication
#2: Numeracy
#3: Critical Thinking and Problem Solving
#4: Information Management
#5: Interpersonal
#6: Personal

Is this course approved as a General Education course?

☒ ***No*** ☐ ***Yes***
Students should refer to their program's restricted General Education courses for final determination.

3. Learning Outcomes

Upon successful completion of this course, the student will be able to:

1. Describe differences between Von Neumann and Harvard computer architectures.
2. List differences between microprocessors, microcontrollers and SoCs/SoMs.
3. Describe the different types of computer memory.
4. Explain the functions of the I/O bus, the ALU, the control unit and the register file within the CPU.
5. Discuss differences between microcode, machine language, assembly language and high-level languages.
6. Explain what a RISC computer is.
7. List differences between Intel and ARM processors.
8. Describe and design systems that use peripherals, GPIOs, buses, interrupts, DACs/ADCs and PWM signals.
9. Write an assembly language program using ARM and Thumb instruction sets.
10. Design a simple 8-bit Harvard architecture CPU using VHDL.

11. Design an instruction set for use on an 8-bit Harvard architecture CPU.

4. Course Objectives

		Learning Outcome Reference Number	
Unit 1	***Microcomputers, Microcontrollers and SoCs***		
1.1	*Describe the Von Neumann architecture.*	[1]	
1.2	*Describe the Harvard architecture.*	[1]	
1.3	*List benefits and drawbacks of both the Von Neumann and Harvard architectures.*	[1]	
1.4	*Explain differences between a microprocessor and a microcontroller.*	[2]	
1.5	*Describe the features of a SoC/SoM.*	[2]	
Unit 2	***Computer Architecture***		
2.1	*Describe the differences between RAM and ROM.*	[3]	
2.2	*Describe the differences between SRAM and DRAM.*	[3]	
2.3	*Describe the differences between EPROM, EEPROM, Flash, Masked ROM and PROM.*	[3]	
2.4	*Describe programmed I/O, interrupt I/O and direct memory access.*	[4]	
2.5	*List the different CPU registers.*	[4]	
2.6	*Describe the functions of the ALU.*	[4]	
2.7	*Explain the main purpose of the control unit.*	[4]	
2.8	*Describe the purpose of an instruction set.*	[5]	
2.9	*Explain the differences between microcode and assembly language.*	[5]	
2.10	*Explain the relationship between machine language and assembly language.*	[5]	
2.11	*Explain the relationship between a high-level programming language and assembly language.*	[5]	
2.12	*Explain what a reduced instruction set computer is and the philosophy behind a RISC.*	[6]	
2.13	*List advantages and disadvantages of a RISC.*	[6]	
2.14	*Describe the Intel x86 CPU architecture.*	[7]	
2.15	*Describe the ARM CPU architecture.*	[7]	
Unit 3	***Peripheral Interfacing***		

Commented [CP1]: Similar as comment made above for "the difference"

3.1	*List some examples of common computer peripherals.*	[8]
3.2	*Explain what a GPIO is.*	[8]
3.3	*Explain how GPIOs are used.*	[8]
3.4	*List the different types of system buses.*	[8]
3.5	*Explain what an interrupt is.*	[8]
3.6	*Explain the difference between a hardware interrupt and a software interrupt.*	[8]
3.7	*Write an interrupt service routine.*	[8]
3.8	*Describe the function of DACs.*	[8]
3.9	*Describe the function of ADCs.*	[8]
3.10	*Explain what the sampling rate of an ADC is.*	[8]
3.11	*Explain the term quantization with respect to ADC.*	[8]
3.12	*Explain what pulse width modulation is.*	[8]
3.13	*Explain the term duty-cycle with respect to PWM.*	[8]

Unit 4 ***Programming***

4.1	*Write an assembly language program that uses ARM instructions.*	[9]
4.2	*Write an assembly language program that uses Thumb instructions.*	[9]
4.3	*Explain the difference between the ARM instruction set and the Thumb instruction set.*	[9]

Unit 5 ***CPU and Instruction Set Design***

5.1	*Design and implement an ALU using VHDL.*	[10]
5.2	*Design and implement a register file using VHDL.*	[10]
5.3	*Design and implement a control unit using VHDL.*	[10]
5.4	*Design a simple 8-bit Harvard architecture CPU using the previously designed ALU, register and control unit.*	[10]
5.5	*Design an instruction set that makes use of the simple 8-bit Harvard microprogrammed CPU.*	[11]

5. Resources and Supplies

a. Required

Digital Design and Computer Architecture: ARM Edition by Sarah Harris and David Harris; Morgan Kaufmann, (May 2016).

b. Supplemental

None

6. Methodology

The course is designed to provide an emphasis on hands on experience via laboratory sessions and assignments. The labs will make use of a modern embedded IDE to implement laboratory programs on an ARM based processor.

7. Student Evaluation

The following elements will determine the student's final grade:

Evaluation Method	Weight	Course Learning Outcome
Tests (#1@10%, #2@15%)	25%	1-11
Assignments (3 @ 5% each)	15%	1-11
Laboratory Sessions (10 @ 6% each)	60%	1-11
Total	**100%**	

The round off mathematical principle will be used. Percentages are converted to letter grades and grade points as follows:

Mark (%)	Grade	Grade Point	Mark (%)	Grade	Grade Point
94-100	A+	4.0	67-69	C+	2.3
87-93	A	3.7	63-66	C	2.0
80-86	A-	3.5	60-62	C-	1.7
77-79	B+	3.2	50-59	D	1.0
73-76	B	3.0	0-49	F	0.0
70-72	B-	2.7			

8. Academic Integrity

Lambton College is committed to high ethical standards in all academic activities within the College, including research, reporting and learning assessment (e.g. tests, lab reports, essays).

The cornerstone of academic integrity and professional reputation is principled conduct. All scholastic and academic activity must be free of all forms of academic dishonesty, including copying, plagiarism and cheating.

Lambton College will not tolerate any academic dishonesty, a position reflected in Lambton College policy. Students should make themselves familiar with the Students Rights and Responsibilities Policy, located on the MyLambton website for details concerning academic dishonesty and the penalties for dishonesty and unethical conduct.

Questions regarding this policy, or requests for additional clarification, should be directed to the Lambton College Centre for Academic Integrity

9. Related Items

Students with Disabilities

If you are a student with a disability please identify your needs to the professor and/or the Accessibility Centre so that support services can be arranged for you. You can do this by making an appointment at the Accessibility Centre, Room L103 ext.3427 or by arranging a personal interview with the professor to discuss your needs.

Student Rights and Responsibility Policy

Acceptable behaviour in class is established by the instructor and is expected by all students. Any form of harassment or violence will not be tolerated. Action will be taken as outlined in Lambton College policy.

Cheating and plagiarism are serious academic offences subject to disciplinary action. It is the student's responsibility to be aware of the cheating policy as described in the Lambton College Student Rights and Responsibilities policy. For further information on all of these policies, links may be found on the Lambton College website.

Prior Learning Assessment Statement

This course is eligible for Prior Learning Assessment
☒ ***Yes*** ☐ ***No***

If yes has been selected, you may choose to contact the Counselling Department for advice on Prior Learning Assessment.

Date of Withdrawal without Academic Penalty

Please consult the Academic Regulations and Registrar's published dates.

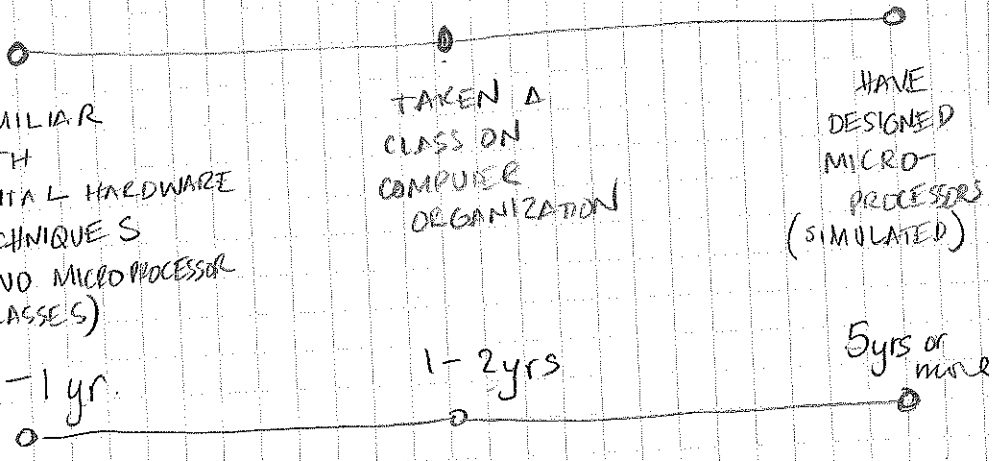
Waiver of Responsibility

Every attempt has been made to ensure the accuracy of this information as of the date of publication. The content may be modified, without notice, as deemed appropriate by the College.

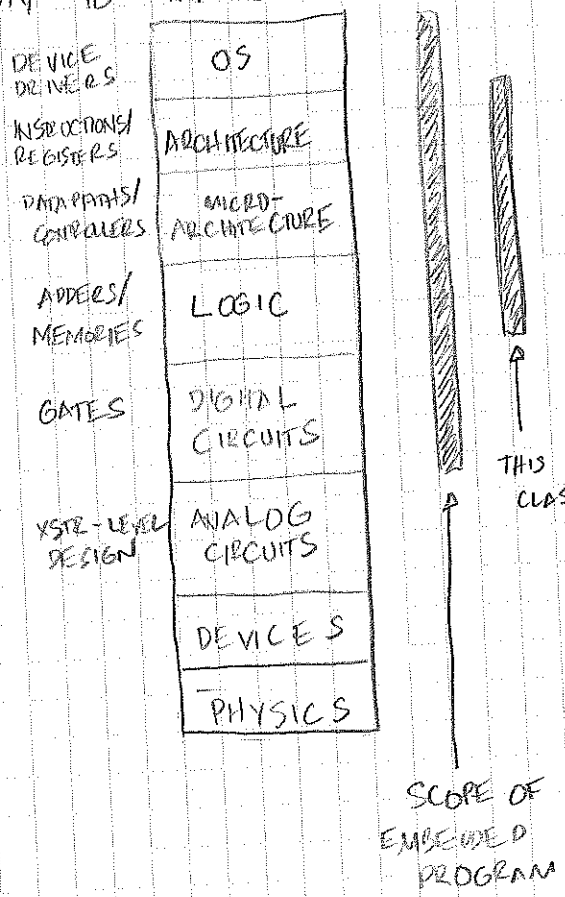
Note: It is the student's responsibility to retain course outlines for possible future use to support applications for transfer of credit to other educational institutions.

DIGITAL HARDWARE REVIEW

- our goal is to go from gate-level theory to the architecture of modern embedded processors
- assuming this is not the first digital logic class you've been
  → informal survey/histogram

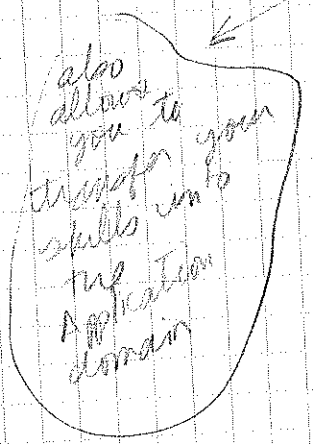


- KEY ASPECT TO DIGITAL DESIGN IS ABSTRACTION
  → A WAY TO MANAGE COMPLEXITY



DIGITAL HARDWARE REVIEW

- much of this program — what makes it cutting edge — is the foregrounding of Linux as the context for embedded design
  - → is Linux the only solution?
  - → what does running Linux demand of the hardware?
    - → not every embedded processor is capable of running Linux — they lack the resources (memory, processor speed)
    - → you should be aware that you don't need Linux or a powerful processor but studying Linux systems should prepare you to handle simpler embedded implementations
    - → looking at A-series ARM processors



Cortex-M Cortex-R Cortex-A

performance →

power consumption →

memory/resource requirements →

  - → why is Linux a good solution, possibly the future

DIGITAL HARDWARE REVIEW

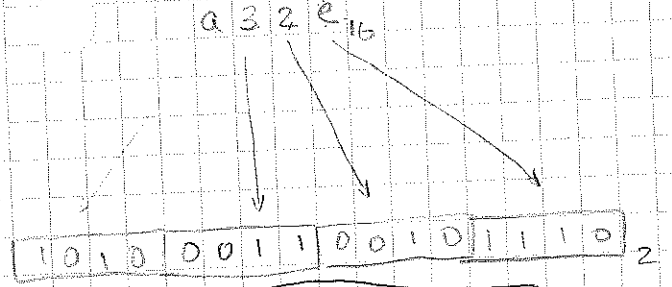
- in addition to abstraction digital hardware designers employ "discipline":
  - intentionally restricting their design choices so that they can work more efficiently at a higher level of abstraction
  - e.g. digital circuits are much easier to design than analog circuits, and by restricting ourselves to simple digital models, we can more easily design systems
  - great for production, perhaps not so great for innovation

- the three "y's"
  - hierarchy - subdivision of system into modules
  - modularity - defining modules and interfaces for easy interconnection
  - regularity - striving for uniformity of the modules

- digital representation itself reflects some of this thinking — it gives us a disciplined means of working with numbers (not as accurate as actual numbers) that leads to some degree of modularity and regularity in the computing hardware

DECIMAL	BINARY	HEX
0	0 0 0 0	0
1	0 0 0 1	1
2	0 0 1 0	2
3	0 0 1 1	3
4	0 1 0 0	4
5	0 1 0 1	5
6	0 1 1 0	6
7	0 1 1 1	7
8	1 0 0 0	8
9	1 0 0 1	9
10	1 0 1 0	A
11	1 0 1 1	B
12	1 1 0 0	C
13	1 1 0 1	D
14	1 1 1 0	E
15	1 1 1 1	F

a great numbering system which complements binary (since groups of 4 bits are commonly used in computing but what about different groupings like 3-bits at a time?



0 0 0	0
0 0 1	1
0 1 0	2
0 1 1	3
1 0 0	4
1 0 1	5
1 1 0	6
1 1 1	7

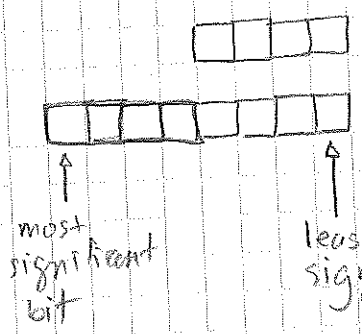
→ octal!

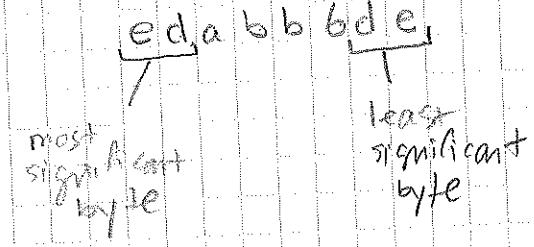
$XYZ = X \times 8^2 + Y \times 8^1 + Z \times 8^0$

5 6 7 2_8 → a 12-bit word

divisible by 3

HEX → BINARY
or
BINARY → HEX
is easy





older convention

k → "kilo" → 10^3, but kbyte is not 1000 bytes, it's 1024 bytes = 2^{10}

M → "mega" → 10^6 → similarly, megabyte is not 10^6 bytes, it's 2^{20} = 1048576 bytes

sometimes you'll see the prefix "mebi" and "kibi" which are exactly 2^{20} and 2^{10}, respectively

and gibi – 2^{30},
tebi – 2^{40},
pebi – 2^{50}.

