

# ENGSCI YEAR 3 FALL 2022 NOTES

---

BRIAN CHEN

*Division of Engineering Science*

*University of Toronto*

*<https://chenbrian.ca>*

*[brianchen.chen@mail.utoronto.ca](mailto:brianchen.chen@mail.utoronto.ca)*

---

## Contents

<b>I ECE349: Introduction to Energy Systems</b>	<b>1</b>
<b>1 Admin stuff</b>	<b>1</b>
1.1 Lecture 1	1
1.1.1 Mark breakdown	1
<b>2 AC Steady State Analysis</b>	<b>1</b>
2.1 Lecture 2	1
2.1.1 TODO	1
2.2 Lecture 3	3
<b>3 AC Power</b>	<b>4</b>
3.1 Lecture 4	5
3.1.1 Root Mean Squared (RMS) Values	6
3.2 Lecture 5: Multi-Phase AC	7
3.3 Lecture 6: Y and Delta connections	10
3.4 Lecture 7: DC-DC conversion	12
3.5 Lecture 8: Average output voltage	15
3.5.1 Filtering	17
3.6 Lecture 9	18
3.6.1 Inductor VoH-Seconds balance	20
3.6.2 Capacitor charge balance	20
3.7 Lecture 10	21
3.7.1 Design Process	23
3.8 Lecture 11	24
3.9 Lecture 12	26
<b>II ECE352: Computer Organization</b>	<b>26</b>
<b>4 Admin stuff</b>	<b>26</b>
4.1 Lecture 1	26
4.1.1 Mark breakdown	26
<b>5 Preliminary</b>	<b>26</b>
5.1 Lecture 2: Using binary quantities to represent other things	26
5.1.1 Floating Point Numbers	27

<b>6</b>	<b>Memory and NIOS II</b>	<b>28</b>
6.1	Lecture 3: Behavioural Model of Memory	28
6.1.1	Memory	29
6.1.2	Physical Interface	30
6.2	Lecture 4: NIOS II Programming Model	31
6.2.1	Adding Two Numbers	32
6.2.2	Adding two numbers using memory	34
<b>7</b>	<b>Assembly Basics</b>	<b>36</b>
7.1	Lecture 5: Simple Control Flow	36
7.2	Lecture 6, 7: For loops and arrays	38
7.3	Lecture 8: Subroutines	40
7.4	Lecture 9	46
7.5	Lecture 10: Recursive Subroutines	48
7.6	Lecture 11: Structs and recursive structures	49
<b>8</b>	<b>GPIO</b>	<b>54</b>
8.1	Lecture 12: Devices	54
8.1.1	The PIT implementation	58
8.1.2	Memory	59
8.2	Lecture 12: UART	61
8.3	Lecture 13: Interrupts & UART	65
<b>III</b>	<b>ECE355: Signal Analysis and Communication</b>	<b>66</b>
<b>9</b>	<b>Admin and Preliminary</b>	<b>66</b>
9.1	Lecture 1	66
9.1.1	Mark Breakdown	67
<b>10</b>	<b>Transformations</b>	<b>67</b>
10.1	Lecture 2	67
10.2	Lecture 3	68
10.2.1	General Continuous Complex Exponential Signals	69
<b>11</b>	<b>Basic Signals</b>	<b>69</b>
11.1	Lecture 4: Step and Impulse Functions	69
11.2	Lecture 5	71
11.2.1	Types of systems	72
11.2.2	System properties	73
<b>12</b>	<b>LTI systems</b>	<b>74</b>
12.1	Lecture 6: Discrete LTI systems	74
12.2	Lecture 7: Continuous LTI systems	76
12.3	Lecture 8	77