- microprocessors handle chunks of data in portions called words

  8-bit words
  16-bit words
  32-bit words
  64-bit words

  } small microcontrollers these days for embedded applications

- signed numbers
  - sign bits
  - two's complement) review yourself

- talk about fixed-point, floating point, trade offs

LOGIC GATES

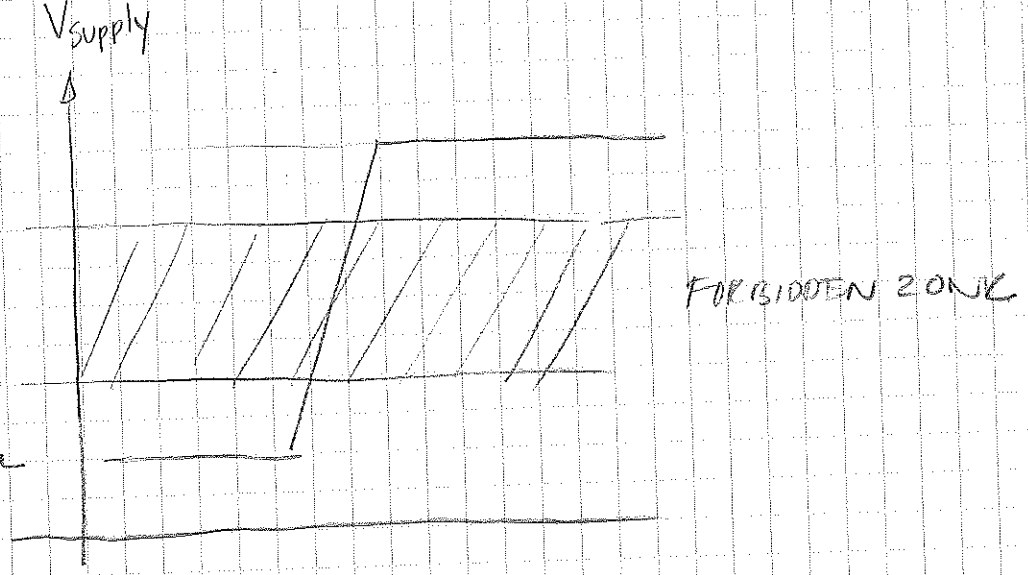
- do NOT, Buffer, AND, OR, XOR, NAND, NOR, XNOR } up to 2-input

  $Y = AB$ $Y = A + B$ $Y = A \oplus B$

- multi-input gate:

  extensions: XNOR: $Y = \overline{A_1 \oplus A_2 \oplus A_3 \oplus \dots \oplus A_N}$

- beneath the abstraction:



Digital Systems Review

Combinational and Sequential Logic

- digital circuits are either combinational or sequential

Combinational: outputs depend only on the current values of the inputs (memoryless)

Sequential: outputs depend on both current and previous states of the inputs (have memory)

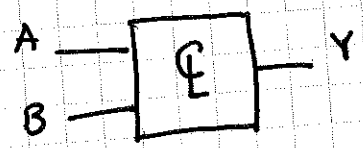
- there are both timing and functional specifications

timing: for comb. logic, we note the lower and upper bounds of the overall delay from input to output

functional: truth table characterization

- we begin with functional specification

combinational logic



follows the rule of OR gate

$Y = F(A,B) = A+B$

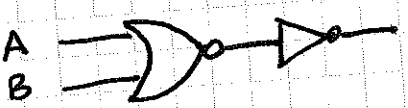
a generalized two-input, single outp comb. logic system/cir

- may be implemented in the case of $F(A,B) = A+B$ as

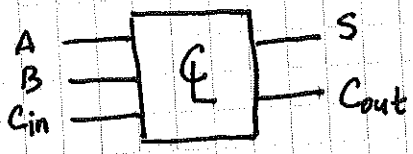
same implementation

i)

ii)



- a full adder example: with multiple outputs



$S = A \oplus B \oplus C_{in}$

$C_{out} = AB + AC_{in} + BC_{in}$

will learn in Chapter 5

to simplify

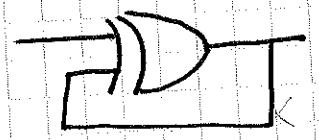
note slash notation:

multiple lines of input or output are called "buses" (singular: "bus")

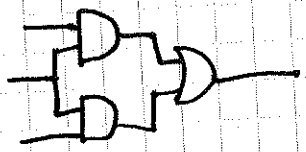
- we can interconnect combinational logic circuits to create new combinational logic circuits, bearing in mind that there can be no cyclic paths

meaning every path through the circuits visit each circuit node at most once

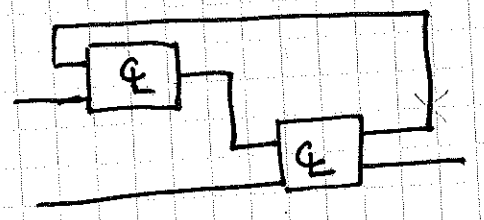
examples:



NOT COMBINATIONAL



COMBINATIONAL



NOT COMBINATIONAL

watch for algebraic loops in systems — every loop must contain some delay (either implicitly or explicitly)

BOOLEAN EQUATIONS

terminology

$\bar{A}$ — the complement or inverse of A (the "true form" of A)
- the variable or its complement are called literals

$\bar{A}B$, $A\bar{B}\bar{C}$, B — the AND of one or more literals is called a product or implicant

$A+B$, $A+\bar{C}$, — the OR of one or more literals is called a sum.

$A\bar{B}\bar{C}$, $\bar{A}BC$, $A\bar{B}C$ — a minterm is a product involving all inputs of the function

(3-input function)

$A+\bar{B}+C$, $A+B+C$, $C+A+\bar{B}$ — a maxterm is a sum involving all inputs to the function

(3-input function)

≈

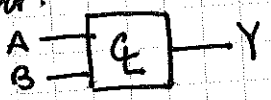
A truth table with N inputs contains 2^N rows.

✱ — we can associate a minterm with each row that is true ✱

→ ~~Sum~~ "sum-of-products form"

EXAMPLE

Suppose for:



we have

A	B	Y	min term	minterm name
0	0	0	$\bar{A}\bar{B}$	m_0
0	1	1	$\bar{A}B$	m_1
1	0	0	$A\bar{B}$	m_2
1	1	0	AB	m_3

thus, $Y = \bar{A}B$

if the truth table is amended so that AB also yields $Y = 1$, then we have

$$Y = \underbrace{\bar{A}B}_{m_1} + \underbrace{AB}_{m_3}$$

$$F(A,B) = \sum(m_1, m_3) \quad \text{or} \quad \sum(1,3)$$

short-form of sum-of-products expression

$\bar{A}B + AB$

EXAMPLE Random 3 input-truth table

A	B	C	Y	
0	0	0	1	m_0
0	0	1	0	m_1
0	1	0	0	m_2
0	1	1	0	m_3
1	0	0	1	m_4
1	0	1	1	m_5
1	1	0	0	m_6
1	1	1	0	m_7

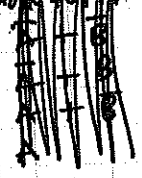
$F(A,B,C) = \Sigma(0,4,5)$

built for $Y = (\bar{A}\bar{B}\bar{C}) + (A\bar{B}\bar{C}) + (A\bar{B}C)$

* we can also associate a maxterm ~~that is~~ with each row that is false *

EXAMPLE

A	B	Y	maxterm	maxterm	maxterm name
0	0	0		$A+B$	M_0
0	1	1		$A+\bar{B}$	M_1
1	0	0		$\bar{A}+B$	M_2
1	1	1		$\bar{A}+\bar{B}$	M_3



check → $\therefore Y = (A+B)(\bar{A}+B) = \Pi(M_0, M_2) = \Pi(0,2)$

BOOLEAN ALGEBRA

- we are interested in using Boolean algebra to simplify the expressions we write using sum-of-products or product-of-sums rules.

AXIOMS

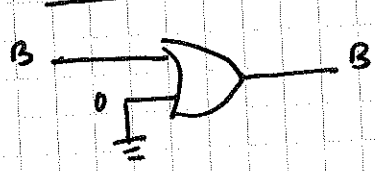
	AXIOM	DUAL	NAME
A1:	$B=0$ if $B\neq 1$	A1': $B=1$ if $B\neq 0$	binary field
A2:	$\bar{0}=1$	A2': $\bar{1}=0$	NOT
A3:	$0\cdot 0=0$	A3': $1+1=1$	AND/OR
A4:	$1\cdot 1=1$	A4': $0+0=0$	AND/OR
A5:	$0\cdot 1=1\cdot 0=0$	A5': $1+0=0+1=1$	AND/OR

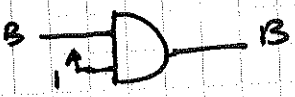
we use the prime (') to denote the dual of a statement

THEOREMS OF ONE VARIABLE

THEOREM		DUAL	NAME
T1:	$B \cdot 1 = B$	T1': $B + 0 = B$	identity
T2:	$B \cdot 0 = 0$	T2': $B + 1 = 1$	null element
T3:	$B \cdot B = B$	T3': $B + B = B$	idempotency
T4:	$\overline{\overline{B}} = B$		involution
T5:	$B \cdot \overline{B} = 0$	T5': $B + \overline{B} = 1$	complements

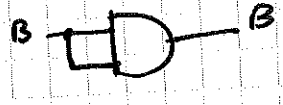
APPLICATIONS:

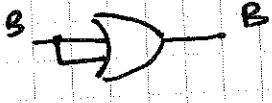




} two possible realizations of a buffer

also:



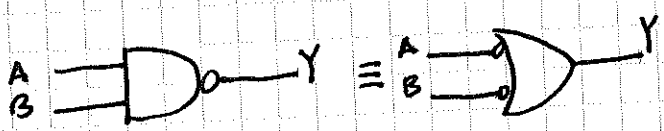


THEOREMS OF SEVERAL VARIABLES

THEOREM		DUAL	NAME
T6:	$B \cdot C = C \cdot B$	T6': $B + C = C + B$	commutativity
T7:	$(B \cdot C) \cdot D = B \cdot (C \cdot D)$	T7': $(B + C) + D = B + (C + D)$	associativity
T8:	$BC + BD = B(C + D)$	T8': $(B + C)(B + D) = B + (CD)$	distributivity
T9:	$B(B + C) = B$	T9': $B + (BC) = B$	covering
T10:	$(BC) + (B\overline{C}) = B$	T10': $(B + C)(B + \overline{C}) = B$	combining
T11:	$BC + \overline{B}D + CD = BC + \overline{B}D$	T11': $(B + C)(\overline{B} + D)(C + D) = (B + C)(\overline{B} + D)$	consensus
T12:	$\overline{B_0 B_1 B_2 \cdots} = \overline{B}_0 + \overline{B}_1 + \overline{B}_2 + \cdots$	T12': $\overline{B_0 + B_1 + B_2 + \cdots} = (\overline{B}_0\, \overline{B}_1\, \overline{B}_2 \cdots)$	De Morgan's Theorem

→ two-input realization of De Morgan's Theorem

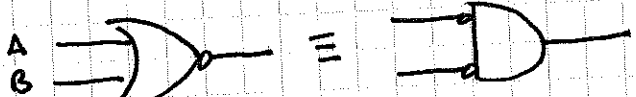
$Y = \overline{AB} = \overline{A} + \overline{B}$



NAND

Nand gate is equivalent to OR gate with inverted inputs

$Y = \overline{A + B} = \overline{A}\,\overline{B}$



NOR gate is equivalent to AND gate with inverted inputs

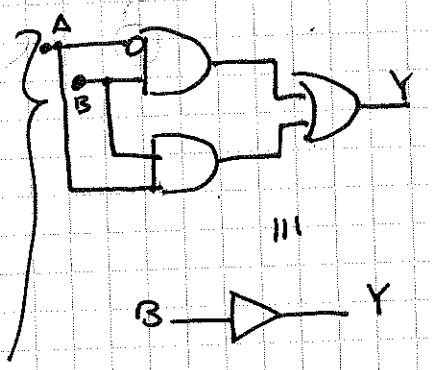
* - can use the theorem to move between sum-of-products and product-of-sums

Theorems, unlike axioms, can be proven. Especially in the case of a finite set of variables one can prove a theorem using a truth table — such a proof is called "perfect induction".

SIMPLIFYING STRATEGIES

Ex.

$$Y = \bar{A}B + AB$$
$$= (\bar{A} + A)B$$
$$= 1 \quad \text{(complements)}$$
$$= B$$



→ the basic approach to simplifying sum-of-products is to use the relationship

$$PA + P\bar{A} = P, \text{ where } P \text{ may be any implicant}$$

ex. $Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$

(1) $= (\bar{A} + A)\bar{B}\bar{C} + A\bar{B}C$

(2) $= \bar{B}\bar{C}(1) + A\bar{B}C$

$= \bar{B}\bar{C} + A\bar{B}C$ → is it the simplest possible?

we can actually replicate terms (idempotency) ... thus

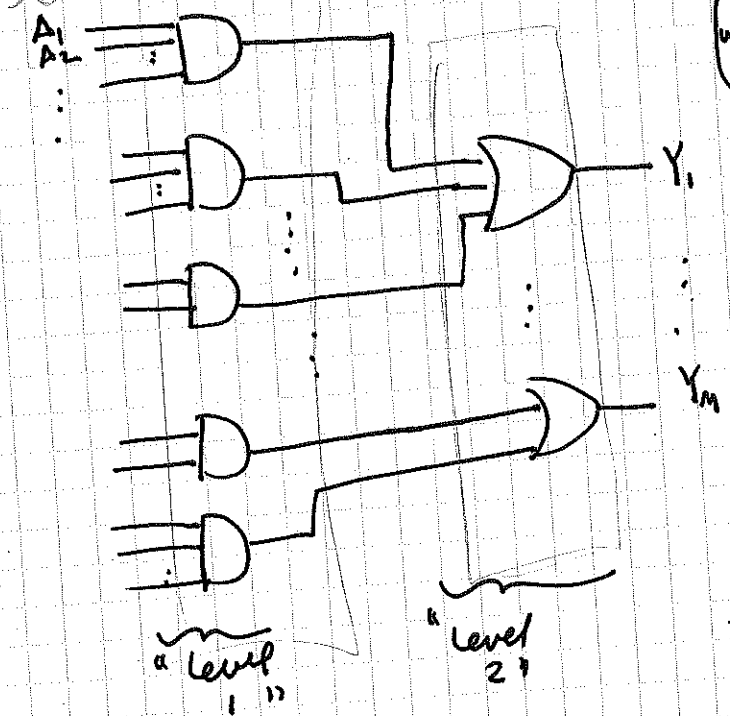
$$Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$$
$$= (\bar{A} + A)\bar{B}\bar{C} + (\bar{C} + C)A\bar{B}$$
$$= \bar{B}\bar{C} + A\bar{B} \quad \text{(which has one fewer and operation)}$$

continued on next page

note: in CMOS (Complementary Metal Oxide Silicon) NANDs and NORs are preferred to ANDs and ORs

MULTI-LEVEL COMB. LOGIC

logic in sum-of-products form is called two-level logic (two levels of gates) (because it consists of literals connected to a level of AND gates connected to a level of OR gates)

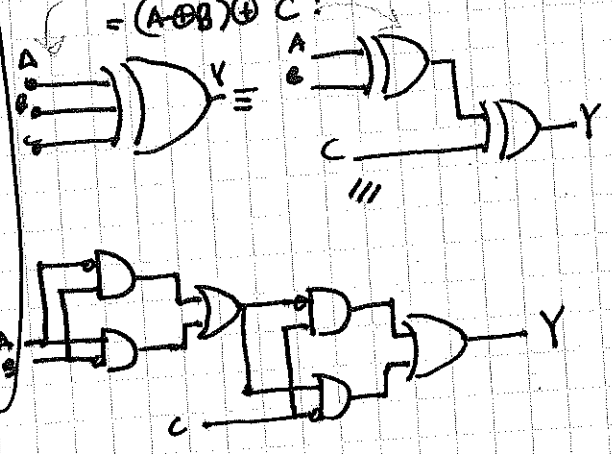


EX. 3-INPUT NOR IS A COMPLICATED CIRCUIT AS SUM OF PRODUCTS... (14 gates to implement)

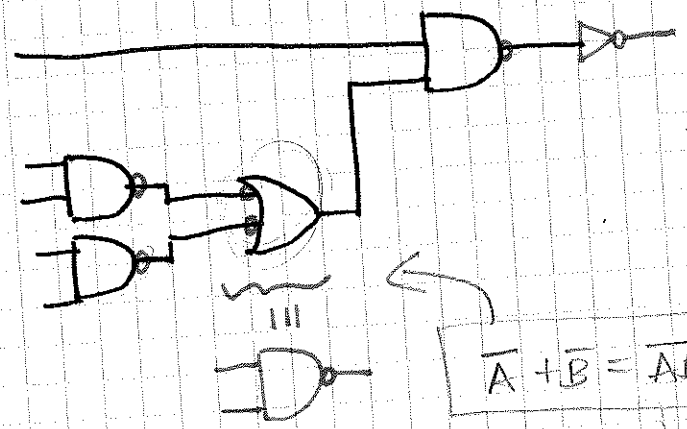
IT CAN BE SIMPLIFIED BY FIRST USING ASSOCIATIVITY:

$Y = A \oplus B \oplus C$

$= (A \oplus B) \oplus C$:



BUBBLE PUSHING — for conversion to NAND/NOR
— also useful in analyzing Multi-level CL ckts.



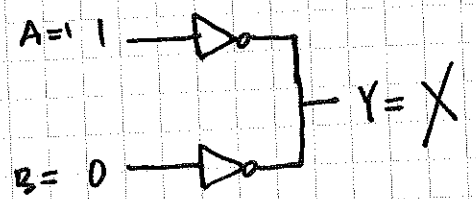
$\overline{A} + \overline{B} = \overline{AB}$

— bubbles and gates added which do not change the logic of the original logic ckt. (if black originally)

next page example

ILLEGAL VALUES : X

(1)



circuit w/ contention, ∩ ERROR that must be resolved.

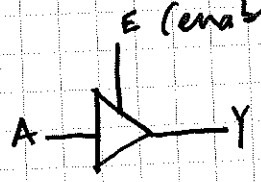
FLOATING VALUE : Z

(2)

- a node is neither being driven high nor low — it is said to be <u>floating</u> or in a high-impedance state.
- not necessarily an error, but there must be resolution of the node value once circuit operation is required.

EX. TRI-STATE BUFFER

(3)

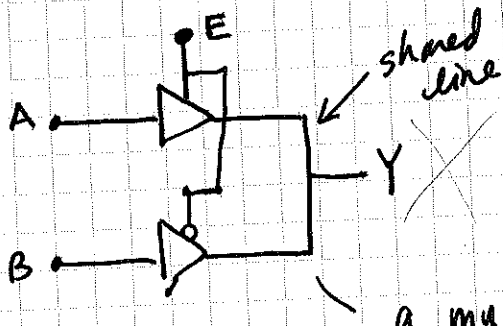


E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

} only "works" in the logic sense when E is HIGH.