<b>IV ECE358: Foundations of Computing</b>	<b>77</b>
<b>13 Admin and Preliminary</b>	<b>77</b>
13.1 Lecture 1	77
13.1.1 Mark Breakdown	77
<b>14 Complexities</b>	<b>77</b>
14.1 Lecture 2	77
14.2 Lecture 3: Logs & Sums	80
14.2.1 Functional Iteration	80
14.3 Lecture 4: Induction & Contradiction	81
14.3.1 Induction	81
14.3.2 Contradiction	82
14.4 Lecture 5: recurrences	83
14.4.1 Recurrence Trees	84
14.4.2 Substitution	85
14.4.3 Master Theorem	85
14.5 Lecture 6	86
14.5.1 Graphs	86
14.5.2 Trees	88
14.6 Lecture 7, 8: Probability and Counting	88
14.7 Lecture 9: Heapsort	89
14.8 Lecture 10, 11: Quicksort	90
14.9 Lecture 12: lower bound, counting and radix sort	90
14.10 Lecture 13: Select & BST	90
<b>V ECE360: Electronics</b>	<b>90</b>
<b>15 Admin and Preliminary</b>	<b>90</b>
15.1 Lecture 1	91
15.1.1 Mark Breakdown	91
15.1.2 Diodes	91
<b>16 Diodes</b>	<b>92</b>
16.1 Lecture 2	92
16.2 Lecture 3	94
<b>17 Lecture 4 &amp; 5: Forward conducting diodes</b>	<b>96</b>
<b>18 Small-Signal Model</b>	<b>98</b>
<b>19 Lecture 6: Small signal model, cont'd</b>	<b>99</b>
19.0.1 Deriving small-signal resistance	100
<b>20 Lecture 7</b>	<b>102</b>

<b>21 Rectifiers</b>	<b>103</b>
21.1 Lecture 8	105
21.2 Lecture 9	108
21.3 Lecture 10	111
21.4 Lecture 11: Introduction to transistors and MOSFETs	114
21.5 Lecture 12: MOSFET Operating Regions	116
21.6 Lecture 13: MOSFET current-voltage characteristics	119
<b>VI MAT389: Complex Analysis</b>	<b>119</b>
<b>22 Complex Numbers</b>	<b>119</b>
22.1 Lecture 1	119
22.2 Lecture 2	121
22.2.1 Functions on complex planes	122
22.2.2 Exponential Functions	123
22.3 Lecture 3: Exponent and Logarithm	125
22.3.1 Exponential	125
22.3.2 Logarithm	125
22.3.3 Powers	127
22.4 Lecture 4	127
22.4.1 Properties of complex derivative	129
22.5 Power series	133
22.6 Lecture 5	134
<b>VII ECE444: Software Engineering</b>	<b>134</b>
<b>23 Preliminary</b>	<b>134</b>
23.1 Lecture 1, 2	135
<b>24 Project Management</b>	<b>135</b>
24.1 Lecture 3	135
24.1.1 Agile	136
24.2 I dropped this course	136
<b>VIII ESC301: Seminar</b>	<b>136</b>

---

# ECE349: *Introduction to Energy Systems*

SECTION 1

Taught by Prof. P. Lehn

## Admin stuff

---

SUBSECTION 1.1

### Lecture 1

---

First lecture was logistical info + a speil about how power systems are one of the great modern wonders. Course will cover sinusoidal AC power systems (1, 3 phase), power systems (dc-dc, dc-ac conversion), and magnetic systems (transformers, actuators, and synchronous machines)

#### 1.1.1 Mark breakdown

- 50 % Final
- 25 % Midterm
- 5 % Quiz
- 15 % Labs
- 5 % Assignments

SECTION 2

## AC Steady State Analysis

---

SUBSECTION 2.1

### Lecture 2

---

#### 2.1.1 TODO

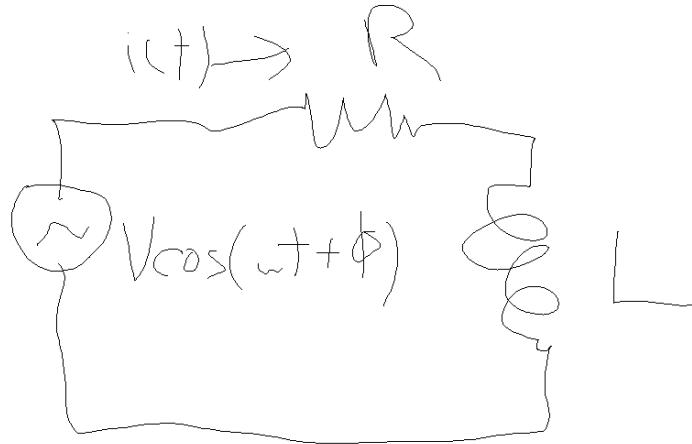
- Review Thomas 669-600

What we have learnt prior for differential equations enables us to arrive at analytical solutions to linear stable AC systems with phasors. A homogeneous and particular solution will be produced. If there's a stable homogeneous solution,  $\rightarrow 0$  as  $t \rightarrow \infty$ . The full solution would be the addition of the two via super position.

We generally use this approach to solve circuits since it's an efficient way to solve circuits and make them into essentially DC circuits.

Recall, for a general phasor  $\hat{P}$

- $\frac{d\hat{P}}{dt} = jw\hat{P}$
- $\int \hat{P} = \frac{1}{jw} \hat{P}$



$$Ri + L \frac{di}{dt} = V \cos(\omega t + \phi) \quad (2.1)$$

But this is a pain to solve. It can be made simpler by applying phasors

$$V \cos(\omega t + \phi) = \operatorname{Re}\{V e^{j(\omega t + \phi)}\} \quad (2.2)$$

Take the real part of  $\hat{I}$ :

$$R\hat{I} + L \frac{d\hat{I}}{dt} = V e^{j(\omega t + \phi)} \quad (2.3)$$

And therefore by inspection the solution is of format  $\hat{I} e^{j\omega t}$ , where  $\hat{I}$  is a phasor. Noting that  $\hat{I}$  contains only amplitude and phase,

$$\begin{aligned} R\hat{I} e^{j\omega t} + L \frac{d}{dt}(\hat{I} e^{j\omega t}) &= V e^{j\omega t + \phi} \\ R\hat{I} + L\hat{I}j\omega &= V e^{j\phi} \end{aligned} \quad (2.4)$$

And now reconstructing:

$$\begin{aligned} \hat{I} &= \frac{V}{\sqrt{R^2(\omega L)^2}} e^{j(\omega t + \phi - \tan^{-1} \frac{\omega L}{R})} \\ i(t) &= \operatorname{Re} \left\{ \hat{I} \right\} \end{aligned} \quad (2.5)$$

And therefore

$$\hat{I} = \frac{V}{\sqrt{R^2(\omega L)^2}} \cos \left( \omega t + \phi - \tan^{-1} \left( \frac{\omega L}{R} \right) \right) \quad (2.6)$$

The steps to solving a phasor problem are:

Notation:  $X e^{j\phi} \leftrightarrow X | \phi$

- Define phasor:  $V \cos(\omega t + \phi) \leftrightarrow V | \phi$
- Map  $L, C$  into phasor domain; find impedances
  - $v = L \frac{di}{dt} \leftrightarrow \hat{V} = j\omega L \hat{I}$

$$-i = C \frac{dv}{dt} \leftrightarrow \hat{V} = \frac{1}{j\omega C} \hat{I}$$

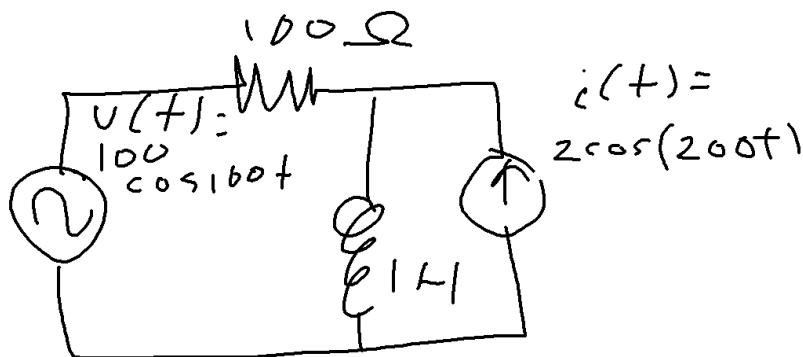
- Do mesh analysis to find  $\hat{I}$ ;  $\hat{I} = \frac{\hat{V}}{\sum \text{impedances}}$

- Reconstruct  $i(t)$  from  $\hat{I}$

SUBSECTION 2.2

## Lecture 3

Phasors allow us to solve circuits with multiple sources of differing frequencies.