→ can be used to resolve contention

(4)



KARNAUGH MAPS

- another innovation from Bell Labs (circa 1950s)
- based on the idea that

$$PA + P\bar{A} = P \quad (A \text{ is eliminated})$$

- rules: ① adjacent squares must only differ by a single variable — need to use Gray codes, e.g., rather than

00, 01, 10, 11

use:

00, 01, 11, 10

or for 3-bit sequence:

000, 001, 010, 011, 100, 101, 110, 111

use: 000, 001, 011, 010, 110, 111, 101, 100

(generalizes)

in other words try to find the largest collection of ones instead of individual ones

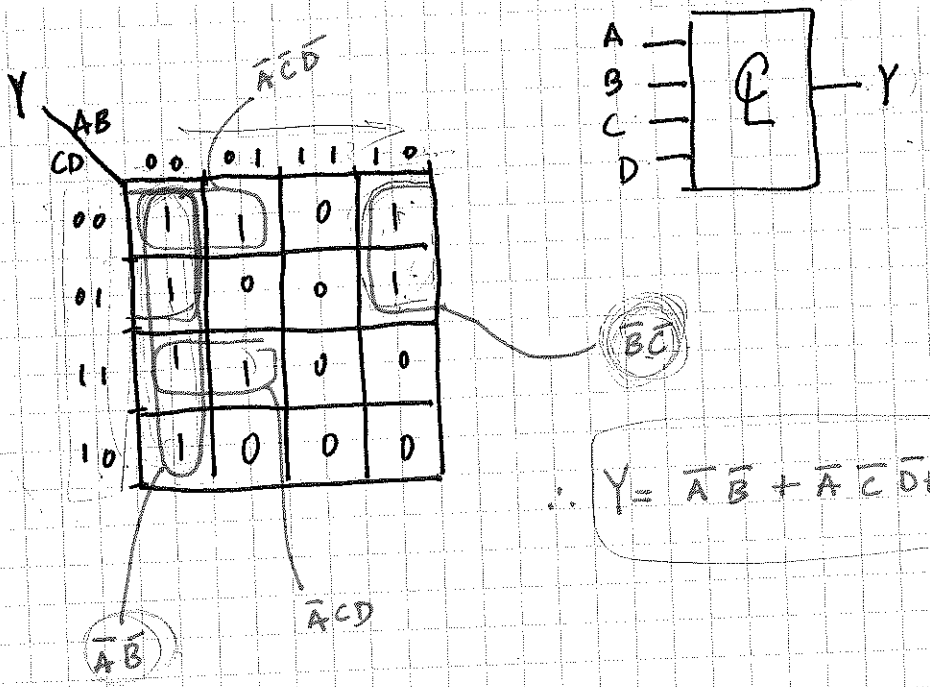
② circle groups of adjacent ones, using the fewest circles needed to cover all the ones
- make circles as large as possible, with only ones inside
- circles must be of sizes which are based on powers of 2, e.g., 1x2, 1x4, 2x2, 4x4, etc.

③ circles may wrap around the edges

④ a one may be circled multiple times

⑤ read off the implicand corresponding to each circle, ignoring the literals whose values change

EX. ↓



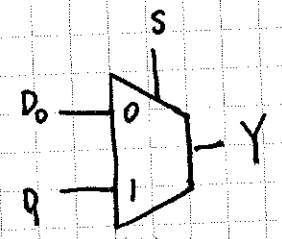
$\therefore Y = \bar{A}\bar{B} + \bar{A}\bar{C}\bar{D} + \bar{A}CD + \bar{B}\bar{C}$

* DON'T CARES CAN BE TREATED AS ONES, ALLOWING FOR LARGER CIRCLES (AND FEWER CIRCLES) — RESULTING IN GREATER LOGIC MINIMIZATION.

MULTIPLEXORS (6)

- multiplexor also called a "mux"
- can be realized using combinational logic.

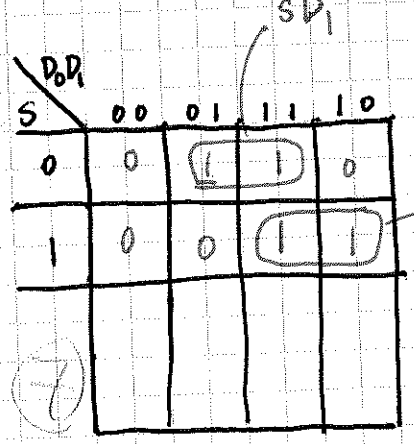
with
- 2 data inputs
- 1 select input S
- 1 output Y



multiplexer

S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

mux is based on 2 data inputs:
1. $S=0, Y=D_0$
2. $S=1, Y=D_1$



$\therefore Y = SD_0 + \bar{S}D_1$

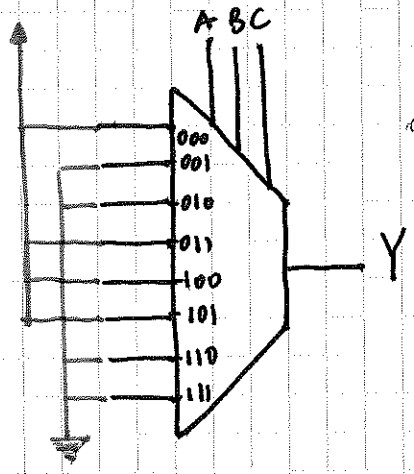
- can also be implemented using tristate logic

MULTIPLEXOR LOGIC

- in general, a 2^N-input multiplexor can be programmed to perform any N-input logic function...

EX. using an 8:1 mux, implement the following function:

$$Y = A\bar{B} + \bar{B}\bar{C} + \bar{A}BC$$



called Alyssa's implementation

truth table:

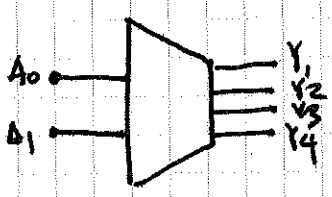
A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

VII

~~COST~~ you can also incorporate one of literals as an input, allowing the size of the multiplexor to be cut in half.

DECODER

(8) - essentially (essentially) enumerates the minterms... using a vector-valued output...

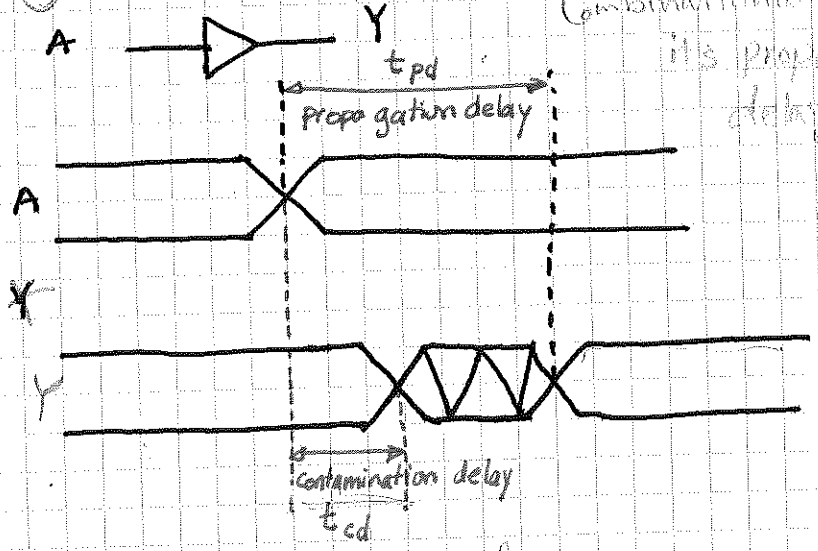


A_1	A_0	Y_1	Y_2	Y_3	Y_4	minterm
0	0	1	0	0	0	$\bar{A}_1\bar{A}_0$
0	1	0	1	0	0	$\bar{A}_1 A_0$
1	0	0	0	1	0	$A_1\bar{A}_0$
1	1	0	0	0	1	$A_1 A_0$

→ can use an OR gate to realize logic based on sum-of-products...

TIMING (1)

(combinational logic is characterized by its propagation delay & contamination delay)



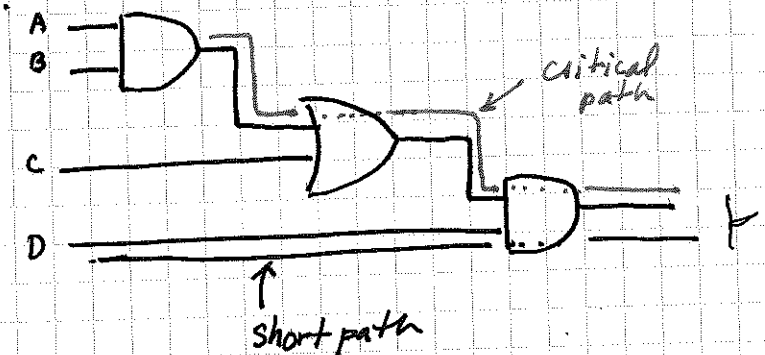
- delays exist owing to gate-capacitance charge times, and the speed of charge propagation in the circuit, generally on the order of nano seconds (10^{-9}) or pico-seconds (10^{-12})

t_{pd} : (maximum) time from when an input changes until the outputs reach their final value

t_{cd} : (minimum) time from when an input changes value until any output starts to change its value

- signals propagate according to paths, which can vary in "length" owing mostly to the number of intervening gates

EX.



- propagation delay is often associated with the critical path
- contamination delay is often associated with the short path

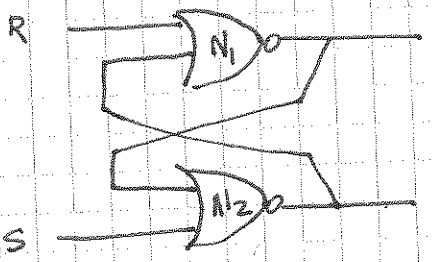
(2)

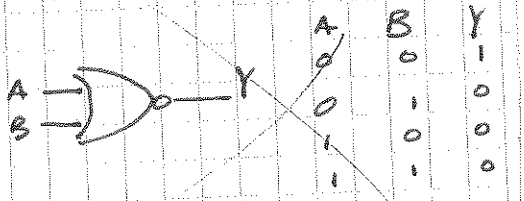
→ GLITCHES are the result of differences in path lengths which can cause the output to change state; essentially, a single variable crosses the boundary between two implicants in the K-map; we can introduce additional logic to produce redundant implicants and eliminate the glitch if necessary (at the expense of extra hardware) — see 2.9 of text

SEQUENTIAL LOGIC DESIGN

- logic with "states" or memory
- a fundamental building block of memory is the bistable element, a circuit with two stable states. ← latch

SR Latch (3)



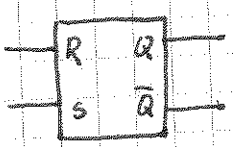


can analyze as a series of cases, as shown in your text, section 3.2.1, summarized as

S	R	Q	$\bar{Q}$	
0	0	Q_{prev}	$\bar{Q}_{prev}$	← nothing changes
0	1	0	1	
1	0	1	0	
1	1	0	0	

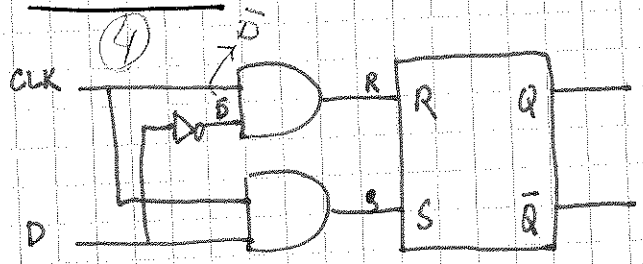
"normal use"

symbolically:



D Latch – separates the ideas of "what" and "when"

(4)



we avoid the awkward S=1, R=1 case

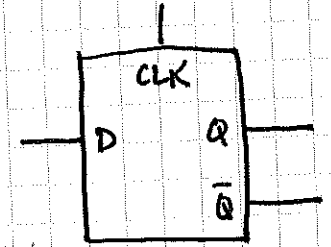
(5)

CLK	D	$\bar{D}$	S	R	Q	$\bar{Q}$
0	X	$\bar{X}$	0	0	Q_{prev}	$\bar{Q}_{prev}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0

X – don't care state

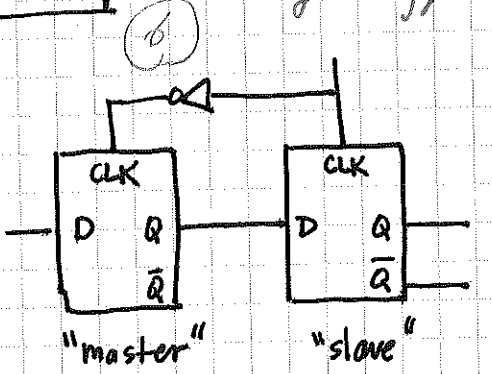
nothing changes unless CLK is HIGH

symbol:

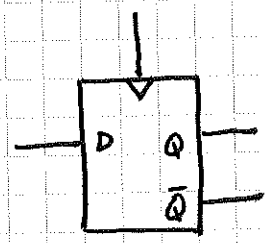


latch is said to be "opaque" – it BLOCKS new data

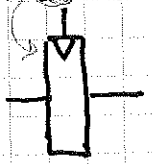
D Flip-Flop – edge-triggered flip-flop



symbolically:



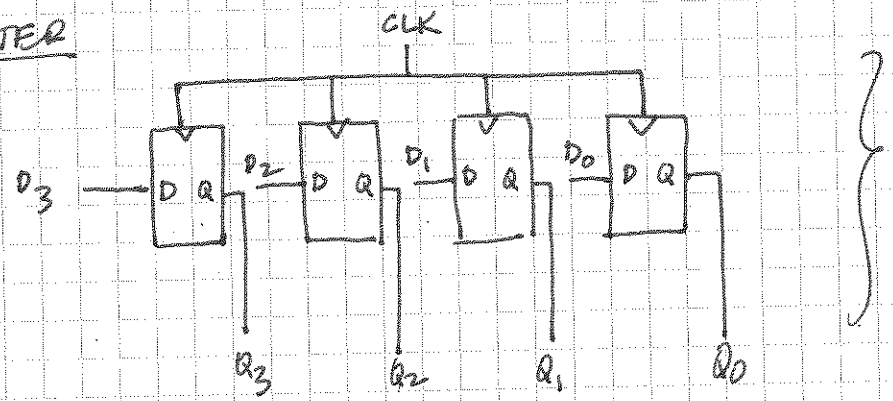
or when $\bar{Q}$ is needed (not)



ignore $\bar{Q}$

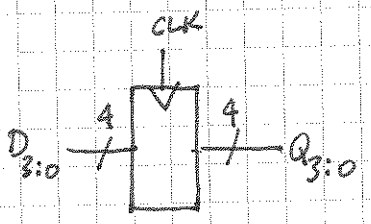
The D flip-flop copies D to Q on the rising edge of the clock, and remembers its state at all other times.

REGISTER

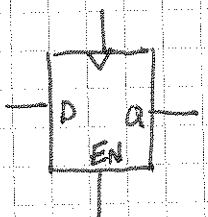


4-bit example

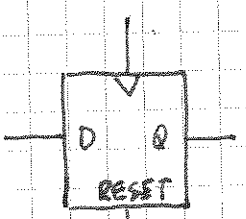
- all bits are updated at the same time
- also written as:



OTHER FLIP FLOPS:



enabled flip-flop
enabled.



(synchronously) resettable flip-flop

(synchronously) resettable

timing considerations in sequential logic...
Considerations

- race conditions can occur when the behaviour of the circuit depends on which path through a logic circuit is the fastest
- delay-dependent behaviour is undesirable and are considered to be malfunctions.
- cyclic paths can be challenging to design correctly! due to the fact that outputs are fed directly back to inputs!
- to AVOID, designers often insert registers into paths effectively creating a collection of combinational circuits and registers — simpler to analyze!