To find the current  $i(t)$  over the inductor we can find it's response due to the voltage and current sources and then apply superposition.

- $I_1 = \frac{100|0}{100+j100} = 0.707|-45^\circ \rightarrow i_1(t) = 0.707 \cos(100t - 45^\circ)$
- $I_2 = \frac{100}{100+j200} 2|0 = 0.894|-65^\circ \rightarrow i_2(t) = 0.894 \cos(200t - 63^\circ)$
- $i(t) = i_1(t) + i_2(t) = 0.707 \cos(100t - 45^\circ) + 0.894 \cos(200t - 63^\circ)$

Non-sinusoidal stimulus may be solved by decomposing the signal with Fourier transforms. For example, square waves:

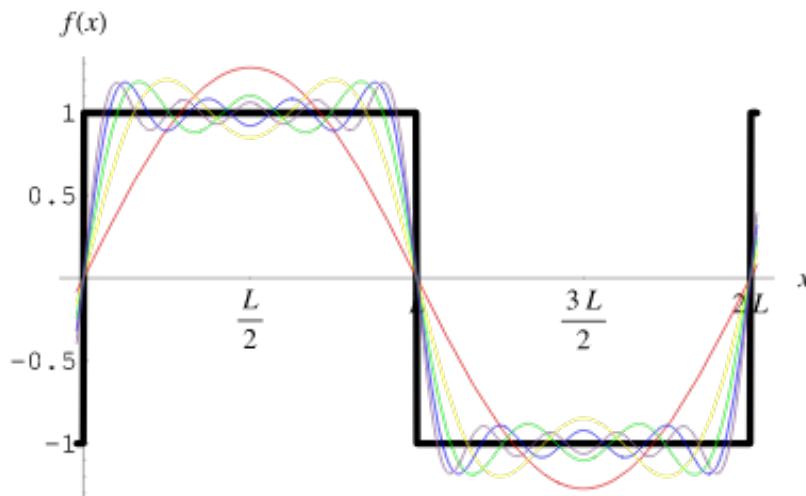


Figure 1. square waves with Fourier series superimposed

The general form of a Fourier transform is given as:

$$v_{equiv}(t) = a_o + \sum_{n=1}^{\infty} a_k \cos(nw_o t) + b_k \cos(nw_o t) \quad (2.7)$$

Where:

$$\begin{aligned} a_o &= \frac{1}{T} \int_0^T v(t) dt \\ a_k &= \frac{2}{T} \int_0^T v(t) \cos(nw_o t) dt \\ b_k &= \frac{2}{T} \int_0^T v(t) \sin(nw_o t) dt \end{aligned} \quad (2.8)$$

Armed with Fourier series and superposition we may now model a non-sinusoidal signal as a superposition of an infinite sum of sources. About half the work can be cut in half by recognizing that  $\sin$  lags  $\cos$  by  $90^\circ$ , so

$$\begin{aligned} a_o &= \frac{1}{T} \int_0^T v(t) dt \\ a_k &= \frac{2}{T} \int_0^T v(t) \cos(nw_o t) dt \\ b_k &= \frac{2}{T} \int_0^T v(t) \cos(nw_o t - 90^\circ) dt \end{aligned} \quad (2.9)$$