Synchronous Sequential Circuits

If the clock is sufficiently slow, so that all inputs to the registers settle before the next clock edge, ALL RACES ARE → before ELIMINATED

↓

SYNCHRONOUS SEQUENTIAL CIRCUITS

↓ sefat

key attributes:

- has a finite set of many discrete states $\{S_0, S_1, \ldots, S_{k-1}\}$ haalat.
- has a clock input, whose rising edges indicate a sequence of times at which state transitions can occur

(1)

- t_{pcq} : time of propagation from CLK to Q
- t_{cq} : time of contamination from CLK to Q

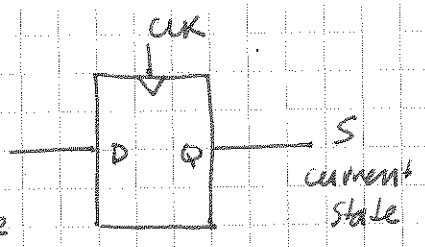
Pcq ← tpcq
ccq ← tcq

RULES:

- every cyclic path contains at least one register
- all registers receive the same clock signal
- every cct. element is either a register or CL → Combinational Circuit
- at least one cct. element is a register
  - somewhat conservative
    + (Somewhat (conservative))

circuit → circuit

Ex. the simplest synchronous sequential ckt.



a flip-flop is a simplest synchronous

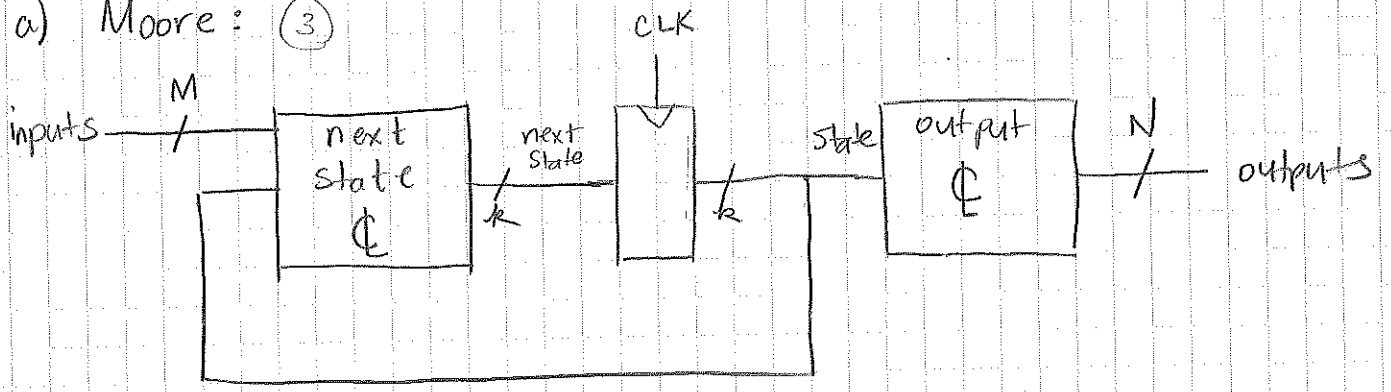
it has one input, D, one clock, one output, Q, and two states $\{0, 1\}$

UNIT 1

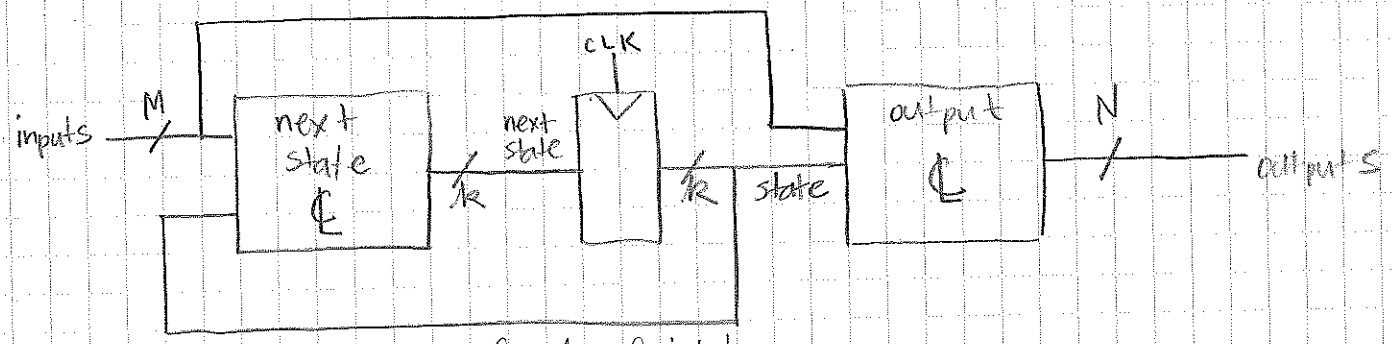
FINITE STATE MACHINES (2)

- an approach for <u>synchronous sequential logic</u>.
- we consider two forms:

a) Moore: (3)



b) Mealy: (4)

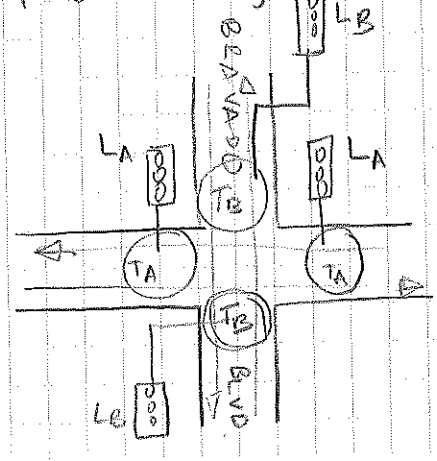


- both Moore (not Gordon, but Edward) and Mealy developed Automata Theory while working at Bell Labs in the 1950s.

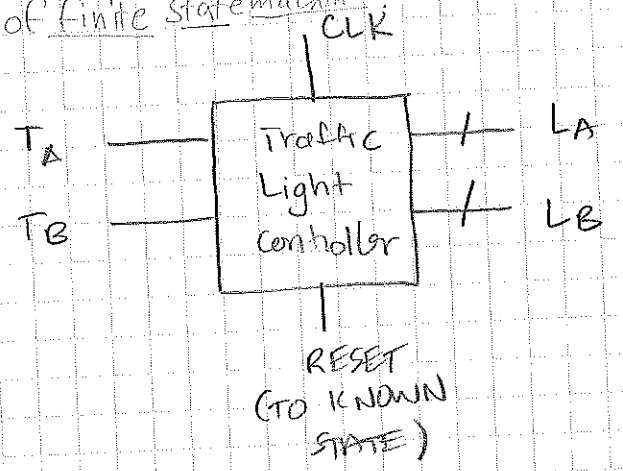
↑ founder of Intel

- Design Example (from text):

(5)



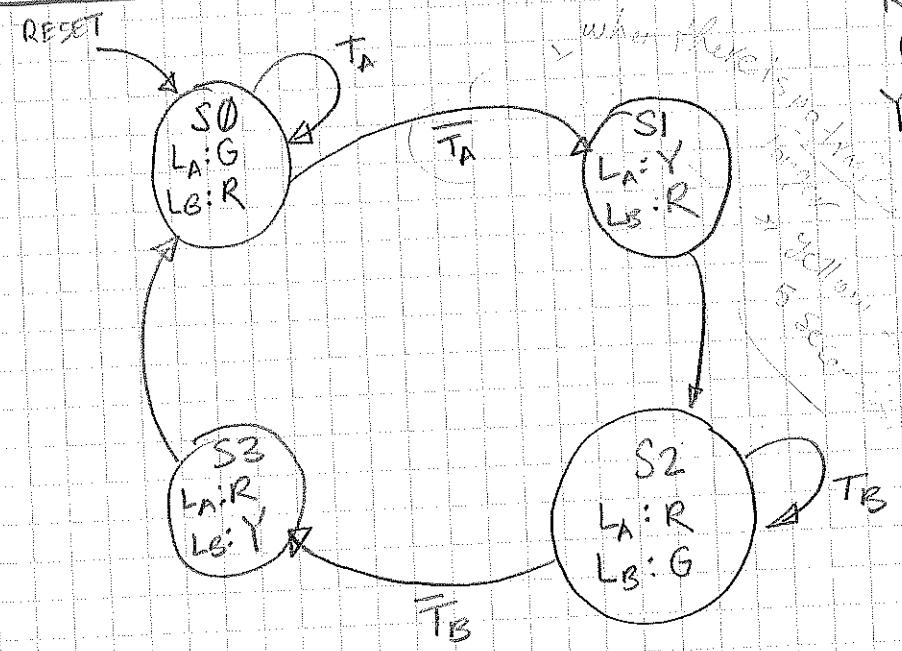
black Box View of finite state machines:



- the RESET signal initializes the system to GREEN on ACADEMIC AVE and RED on BRAVADO BLVD.
- as long as traffic is present on AA, the light remains GREEN. If traffic stops on AA, then the light turns YELLOW followed by RED.
- similarly, the lights on BB remain GREEN as long as traffic is present.

STATE TRANSITION DIAGRAM

R: RED
G: GREEN
Y: YELLOW



CLOCK PERIOD: 5 s.

STATE TRANSITION TABLE

(6)

current state, S	inputs T_A	inputs T_B	next state, S'
S0	1	X	S0
S0	0	X	S1
S1	X	X	S2
S2	X	1	S2
S2	X	0	S3
S3	X	X	S0

whenever the next state (X): doesn't depend on a particular input

*: don't care

binary encodings

$S_{1:0} = \{00, 01, 10, 11\}$
$S_0 \quad S_1 \quad S_2 \quad S_3$

$L_{1:0} = \{$ GREEN, YELLOW, RED $\}$
$00 \quad 01 \quad 10$

TABLE WITH BINARY ENCODINGS:

S_1	S_0	T_A	T_B	S_1'	S_0'
0	0	1	X	0	0
0	0	0	X	0	1
0	1	X	X	1	0
1	0	X	1	1	0
1	0	X	0	1	1
1	1	X	X	0	0

sum of products for next state logic:

$$S_1' = \bar{S}_1 S_0 + S_1 \bar{S}_0 T_B + S_1 \bar{S}_0 \bar{T}_B = \bar{S}_1 S_0 + S_1 \bar{S}_0 = S_1 \oplus S_0$$

(by inspection)

$$S_0' = \bar{S}_1 \bar{S}_0 \bar{T}_A + S_1 \bar{S}_0 \bar{T}_B$$

OUTPUT TABLE

- state → outputs

S_1	S_0	L_{A_1}	L_{A_0}	L_{B_1}	L_{B_0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

Therefore:

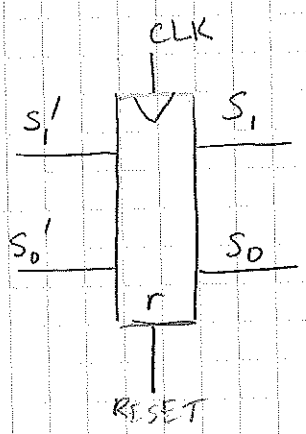
$$L_{A_1} = S_1$$

$$L_{A_0} = \bar{S}_1 S_0$$

$$L_{B_1} = \bar{S}_1 \bar{S}_0 + \bar{S}_1 S_0 = \bar{S}_1$$

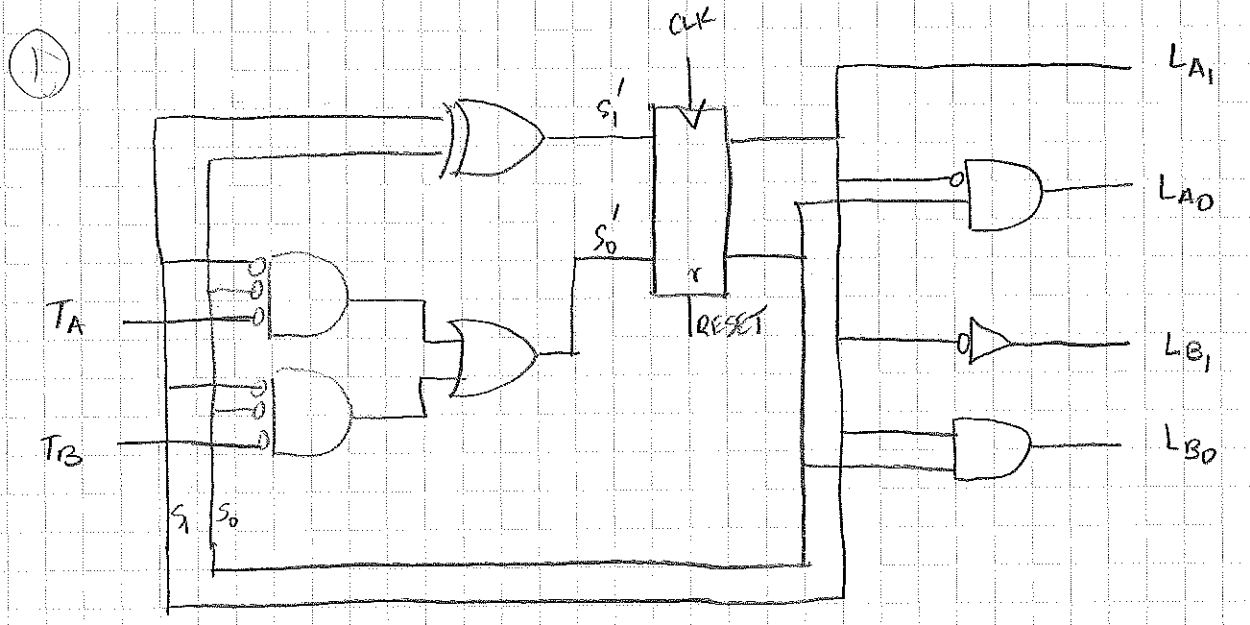
$$L_{B_0} = S_1 S_0$$

REGISTER



on the rising edge of each clock, it copies $S_1'S_0'$ to S_1S_0 so that the current state becomes the next state

state-machine circuit:



(1)

- note that implementation depends on the choice of state and output encodings — therefore, some encodings may produce simpler (and more cost-effective) implementations than others.

→ there is no general approach to find the best encodings, but we can use inspection and CAD tools to help us optimize.

Moore vs. Mealy

Explained in an example:

- consider a robotic snail that is reading a binary signal painted on the floor — each time it encounters a "01" it "smiles" and it crawls along, reading in a new bit each cycle.

(2)

smiles

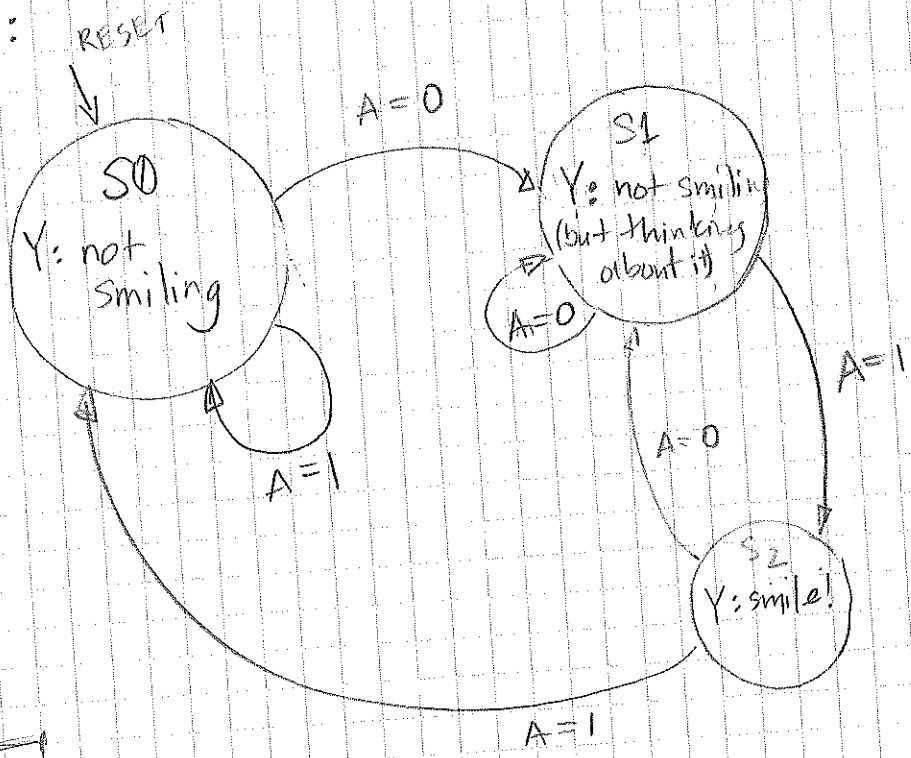
crawls

Robot Snail Example from Textbook

input: A — current bit read

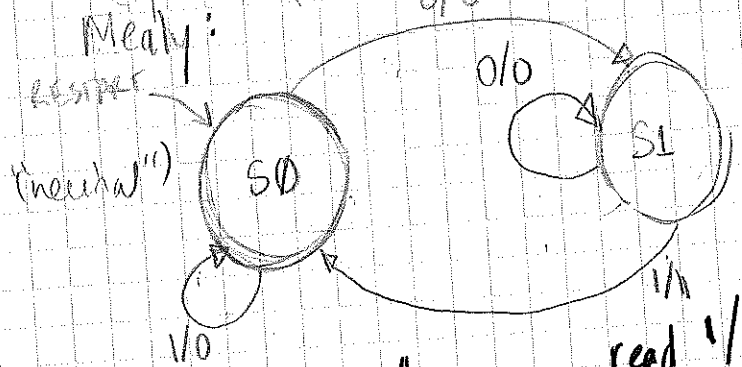
output: smile / not smile

Moore:



0 1 0 0 1 0 1 1 0 0 0 read 0 / "not smiling"

Mealy:



read 1 / "not smiling"

read 1 / "smiling"

1/1

1/0

each Arc is labeled as A/Y

- Moore State Transition Table

Current State S	Input A	Next State S'
S0	0	S1
S0	1	S0
S1	0	S2
S1	1	S1
S2	0	S1
S2	1	S0

with binary encodings

S_1 S_0	A	S_1' S_0'
0 0	0	0 1
0 0	1	0 0
0 1	0	0 1
0 1	1	1 0
1 0	0	0 1
1 0	1	0 0

S0 = 00
S1 = 01
S2 = 10

$S_1' = \bar{S}_1 S_0 A$

$S_0' = \bar{S}_1 S_0 \bar{A} + S_1 \bar{S}_0 \bar{A} + \bar{S}_1 S_0 A$

$= \bar{A}(\bar{S}_1 S_0 + S_1 \bar{S}_0) + \bar{S}_1 S_0$ (1)

$\Rightarrow \bar{A}(1 + S_1 S_0)$

$\Rightarrow$

- Moore Output Table

Current State S	Output Y
S0	0
S1	0
S2	1

S	Y
S_1 S_0	
0 0	0
0 1	0
1 0	1

$Y = S_1$

- Mealy State Transition and Output Table

S	A	S'	Y
S0	0	S1	0
S0	1	S0	0
S1	0	S1	0
S1	1	S0	1

binary encodings:

S_0	A	S_0'	Y
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

$S_0' = \overline{A}$ → $S_0 A + S_0 \overline{A}$

$Y = S_0 A$ → $(A)(S_0 + S_0)$

T

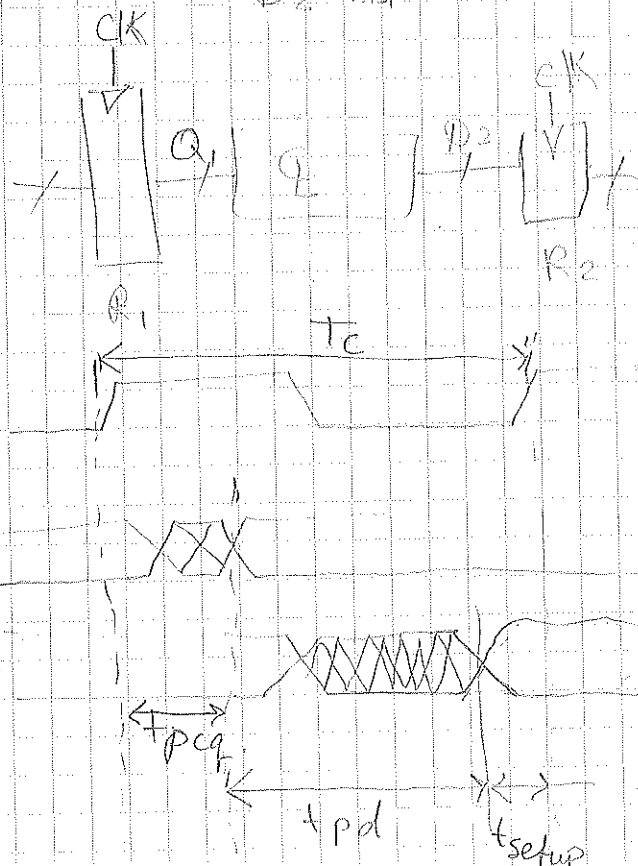
- additional topics:
  - FACTORING FSMs — breaking a problem down into simpler pieces
    - breaking a problem down into simpler pieces
  - DERIVING FSMs FROM SCHEMATICS
  - THE DYNAMIC DISCIPLINE
    - aperture — the setup time (before the clock edge) and the hold time (after the clock edge), during which the digital input (aperture) remains stable
      - input → output settling

Cycle time

$T_{c1} \geq t_{pcq} + t_{pd} + t_{setup}$

↑ clock to Q propagation delay (first stage)

to satisfy the setup time of R_2 → D_2 must be "stable"



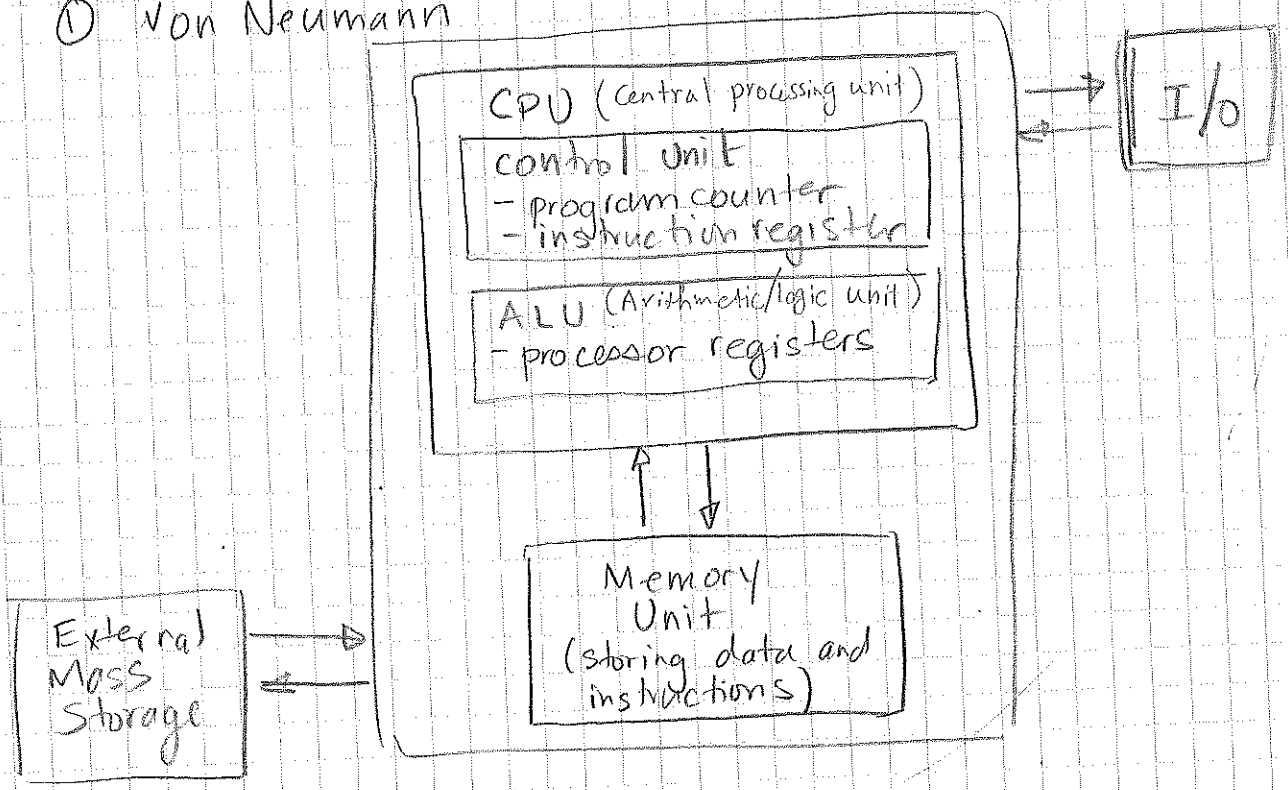
UNIT 1

STORED PROGRAM COMPUTER ARCHITECTURES

two kinds we consider:
- von Neumann (Princeton) architecture
- Harvard architecture

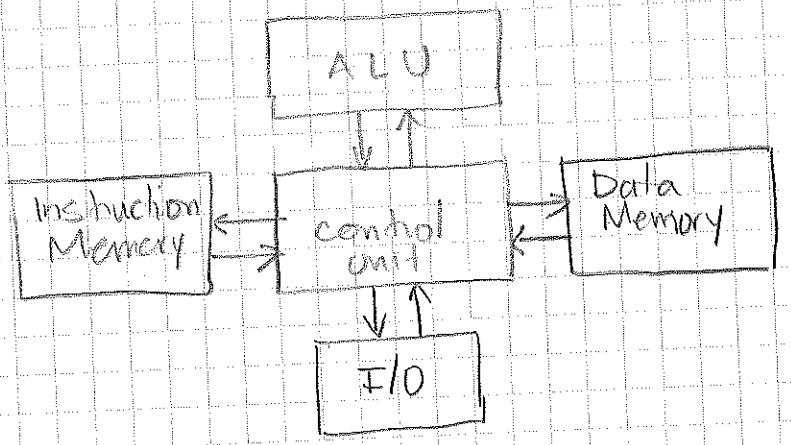
- original programmable computer architecture is, possibly Turing's "universal computing machine," now known as the Universal Turing Machine (1)
- stored program computer

① von Neumann



- instruction fetch and a data operation (read/writ cannot occur simultaneously because they share a common bus (common)
- it is a simplified design compared to Harvard architecture, but this "von Neumann bottleneck" limits the speed of the machine

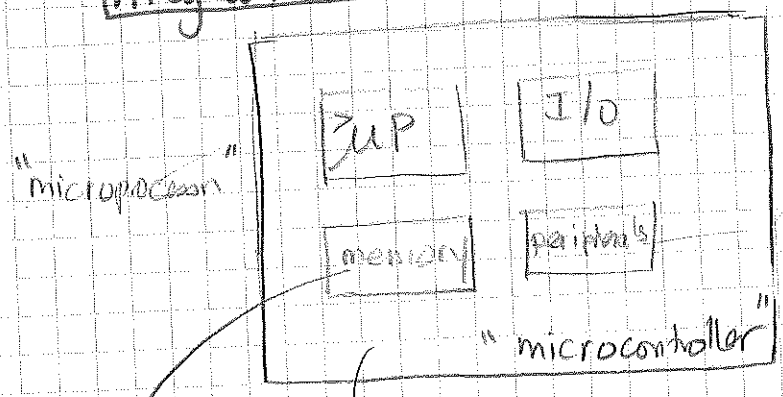
(2) Harvard architecture
- separate and independently accessible instruction and data storage



(2) • has separate sizes for instruction and data memories (memories can differ (as can associated word lengths)

• elements of Harvard Architecture are used effectively in modern DSP and microcontrollers by companies like TI, Microchip and Atmel.

Integration Methods



often flash ROM w/ a small amount of RAM

"digital signal processors (specialized microprocessor with its architecture optimized for the operational needs of digital signal processing)"

the microcontroller or MCU is an example of a system-on-a-chip (SoC) → a system integrated on a single silicon die.

- often these components are reduced in size and complexity compared with the components associated with powerful microprocessors, but can offer considerable functionality at

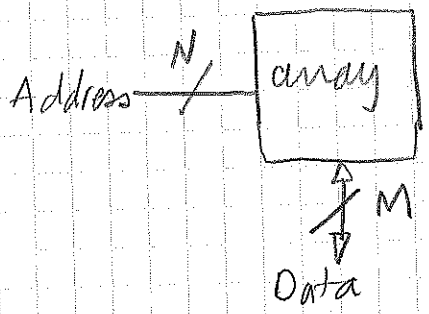
Unit 2

Memory Arrays

- building blocks that can efficiently store large amounts of data
- three types:
  1. Dynamic Random Access Memory (DRAM)
  2. Static Random Access Memory (SRAM)
  3. Read-Only Memory (ROM)

Overview

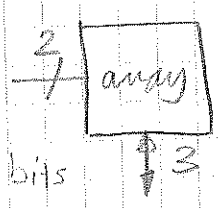
- a generic memory array



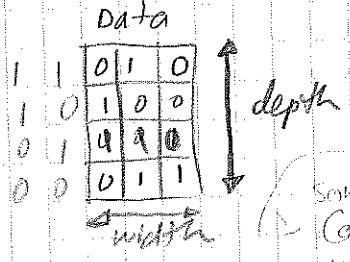
2^N M-bit words are contained in this (3)

- memory is organized (arranged) as a two-dimensional array of "memory cells" (1 bit)
- we read or write one row of this array, and rows are identified by an address.
  - the value read or written is called data

ex.

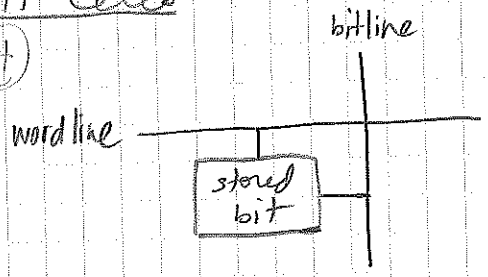


Consists of:
2 address bits
3 data bits
↳ columns → # of Rows → $2^2 = 4$



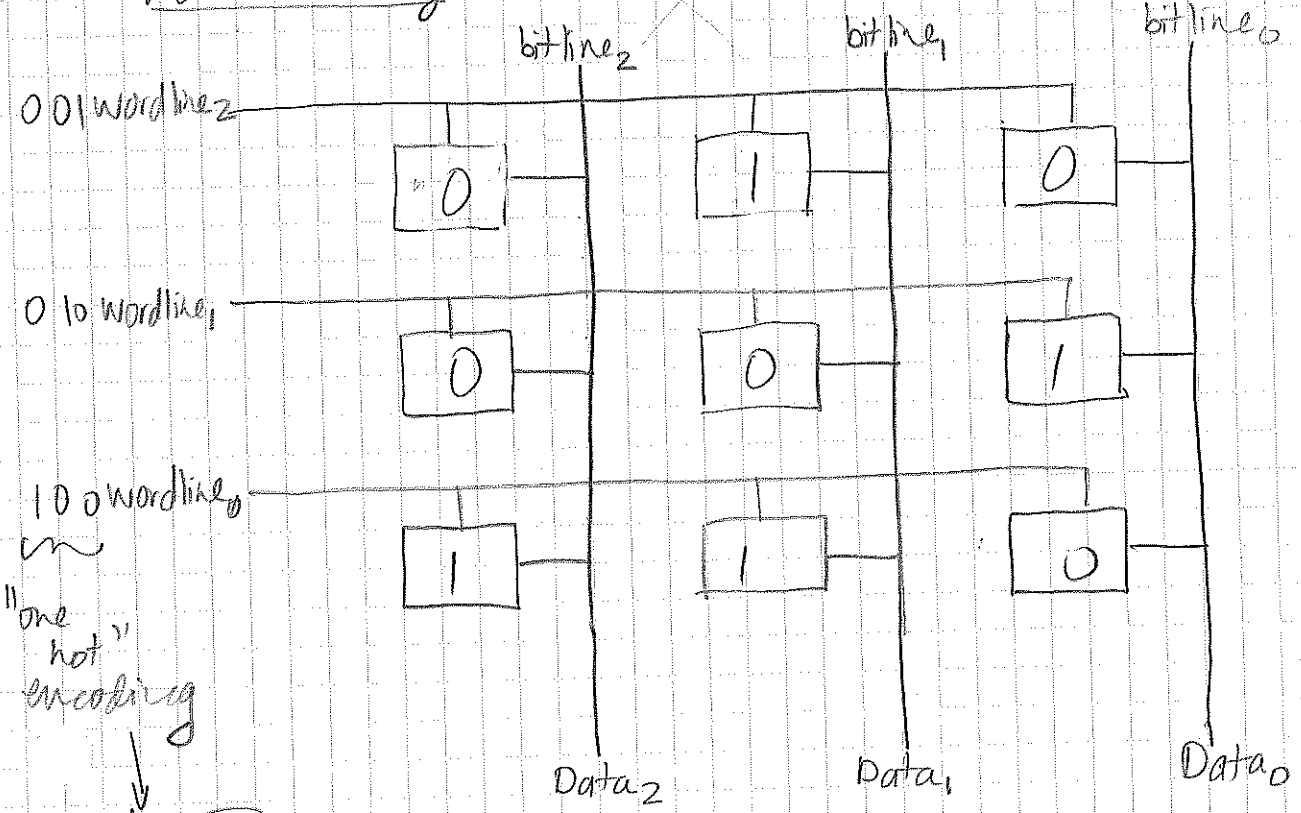
(Some possible Content of a Memory Array)

Bit Cells (4)



- to read a bit, bit line is left floating, initially.
  - then the wordline is turned ON, allowing the stored value to drive the bitline to 0 or 1.
- to write a bit, the bit line is DRIVEN (drive) to the desired value
  - the wordline is then turned ON, allowing the bit line (driven strongly) to overpower the content of the cell, writing the desired value to the stored bit.

bit cell array: do (5)

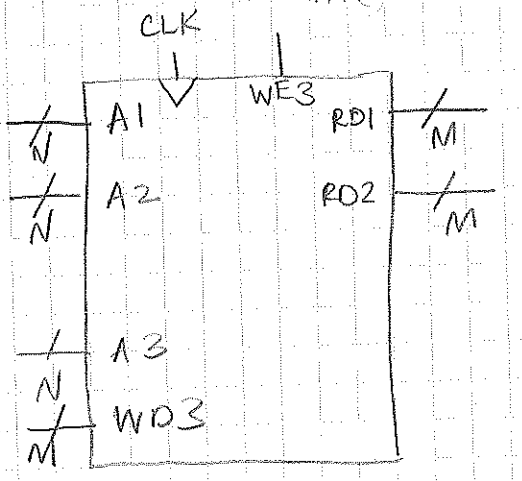


"one hot" encoding

need a decoder

Memory Ports (Ports)

- each port gives read/write access to one memory address — previous examples were single-ported memory arrays.
  single-ported
- an example of a memory w/ two read ports and one write port:



read data from address A1 onto the read data output RD1

PORT 1: A1 + RD1

PORT 2: A2 + RD2

PORT 3: A3 + WD3

(writes on rising edge of CLK if WE3 is asserted)

memory classifications:

Digital Memory

- RAM (random access memory)
  - "random access" means that the delay associated with accessing any element is constant
    (accessing)
- ROM (read-only memory)
  - modern ROMs are also random access and can be written!

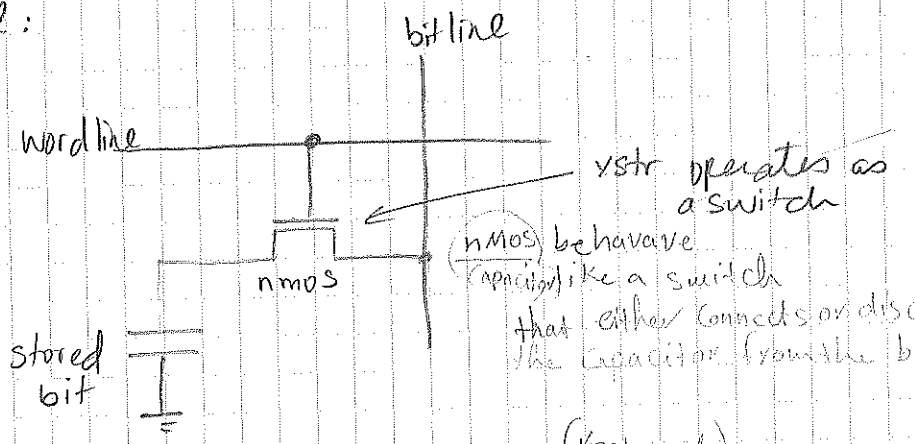
major distinction is → volatile

major distinction is that RAM is volatile (needs power) and ROM is non-volatile
non-volatile

DRAM (dee-RAM) (Dynamic Random access memory)

- bit value is stored on a capacitor

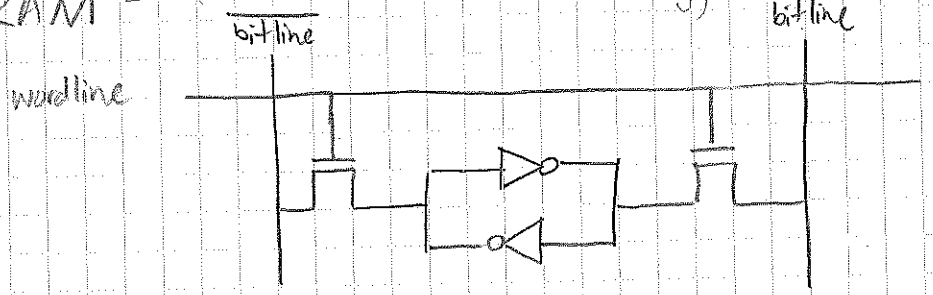
DRAM bit cell:



(restored)
1b, stored

- reading destroys stored value (on capacitor), so the word must be rewritten after each read because of leakage, the charge on the capacitor must be restored every few milliseconds.
→ refreshed.