### SECTION 3

## AC Power

---

**Definition 1** **Instantaneous Power:**  $p(t) = v(t) \times i(t) [W, \frac{J}{s}]$

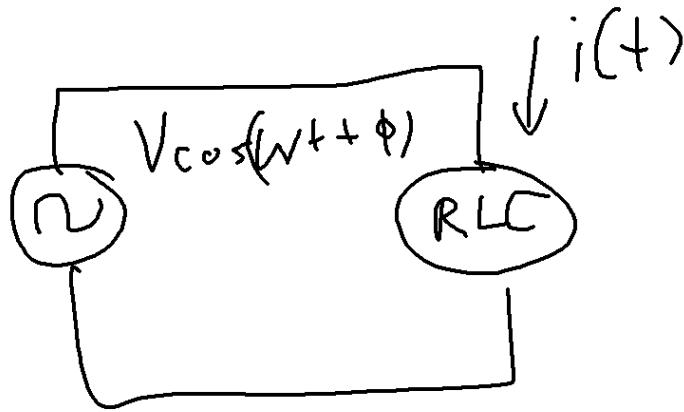
*Example* For a circuit with a voltage source,  $v(t) = V \cos(\omega t)$  and a resistor  $\Omega$ ,  $i(t) = I \cos(\omega t)$ ,  

$$p(t) = VI \cos^2(\omega t) = \frac{VI}{2}(1 + \cos(2\omega t))$$

**Definition 2** **Average Power over Cycle:**  $P(t) = \frac{1}{T} \int_0^T p(t) dt = \frac{VI}{2}$

If we were to plot the instantaneous power we see that due to the sinusoidal response there are times where 0 power is supplied. This will always be true for a single phase power supply; real-world supplies always have multiple phases; this is why computer PSUs always contain a ton of capacitors.

**Definition 3** **Reactive Power:**  $Q$



If  $\phi_i = 0$  and taking  $\phi = \phi_v$ ,

$$\phi = \phi_v - \phi_i$$

$$\begin{aligned}
 p(t) &= V \cos(\omega t + \phi) * I \cos(\omega t) \\
 &= \frac{VI}{2} \cos(\phi) + \cos(2\omega t + \phi) \\
 &= \underbrace{\frac{VI}{2} \cos(\phi)(1 + \cos(2\omega t))}_{\text{real power e.g. heat}} - \underbrace{\frac{VI}{2} (\sin \phi \sin 2\omega t)}_{\text{stored and released back to source}}
 \end{aligned} \tag{3.1}$$

Taking the average power of the reactive power we get

$$P_{avg} = \frac{VI}{2} \cos \phi \tag{3.2}$$

Another quantity, reactive power, can be defined with regards to the energy sloshing back and forth:

$$Q = \frac{VI}{2} \sin \phi \tag{3.3}$$

#### SUBSECTION 3.1

## Lecture 4

### Definition 4

#### Displacement factor

$$DF \equiv \cos \phi$$

Where  $\phi$  is the angle measured from the  $\hat{V}$  to the  $\hat{I}$  phasors. A  $DF = 1$  means the system is transferring the most power possible.

- Lagging DF:  $\phi$  is -'ve
- Leading DF:  $\phi$  is +'ve

We can also write  $P, Q$  from phasors.

$$\begin{aligned}
P &= \frac{\hat{V}\hat{I}}{2} \cos(\phi_v - \phi_i) \\
&= \frac{1}{2} \hat{V}\hat{I} \operatorname{Re}\{e^{j\phi_v - \phi_i}\} \\
&= \frac{1}{2} \operatorname{Re}\{V e^{j\phi_v} \cdot I e^{-j\phi_i}\} \\
&= \frac{1}{2} \operatorname{Re}\{\hat{V}\hat{I}^*\}
\end{aligned} \tag{3.4}$$

$$\begin{aligned}
Q &= \frac{\hat{V}\hat{I}}{2} \sin(\phi_v - \phi_i) \\
&\vdots \\
&= \frac{1}{2} \operatorname{Im}\{\hat{V}\hat{I}^*\}
\end{aligned} \tag{3.5}$$

The expressions for  $Q$  and  $P$  are basically the same so we define a new quantity, complex power, which incorporates both the imaginary and real components.

Reference: Thomas 16.1, 16.2, 16.3

Definition 5

### Complex Power

$$S = \frac{1}{2} \hat{V}\hat{I}^* = P + jQ \tag{3.6}$$

#### 3.1.1 Root Mean Squared (RMS) Values

A RMS value measures the average power of a periodic signal. Consider a circuit with an AC voltage source with  $v(t)$  flowing through a resistor.

The power is:

$$p(t) = v(t)i(t) = v(t) \frac{v(t)}{R} = \frac{1}{R} v(t)^2 \tag{3.7}$$

The average power is therefore

$$P = \frac{1}{R} \int_0^T v(t)^2 dt \tag{3.8}$$

Evaluating this becomes easier by defining an useful quantity, RMS voltage

$$v_{rms} = \sqrt{\frac{1}{T} \int_0^T v(t)^2 dt} \tag{3.9}$$

And using  $v_{rms}$  we get a nice expression for average power,

$$P = \frac{1}{R} v_{rms}^2 \tag{3.10}$$

More generally speaking we can define RMS values of sinusoidal signals

Definition 6

$$v_{rms} = \sqrt{\frac{1}{2\pi} \int_0^{2\pi} (\hat{V} \cos wt)^2 dt} = \frac{1}{\sqrt{2}} \hat{V} \tag{3.11}$$

Plugging this into the expressions for complex power:

$$S = \underline{V} \cdot \underline{I}^* \quad (3.12)$$

Where  $\underline{V}$ ,  $\underline{I}^*$  are RMS phasors at a given common frequency.

And more generally yet, the RMS values of non-sinusoidal signals can be found with help of a Fourier expansion

**Definition 7**

Let  $v(t) = \hat{v}_0 + \hat{v}_1 \cos(wt + \phi_1) + \dots$

Then,

$$v_{rms} = \sqrt{\hat{v}^2 + \frac{\hat{v}_1^2}{\sqrt{2}} + \frac{\hat{v}_2^2}{\sqrt{2}} \dots} = \sqrt{v_0^2 + \sum_{n=1}^{\infty} \left( \frac{V_m^2}{2} \right)} \quad (3.13)$$

And if  $V_m$  is the *rms* value,

$$v_{rms} = \sqrt{v_0^2 + \sum_{n=1}^{\infty} V_m^2} \quad (3.14)$$

One thing to watch out for is that we could have a system with high voltage but near-zero current transferring little to no power. To account for this we look at the power factor 'PF'

**Definition 8**

**Power Factor**

$$PF \equiv \frac{\text{average power}}{\text{rms voltage} \cdot \text{rms current}} \quad (3.15)$$

For a sinusoidal  $V, I$ :

$$\begin{aligned} PF &= \frac{\frac{1}{2} \hat{V} \hat{I} \cos \phi}{\frac{\hat{V}}{\sqrt{2}} \cdot \frac{\hat{I}}{\sqrt{2}}} \\ &= \cos \phi \end{aligned} \quad (3.16)$$

If signals are not harmonics  $PF = DF$ . This can be a source of confusion.

For non-sinusoidal systems this becomes more difficult because, unlike pure signals, they may contain harmonics. In these systems either  $V, I$ , or both may contain harmonics. Generally in household power  $V$  is clean but  $I$  contains harmonics.

The effect of harmonics in currents is that  $I_{\text{harmonics}}$  causes a higher  $I_{rms}$ ; there is more current but no higher power! This reduces  $PF$  and causes  $PF < DF$ .