SRAM → (static Random access memory)



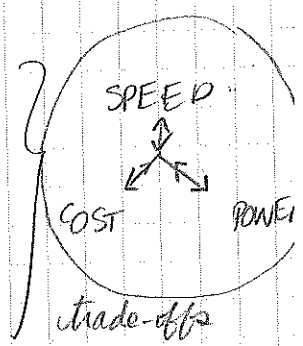
- static RAM because stored bits do not need to be refreshed — the cross coupled inverters are regenerative
Cross-coupled inverters.

- each cell has two inputs/outputs, bitline and $\overline{\text{bitline}}$

Volatile Memories

- include DRAM, SRAM and FFs
flip flops.

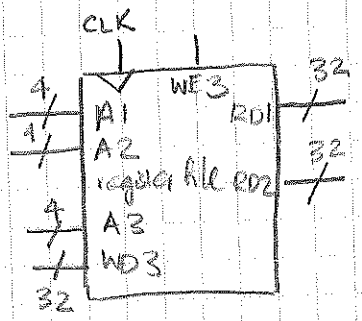
type	xstrs per bit cell	latency
flip-flop	~20	fast
SRAM	6	medium
DRAM	1	slow



Register File (1)

- usually an SRAM array with multiple ports
- more compact than an array of flip-flops

ex. 16 register x 32 bit three ported register file



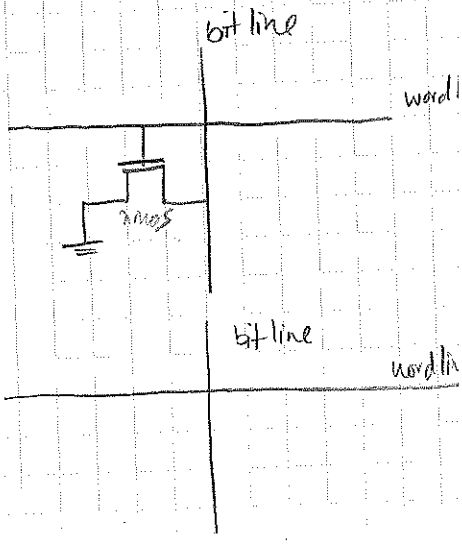
two read ports, one write port

ROM

- stores a bit as the presence or absence of a transistor
- to read the cell, the bitline is weakly pulled high, then the wordline is turned ON
  - if a transistor is present, it pulls the bitline LOW, if absent, the bitline remains HIGH

schematics:

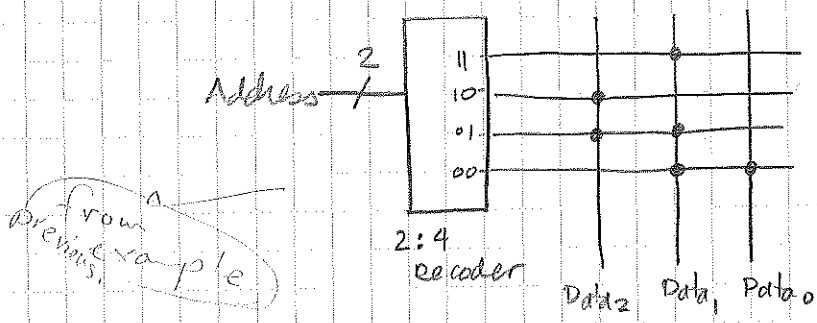
bit cell containing 0:



bit cell containing 1:

we can represent the contents (Contents) of a ROM using dot notation

EX. 4-word by 3-bit ROM



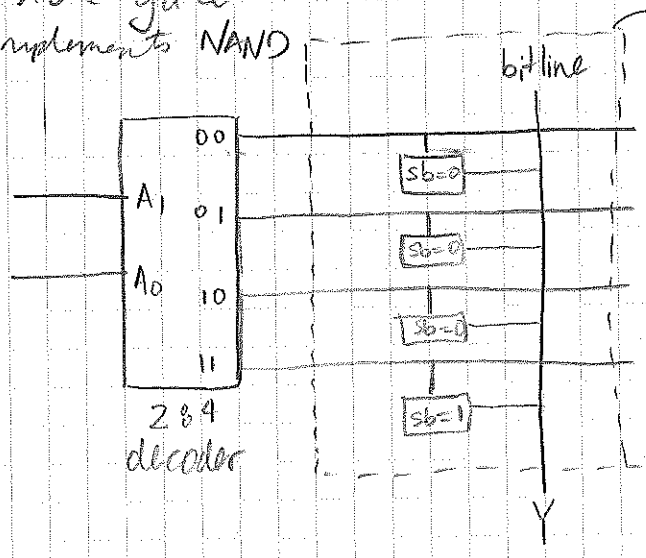
address	contents (data)
0 0	0 1 1
0 1	1 1 0
1 0	1 0 0
1 1	0 1 0

ROMs can be used to perform combinational logic functions using this technique.

- in the above, Data$_2$ is HIGH if input is either 01 or 10 → an XOR gate
- Data$_0$ implements NAND

(2)

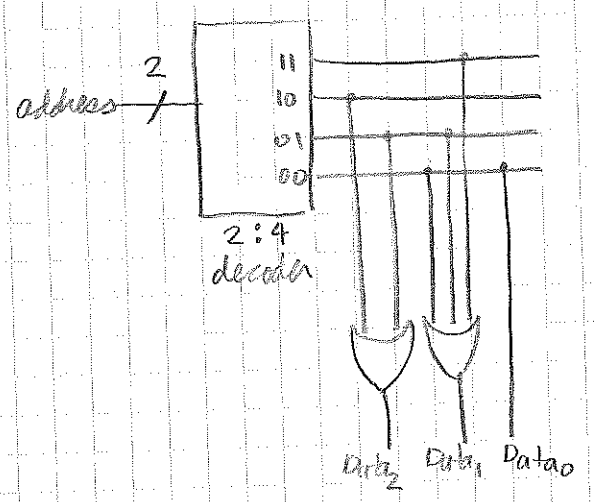
(3) in general, for a 2-input logic function



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

$Y = AB$

- also, a ROM can be realized using combinational logic
- we can build the above 4-word 3-bit example as

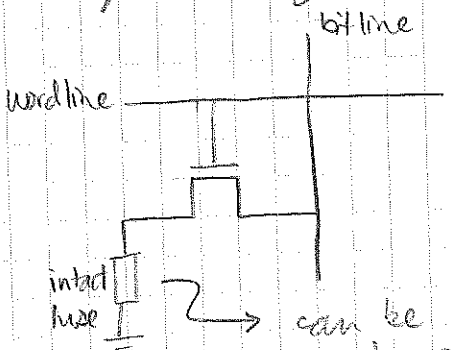


this effectively "hard codes" a ROM, and some ROMs such as MROM (masked ROM) are programmed by the IC manufacturer and cannot be (re)programmed by the user

PROM

(4) - programmable ROM

- a one-time programmable ROM uses fuses which are blown (permanently opened) using a high strength signal:



→ possibly used as MROM

→ can be blown → blown. to give the bit cell a value of 1

- other variations:
  - EPROM — can use UV light to turn off transistors (UV-EPROM)
  - EEPROM (electrically erasable) and Flash memories use on-chip circuitry for programming bit cells.

Home work: 5.55, 5.57

Flash memory (unlike EPROMs) does not have to be completely erased before being rewritten —

NAND flash — memory is read and written in blocks (which are much smaller than the total memory)

don't write ✗ not a random access memory, it is more like a mass storage device
- high density

NOR flash — allows a single word to be read or written independently

(but has long erase and write times)

→ good when rare updates are needed (as in the case of BIOS or bootloader code)

boot loader.

don't write ✗ — all flash has finite endurance, from as few as 100 up to 1 000 000 erase cycles

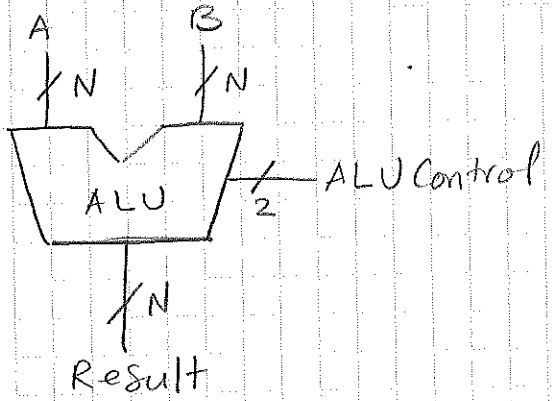
→ flash memory must be managed

to spread out erasure of blocks evenly in order to maximize the lifetime of the device.

UNIT 3

*do this before Programmed I/O for Lab 5

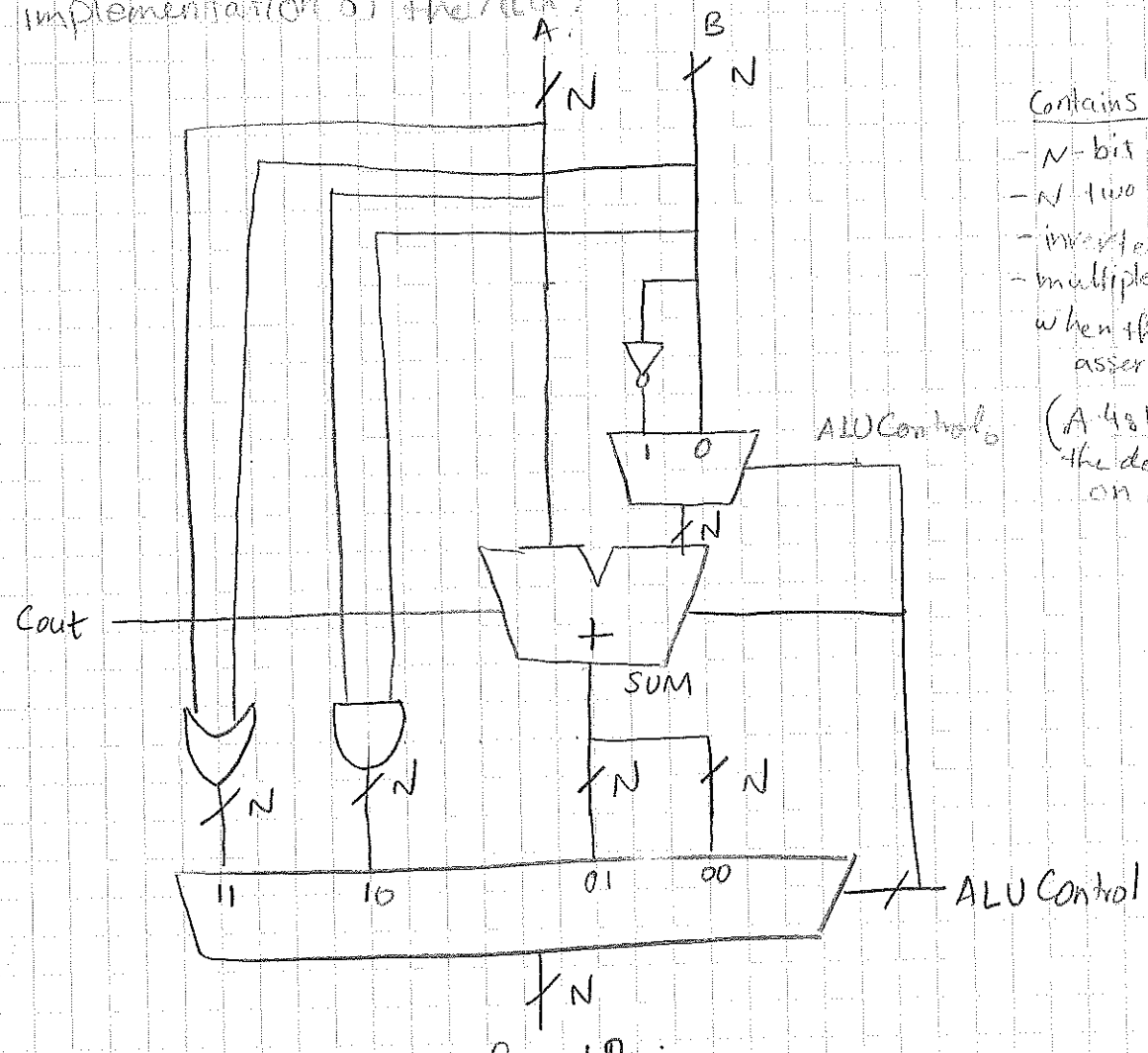
1. The ALU (Arithmetic/Logical Unit)



ALUControl$_{1,0}$	Function
00	Add
01	Subtract
10	AND
11	OR

the list of typical functions that ALU can perform.

implementation of the ALU:



Contains:
- N-bit adder
- N two input AND & OR gates
- inverters
- multiplexer to invert inputs when the ALUControl$_0$ is asserted.

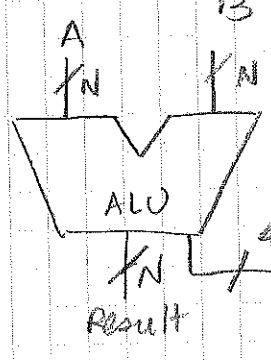
(A 4:1 multiplexer chooses the desired function based on ALUControl)

(2)

- some ALUs produce output flags, such as
  - oVerflow
  - Carry
  - Negative
  - Zero

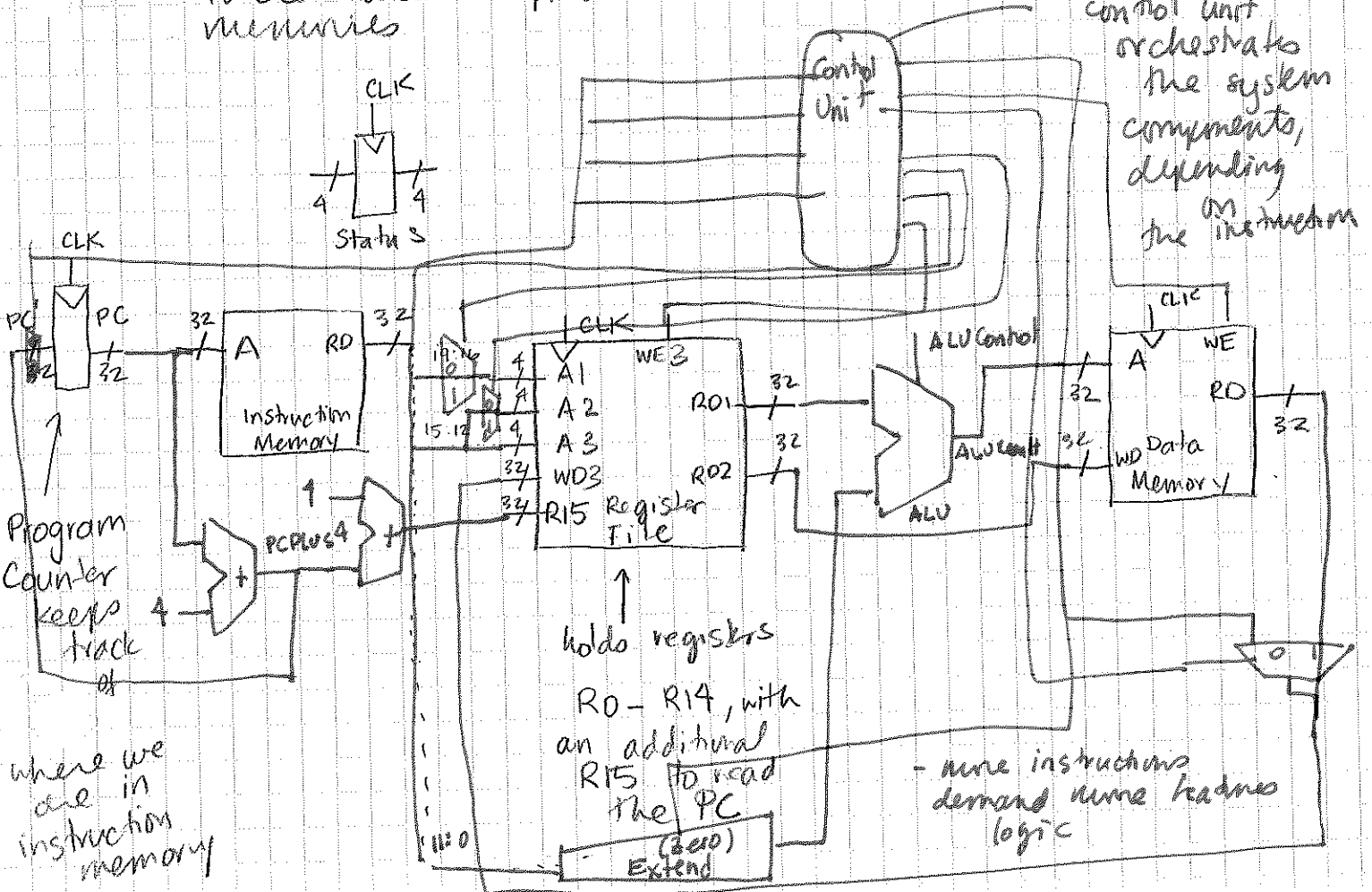
} conditions detected using combinational logic circuits added to the basic N-bit ALU design, above.

carry out becomes a flag



- ARM 32-bit Micro Architecture — a big FSM (or collection of FSMs!)