To summarize,

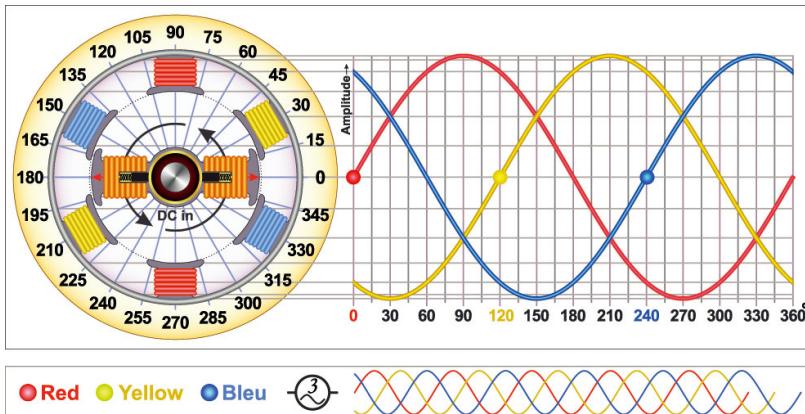
- In ideal systems,  $PF = DF$  if all  $V$  and  $I$  are at one frequency.
- $PF = 1$  means no energy sloshing between load and source.

Harmonics are bad because they reduce the usefulness of the system. There are very tight standards for how many harmonics one is allowed to inject into the system via generators or loads. See: textbook 16.6

SUBSECTION 3.2

## Lecture 5: Multi-Phase AC

AC power is generated by spinning a magnet between some coils. Some pixies get excited and by some Maxwell's equations and EMF and ECE259 we get a voltage induced in the coils.



Current from a generator with a single pair of coils is *single phase*. Most generators, like the one in the picture, have three pairs of coils and therefore generate three phases. For a typical three-phase setup with coils arranged at  $0^\circ, 120^\circ, 240^\circ$ , the voltages can be found with a little bit of trigonometry.

$$\begin{aligned}
 v_a(t) &= \sqrt{2}V \cos \omega t \rightarrow \underline{V_a} = V|0 \\
 v_b(t) &= \sqrt{2}V \cos(\omega t - 120^\circ) \rightarrow \underline{V_b} = V|-120^\circ \\
 v_c(t) &= \sqrt{2}V \cos(\omega t - 240^\circ) \rightarrow \underline{V_c} = V|240^\circ
 \end{aligned} \tag{3.17}$$

And similar expressions may be derived for single-phase and two-phase power.

The reason why three-phase power is typically used has to do with efficiency of power transfer relative to the amount of wires and copper needed. Whereas single-phase power requires two wires to carry power and two-phase power requires four wires, three-phase power can be transmitted over 6, 4, or three wires. This saves a lot of copper as three-phase  $3\phi$  power can carry 50% more power than  $2\phi$  power for less copper.

In  $3\phi$  systems the voltages sum to zero;  $v_a(t) + v_b(t) + v_c(t) = 0 \forall t$

Four-wire three-phase power:

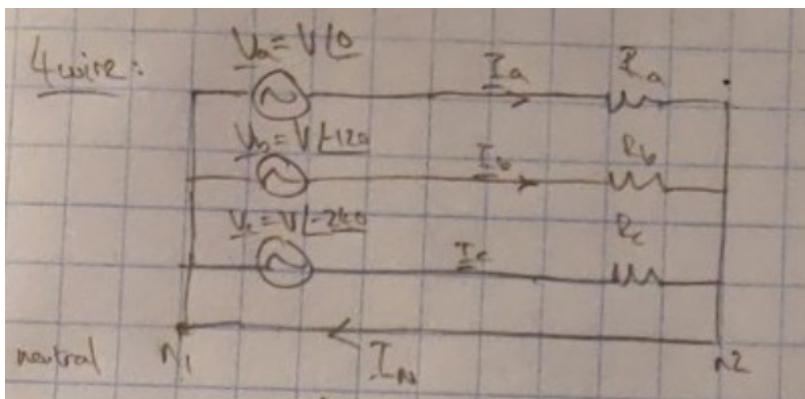


Figure 2. A four wire system for three phase power

If  $R_a = R_b = R_c = R$ , we have a balanced load and then the currents are related by  $i_n = i_a(t) + i_b(t) + i_c(t) = 0$ .

Three-wire systems drop the fourth neutral wire since it carries no current. This can be problematic if the load is not balanced; though  $I_a + I_b + I_c = 0$  will still hold true since there is

Proof: just substitute the condition earlier that voltages sum to zero and the fact that all resistances are equal into  $I = \frac{V}{R}$

no return path, the nodes at either end of the AC source and resistor pair would have differing voltages.