① we begin with state elements — notice how there are separate instruction and data memories



- the processor microarchitecture is built to realize various instructions

UNIT 2

ARM Architecture

- we're going to jump up a couple levels in the hierarchy to examine architecture.
  hierarchy
- the architecture is the "programmer's view" of the computing machine
- each processor has a native language comprised of words we call instructions which are themselves comprised of bits (ones and zeros), like words in a language are comprised of letters.
- the instruction set is the complete set of instructions used by an architecture, and in the computing world, we have two major categories of architecture, distinguished by their instruction sets, as:
  1. CISC – complex instruction set computer
  2. RISC – reduced instruction set computer

our focus ←

- each of these represents a distinct philosophy, and the goal ultimately is allowing programs to execute as quickly as possible. sometimes CISC seems right, other times RISC leads the way...
- the instructions can be expressed in binary form, or what is known as machine language; alternatively, because the binary form is challenging for humans to remember, we can use mnemonics, which are simple english-word versions of each instruction

- therefore, assembly language is simply the human-readable version of machine language.

EXAMPLES

High-level Code	ARM Assembly Code
`a = b+c;`	`ADD a, b, c`
`a = b-c;`	`SUB a, b, c`

`a = b+c-d; // single-line comment`	`ADD t, b, c ; t = b+c`
	`SUB a, t, d ; a = t-d`
	↑ single-line comments

Register operands.

```
a = b+c;
```

```
; R0 = a, R1 = b, R2 = c
ADD R0, R1, R2 ; a = b+c
```

Highlevel code

```
a = b-c;
f = (g+h) - (i+j);
```

```
; R0 = a, R1 = b, R2 = c
; R3 = f, R4 = g, R5 = h
; R6 = i, R7 = j
SUB R0, R1, R2 ; a = b-c
ADD R8, R4, R5 ; R8 = g+h
ADD R9, R6, R7 ; R9 = i+j
SUB R3, R8, R9 ; R3 = f
```

Immediate operands.

```
a = a+4;
b = a-12;
```

```
; R7 = a, R8 = b
ADD R7, R7, #4 ; a = a+4
SUB R8, R7, #0xC ; b = a-12
```

```
if (apples == oranges)
    f = i+1;
f = f-i;
```

```
; R0 = apples, R1 = oranges
; R2 = f, R3 = i
    CMP R0, R1 ; apples == oranges?
    BNE L1 ; if NE, skip next
    ADD R2, R3, #1 ; f = i+1
L1  SUB R2, R2, R3 ; f = f-i
```

- while CISC machines tend to make it easier to do more at the machine language level, RISC machines are based on a different philosophy:

① regularity supports simplicity → ARM: consistent number of operands (three: one destination and two source operands)
② make the common case fast
③ smaller is faster → ARM: the fewer the registers, the faster they can be accessed
④ good design demands good compromises.

ARM: includes only simple, commonly used instructions

READ: 6.1 → 6.3

Operands: Registers, Memory, and Constants

- 16 ARM registers:

R0 – R12 : variable storage
R13 : SP (stack pointer)
R14 : LR (link register)
R15 : PC (program counter)

- constants or immediates also allowed; values come directly from the instruction itself

Ex.

```
i = 0;
x = 4080;
```

```
; R4 = i, R5 = x
MOV R4, #0 ; i=0
MOV R5, #0xFF0 ; x=4080
```

move instruction

- ARM provides a load register, LDR, to read a data word from memory into a register

High-Level Code

```
a = mem[2];
```

ARM Assembly Code

```
; R7 = a
MOV  R5, #0; base address = 0
LDR  R7, [R5, #8];
```

↳ R7 <= data at memory address (R5+8)

↓

Q: why +8?

- ARM uses the store register, STR instruction, to write a data word from

HCL

```
mem[5] = 42;
```

AAC

```
MOV  R1, #0; base add = 0
MOV  R9, #42
STR  R9, [R1, #0x14];
```

↳ value stored at mem. add. (R1+20) = 42

- <u>word-alignment</u> for LDR and STR instructions as required by ARM → this means that the word address is divisible by 4 (there are also LDRB and STRB for "load byte" and "store byte" which do not need to be word-aligned)

→ to access bytes of individual words, two forms of addressing are used:

"big endian" and "little endian" — taken from Jonathan Swift's <u>Gulliver's Travels</u> novel.

> System on module

- an SoM generally has higher performance than SoC (System on a Chip) because it is like a dedicated CPU design but rather than made on a PCB, (Printed circuit board) the components are integrated into a common, small package.

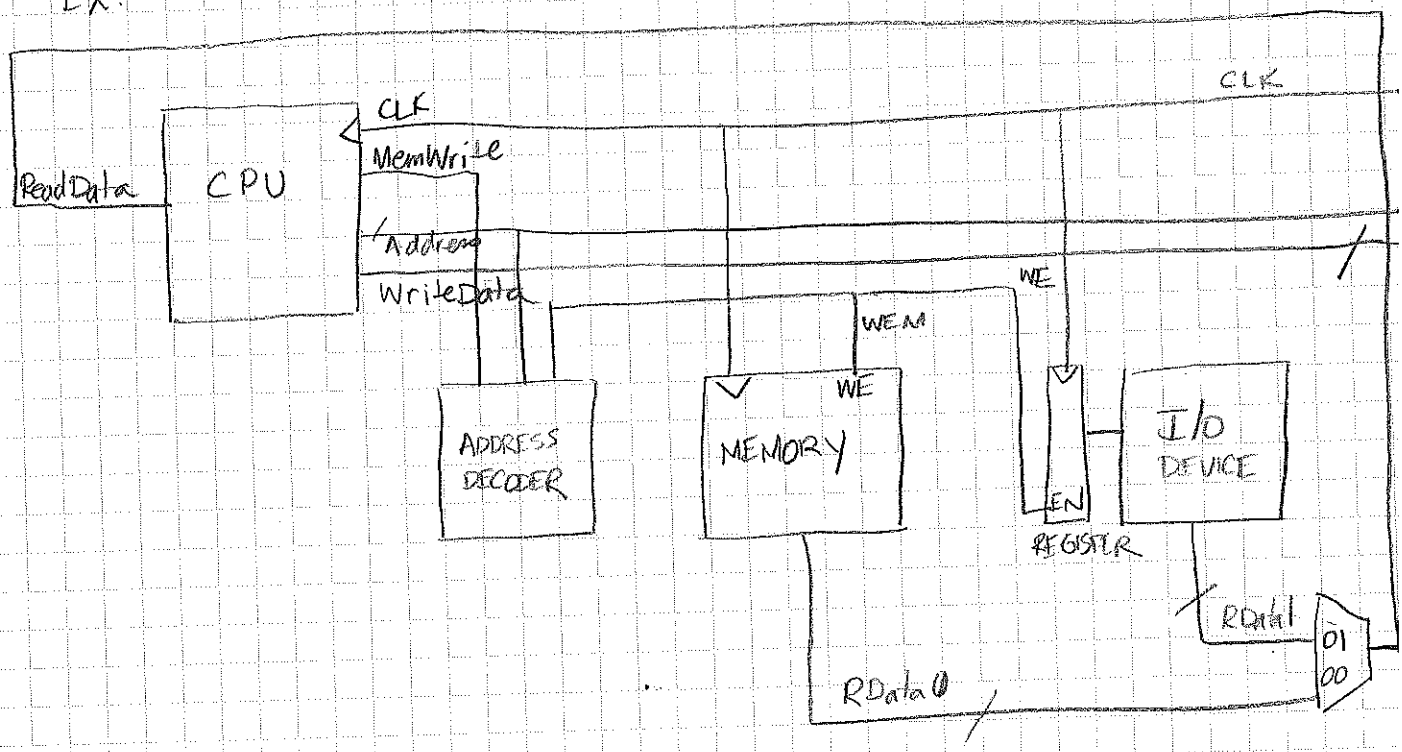
  - can reduce cost in one sense by eliminating the main PCB (printed circuit board)
  - (short) short length interconnects allow for high performance – the benefits of both SoC and dedicated µP designs.

UNIT 2 / UNIT 3

PROGRAMMED I/O

- Memory-mapped I/O
  - a portion of the processor's physical address space is dedicated to I/O devices rather than memory
  - to realize memory-mapped I/O, we make use of an address decoder, registers and a multiplexer.

EX.



- software that allows communication with an I/O device is called a device driver; obtaining or writing device drivers is an important aspect of embedded systems design.
- many kinds of peripherals appear as memory-mapped I/O, from communication ports, displays, A/D and D/A converters, and general-purpose I/

- **interrupt I/O**

  - interrupts allow the development of event-driven systems — when an event occurs, it triggers circuitry to send an interrupt signal to a microcontroller / microprocessor

  - the microcontroller stops whatever it is doing to service the interrupt

  - the interrupt is mapped to a location in the computer's memory which contains the interrupt service routine which is dedicated to meeting the demands of the event.

EX. many x86 (Intel systems) have multiple interrupt lines on a separate chip, the 8259 Programmable Interrupt Controller, usually programmed in edge-triggered mode

Interrupt request table:

IRQ	0	– system timer
IRQ	1	– keyboard controller
IRQ	3/4	– serial port controllers
IRQ	8	– RTC (real-time clock)
IRQ	10/11	– left open for peripherals
IRQ	12	– PS/2 mouse

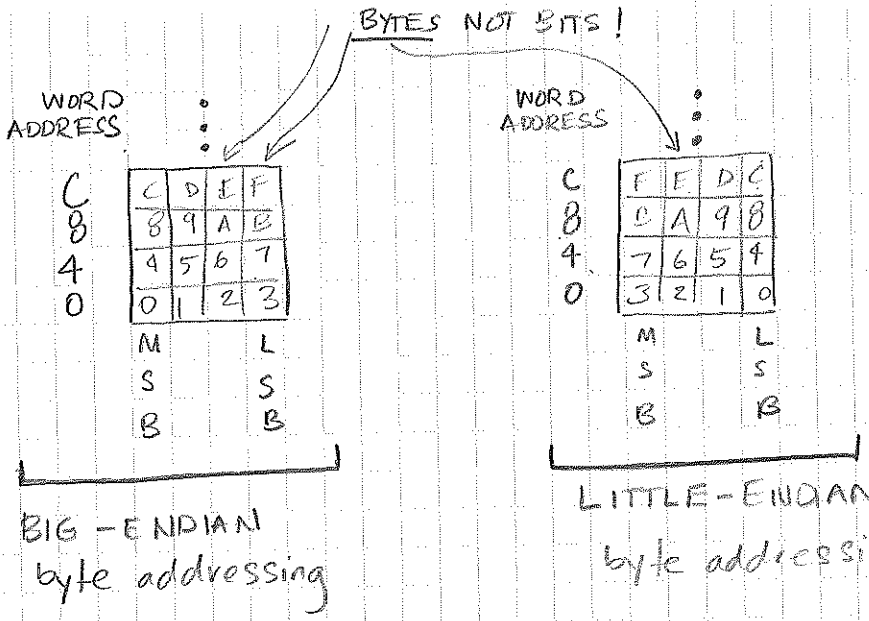
→ the 8259 has been surpassed by the APIC (advanced programmable interrupt controller) on newer Intel systems

→ under Linux, execute

```
$ cat /proc/interrupts
```

to view the interrupt mapping on your PC

- **Direct Memory Access** (DMA)
  - a technology that allows an I/O device to send or receive data to or from the computer memory, without the intervention of the CPU.
  - a process often managed by a DMA controller chip (a DMAC)
  - the idea is to free up the CPU to handle other tasks while an I/O device functions; the DMAC issues an interrupt when data transfers are completed.
  - there may be multiple DMA channels on a computer system, with data transfer rates dependent on the system specifications



EX.

- therefore, second-LSB of the second memory word has the address
  - $4+2=6$ in BE convention
  - $4+1=5$ " LE "

- the second-MSB of the memory word at address C has the address:
  - $C+2=E$ in LE
  - $C+1=D$ in BE

PROGRAMMING

– while not as accessible as high-level programming, there are ways to translate high-level language constructs into assembly language.

DATA PROCESSING INSTRUCTIONS

logical instructions

- AND (bitwise AND)
- ORR (" OR)
- EOR (" XOR)
- BIC (" bit clear)

two operands → MVN (bitwise NOT on the second source, writes result to destination)

three operands: Rdest, Rsource1, Rsource2/I_2
(Rdest: a register; Rsource1: a register; Rsource2/I_2: a register or immediate)

Ex.

```
AND   R3, R1, R2
ORR   R4, R1, R2
EOR   R5, R1, #73

MVN   R7, R2
MVN   R7, #537
```

Shift instructions

```
LSL   (logical shift left)  → LSBs are filled with zeros
LSR   (  "    "   right)   → MSBs  "   "   "   "
ASR   (  "    shift right)  → preserves sign bit
ROR   (rotate right)
```

not ROL since an RA can be performed by ROR using the complementary amount

Ex.

```
LSL   R0, R5, #7
LSR   R1, R5, #17
ASR   R2, R5, #3

LSL   R4, R8, R6
ROR   R5, R8, R6
```

multiply instructions

- 32 bit operands yield a product of upto 64 bit ($2^{32} \times 2^{32} = 2^{64}$)

MUL ← discards the MSBs

produce 64-bit results:

- UMULL (unsigned multiply long) (or $2^{17} \times 2^{16} = 2^{33}$)
- SMULL (signed "

Ex.

```
UMULL R1, R2, R3, R4 ; R3×R4 → R2:R1
```

product is stored in two registers

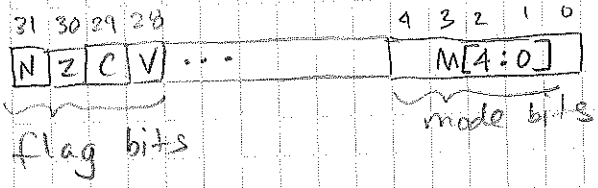
MLA, SMLAL, UMLAL } multiply-accumulate variants, or DSP-like instructions

(product is added to a running sum)

CONDITION FLAGS

→ we use a special register, the current program status register (CPSR)

→ we can affect program flow using this register



eg.
CMP R4, R5
→ADDEQ R1, R2, R3
only executes if Z is set

CONDITION MNEMONICS → appended to regular instructions

cond	mnemonic	name	CondEx
0000	EQ	equal	Z
0001	NE	not equal	$\overline{Z}$
⋮			
0110	VS	overflow/overflow set	V
0111	VC	no overflow	$\overline{V}$
⋮			
1100	GT	signed greater than	$\overline{Z}(\overline{N \oplus V})$
1101	LE	" less than or equal	Z OR $(N \oplus V)$

→ see text. Table 6.3 for full list, in Harris & Harris

BRANCHING

→ branch instructions:

```
B     THERE   ; go to THERE
BEQ   THERE1  ; branch if equal
BGT   THERE2  ;  "   "  (signed) greater than
BVS   THERE3  ;  "  if overflow
```

CONDITIONAL STATEMENTS

- if/else

HIGH LEVEL CODE:

```
if (X == Y)
    f = i + 1;
else
    f = f - i;
```

ARM ASSEMBLY CODE:

```
; R0 = X , R1 = Y , R2 = f, R3 = i
        CMP  R0, R1   ; X == Y ?
        BNE  L1       ; if not equal, skip if block
        ADD  R2, R3, #1 ; if block: f = i + 1
        B    L2       ; skip else block
L1      SUB  R2, R2, R3 ; else block: f = f - i
L2
```

- switch/case

HIGH-LEVEL CODE

```
switch (X) {
    case 1:  Y = 20; break;
    case 2:  Y = 50; break;
    case 3:  Y = 100; break;
    default: Y = 0;
}
```

ARM ASSEMBLY CODE

```
; R0 = X, R1 = Y
        CMP  R0, #1     ; is case X = 1 ?
        MOVEQ R1, #20   ; Y = 20
        BEQ  DONE       ; break
        CMP  R0, #2     ; is case X = 2 ?
        MOVEQ R1, #50   ; Y = 50
        BEQ  DONE       ; break
```

make assignment #2 due next week

Homework: 6.1 – 6.5

... con't'd

```
        CMP   R0, #3      ; is case X=3?
        MOVEQ R1, #100    ; Y=100
        BEQ   DONE        ; break
        MOV   R1, #0      ; default case, Y=0
DONE
```

Loops

- while loops

HLC: → finds power of 2 that yields 128

```
int pow = 1;
int x = 0;
while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

AAL:

```
; R0 = pow, R1 = x
    MOV  R0, #1   ; pow = 1
    MOV  R1, #0   ; x = 0
WHILE
    CMP  R0, #128 ; pow == 128?
    BEQ  DONE     ; if so, exit
    LSL  R0, R0, #1 ; pow = pow * 2
    ADD  R1, R1, #1 ; x = x + 1
    B    WHILE    ; unconditional branch to continue loop
DONE
```

- for loops

HLC: adds the numbers from 0 to 9

```
int i;
int sum = 0;
for (i=0; i<10; i=i+1) {
    sum = sum + i;
}
```

AAC:

```
; R0 = i, R1 = sum
MOV  R1, #0   ; sum = 0
MOV  R0, #0   ; i = 0      loop initialization

FOR
    CMP  R0, #10   ; i < 10 ?
    BGE  DONE      ; check OPPOSITE CONDITION, i >= 10, to exit
    ADD  R1, R1, R0 ; sum = sum + i
    ADD  R0, R0, #1 ; increment index
    B FOR          ; repeat (unconditional branch)

DONE
```

MEMORY

- one of the simplest data structures is a contiguous set of memory words allocated at consecutive addresses...

eg. consider an array scores[ ] with 200 elements

memory address	MAIN MEMORY
1400031C	scores[199]
14000318	scores[198]
⋮	⋮
14000008	scores[2]
14000004	scores[1]
14000000	scores[0]

HLC:

```
int i;
int scores [200];
...
for (i=0; i<200; i=i+1)
    scores [i] = scores [i] + 10;
```

AAC:

```
; R0 = array base address, R1 = i
; initialization code ...
    MOV   R0, #0x14000000 ; R0 = base address
    MOV   R1, #0 ; i = 0
LOOP
    CMP   R1, #200 ; i < 200 ?
    BGE   L3       ; if i ≥ 200, exit
    LSL   R2, R1, #2 ; R2 = i * 4 (32-bit addressing)
    LDR   R3, [R0, R2] ; R3 = scores [i]
    ADD   R3, R3, #10 ; R3 = scores [i] + 10,
    STR   R3, [R0, R2] ; scores [i] = scores [i] + 10
    ADD   R1, R1, #1   ; i = i + 1
    B     LOOP         ; repeat
L3
    ; program comes here when loop is complete
```

ARM INDEXING MODES

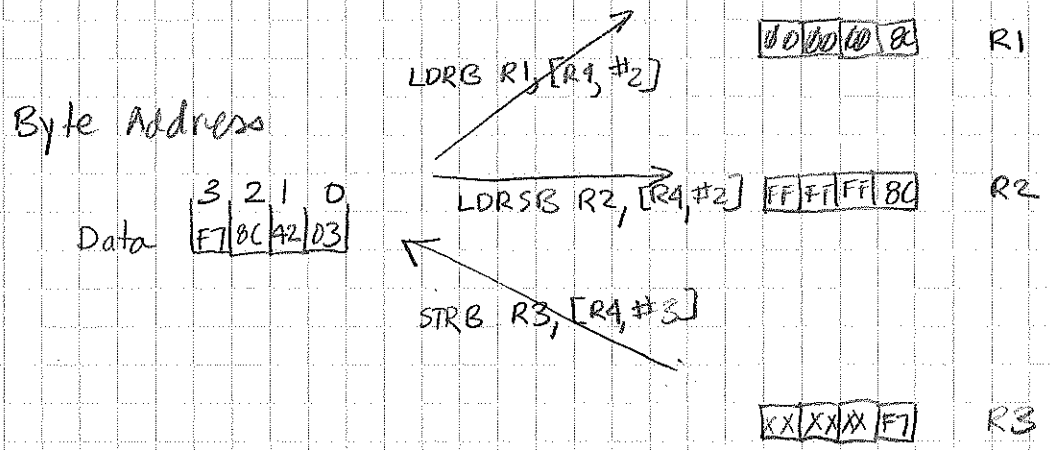
mode	ARM ASSEMBLY	ADDRESS	BASE REGISTER (R1)
Offset	LDR R0, [R1, R2]	R1 + R2	unchanged
Pre-Index	LDR R0, [R1, R2]!	R1 + R2	R1 = R1 + R2
Post-Index	LDR R0, [R1], R2	R1	R1 = R1 + R2

- see example in text → it's possible to simplify program further
- read about ASCII codes

Byte-level addressing

LDRB
LDRSB
STRB

↳ fill the entire 32-bit register with zeros or sign, respectively, placing contents in the LS Byte



→ see example 6.2 in text for character arrays - characters use byte-level addressing

FUNCTION CALLS

- ARM architecture uses the branch and link instruction (BL) to call a function and moves the link register to the program counter (PC) (MOV PC, LR) to return from a function

HLC:

```
int main() {
    simple();
    ...
}
void simple() {
    return; // void means no return value
}
```

AAC:

```
MAIN  ...          ; main code begins here
      :

      BL SIMPLE    ; call the simple() function

SIMPLE

      MOV PC, LR   ; return
```

- recall LR ↔ R14
          PC ↔ R15

- more complex example:

HLC:

```
int main () {
    int y;
      :
    y = diff of sums(2,3,4,5);
      :
}
int diff of sums (int f, int g, int h, int i) {
    int result
    result = (f+g) - (h+i);
    return result;
}
```

```
AAC:
; R4 = y
MAIN
        :
        MOV R0, #2   ; argument 0 = 2
        MOV R1, #3   ;     "    1 = 3
        MOV R2, #4   ;     "    2 = 4
        MOV R3, #5   ;     "    3 = 5
        BL  DIFFOFSUMS ; call function
        MOV R4, R0   ; y = returned value
        :
; R4 = result
DIFFOFSUMS
        ADD R8, R0, R1 ; R8 = f+g
        ADD R9, R2, R3 ; R9 = h+i
        SUB R4, R8, R9 ; result = (f+g)-(h+i)
        MOV R0, R4   ; put return value in R0
        MOV PC, LR   ; return to caller
```

VHDL TUTORIAL — largely taken from Jan Van der Spiegel's on-line VHDL tutorial...

1) Introduction

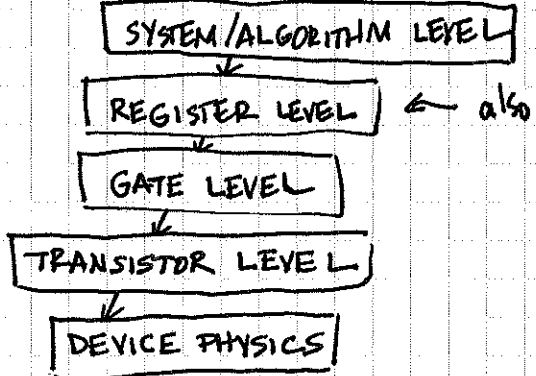
- developed in mid-1980s to facilitate the design of very high speed digital integrated circuits
- created jointly by the IEEE and the U.S. DoD

Q: how do you feel about much of technology's roots in militarism?

- other widely used Hardware Description Language is Verilog (perhaps more widely used) — we'll also cover some Verilog
- VHDL stands for Very high speed IC Hardware Description Language
- although HDLs bear some resemblance to most conventional programming languages, there are differences, namely:
  - → HDLs are inherently parallel
  - → HDLs mimic the behaviour of a physical digital system
  - → allows incorporation of timing specs and the ability to describe a system as an interconnection of components (like Simulink/Matlab if you have the experience with that)

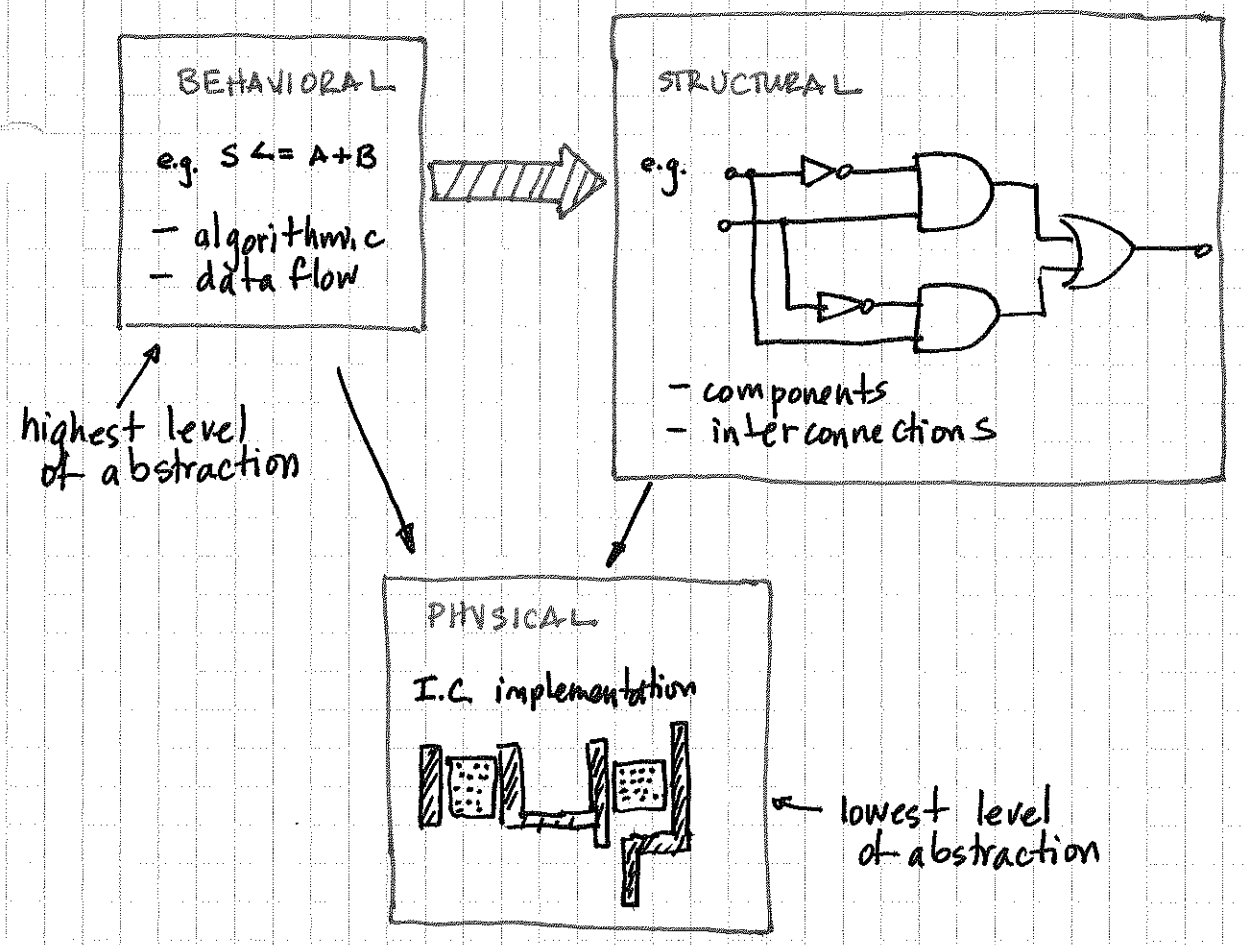
2) Representation and Abstraction

- a recurring theme in digital design: levels of abstraction
  - → allows us to manage complexity



← also called "Register-Transfer Level"

↓ describes data flow/timing characteristics using register-level component descriptions



→ with VHDL, both behavioural and structural descriptions of a system are possible

BEHAVIORAL
- DATA FLOW → concurrent statements (parallel execution)
- ALGORITHMIC → sequential processes/statements

3) VHDL File Structure

BASIC BUILDING BLOCK → VHDL ENTITY

- declaration:

```
entity NAME_OF_ENTITY is [ generic (generic_declarations); ]
    port ( signal_names_1 : mode type;
           signal_names_2 : mode type;
                  :
           signal_names_K : mode type );
end [ NAME_OF_ENTITY ] ;
```

user-selected

optional

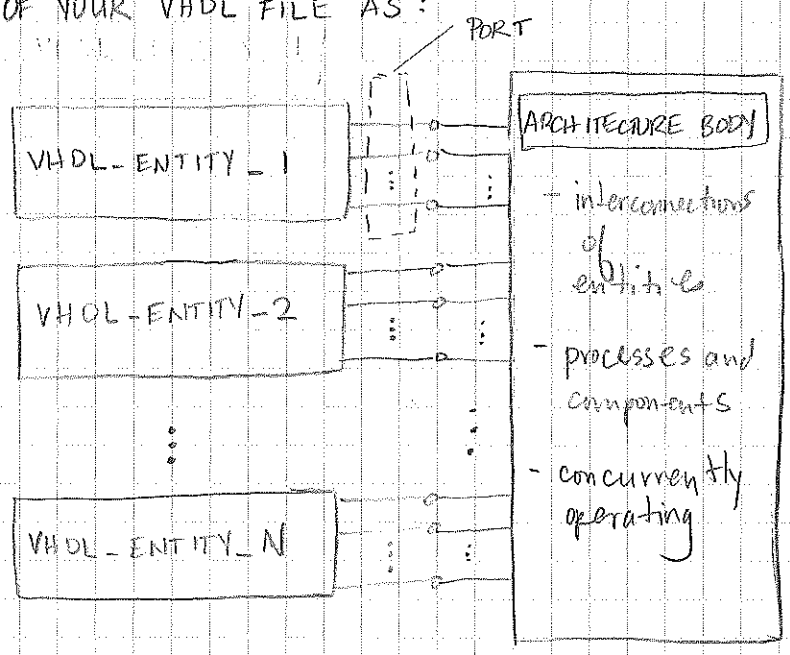
a comma-separated list of one or more user-selected identifiers specifying external signals

mode: one of: in, out, buffer, inout

type: one of several pre-defined signal types — see next page

optional

THINK OF YOUR VHDL FILE AS:



NOTE: • VHDL:

- has reserved keywords that cannot be used as names for signals or entities;
  → keywords and user-defined identifiers are case insensitive
- employs comment lines by -- ;
- ignores line breaks and extra spaces
- is a strongly typed language which employs type declarations of every signal, constant or variable

• regarding mode keywords:

in - indicates that the signal is an input
out - indicates that the signal is an output which can ~~by~~ be read only by other entities

buffer - an output which can be read by the entity itself

inout - a signal which can be either an input or an output

• regarding signal types:

bit - can have the values 0 or 1
bit_vector - a vector of bit values
std_logic
std_ulogic
std_logic_vector
std_ulogic_vector
} can have 9 values corresponding to signal strength ⟹ preferred over corresponding bit and bit_vector types

boolean - can have the value, TRUE and FALSE
integer - can have a range of integer values
real - can have a range of real values
character - any printing character
time - to indicate time

- generic declarations are optional
  - → provide local constants used for timing and sizing (bus widths) for the entity
  - syntax:

```
generic (
    constant_name_1 : type [:= value_1];
    constant_name_2 : type [:= value_2];
                    :
    constant_name_K : type [:= value_K] );
```

Some Examples:

```
-- buzzer example
entity BUZZER is
    port (DOOR, IGNITION, SBELT : in std_logic;
          WARNING : out std_logic);
end BUZZER;
```

```
-- four-to-one multiplexor for 8-bit words
entity mux4_to_1 is
    port ( I0, I1, I2, I3 : in std_logic_vector (7 downto 0);
           SEL : in std_logic_vector (1 downto 0);
           OUTI : out std_logic_vector (7 downto 0));
end mux4_to_1;
```

```
-- D-flip-flop with set and reset
entity dff_sr is
    port (D, CLK, S, R : in std_logic;
          Q, Qnot : out std_logic);
end dff
```

Architecture Body

- specifies how the circuit operates and how it is implemented
- syntax:

```
architecture architecture_name of name_of_entity is
-- declarations (components, signals, constants, etc.)
    :
begin
-- statements
    :
end architecture_name;
```

- examples:

```
entity AND2 is
    port (in1, in2 : in std_logic;
          out1 : out std_logic);
end AND2;

architecture behavioural_2 of AND2 is
begin
    out1 <= in1 and in2;
end behavioural_2;
```

- for BUZZER example above:

```
architecture behavioural_buzzer of BUZZER is
begin
    WARNING <= (not DOOR and IGNITION) or
               (not SBELT and IGNITION);
end behavioural_buzzer;
```

→ assignment operator

another example:

```
entity AND2 is
    port (in1, in2 : in std_logic;
          out1 : out std_logic);
end AND2;

architecture behavioural_2 of AND2 is
begin
    out1 <= in1 and in2;
end behavioural_2;

entity XNOR2 is
    port (A, B : in std_logic;
          Z : out std_logic);
end XNOR2;

architecture behavioural_xnor of XNOR2 is
    -- signal declaration (of internal signals X, Y)
    signal X, Y : std_logic;
begin
    X <= A and B;
    Y <= (not A) and (not B);
    Z <= X or Y;
end behavioural_xnor;
```

logic operators: and, or, nand, nor, xor, xnor, not

Concurrency

The statements in the Architecture Body

```
begin
    X <= A and B;
    Y <= (not A) and (not B);
    Z <= X or Y;
end behavioral_xnor2;
```

are concurrent statements ⇒ they are executed when one or more of the signals on the RHS change value (i.e. an "event" occurs on one of the signals).

* There may be (must be?) a propagation delay associated with this change — cannot have algebraic loops!

Structural Description

Here is a structural version of BUZZER:

```
architecture structural_buzz of BUZZER is
  -- declarations
  component AND2
      port (in1, in2 : in std_logic;
            out1 : out std_logic);
  end component;
  component OR2
      port (in1, in2 : in std_logic;
            out1 : out std_logic);
  end component;
  component NOT1
      port (in1 : in std_logic;
            out1 : out std_logic);
  end component;
  -- signals for interconnect
  signal DOOR_NOT, SBELT_NOT, B1, B2 : std_logic
begin
    U0 : NOT1 port map (DOOR, DOOR_NOT);
    U1 : NOT1 port map (SBELT, SBELT_NOT);
    U2 : AND2 port map (IGNITION, DOOR_NOT, B1);
    U3 : AND2 port map (IGNITION, SBELT_NOT, B2);
    U4 : OR2  port map (B1, B2, WARNING);
end structural;
```