If the system is balanced then to save ourselves from drawing everything out all the time, only a single diagram is drawn for a characteristic phase and then solved once.

*Example*

$$Z_a = Z_b = Z_c = Z \quad (3.18)$$

$$\underline{V_b} \underline{V_a} e^{\frac{-j_2 n}{3}} \quad (3.19)$$

$$\underline{V_c} \underline{V_a} e^{\frac{-j_4 n}{3}} \quad (3.20)$$

Then,

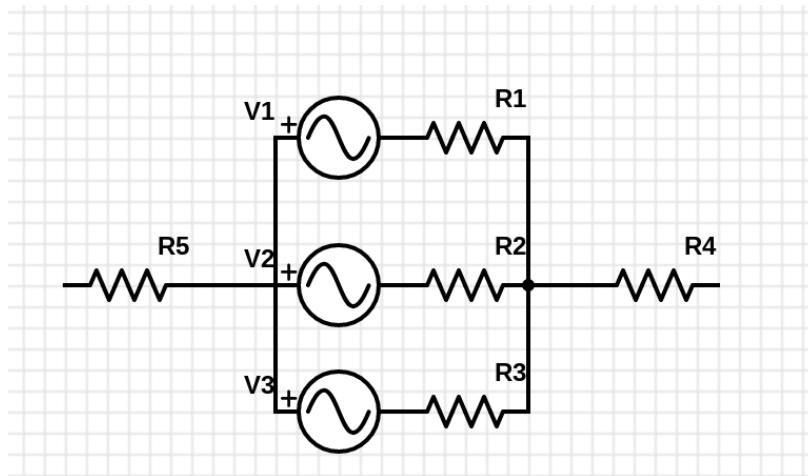
$$\underline{I_a} = \frac{\underline{V_a}}{Z} \quad (3.21)$$

$$\underline{I_b} = \frac{\underline{V_b}}{Z} = \frac{\underline{V_a}}{Z} e^{\frac{-j_2 n}{3}} \quad (3.22)$$

$$\underline{I_b} = \frac{\underline{V_c}}{Z} = \frac{\underline{V_a}}{Z} e^{\frac{-j_4 n}{3}} \quad (3.23)$$

And then the solutions for phase b and c are the same as that for phase a except for the  $120^\circ, 240^\circ$  offsets.

Because of this property, instead of drawing three separate diagrams for each phase, we can just draw one diagram and then solve for the other two phases.



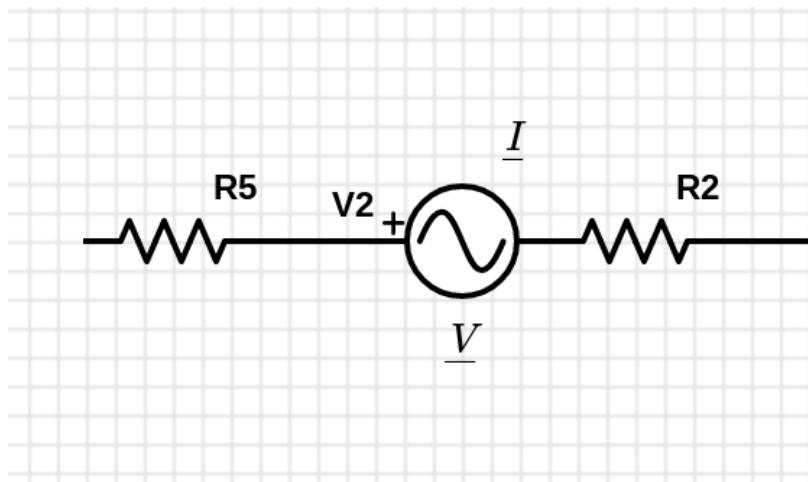


Figure 3. Condensed diagram for three phase power

SUBSECTION 3.3

**Lecture 6: Y and Delta connections**

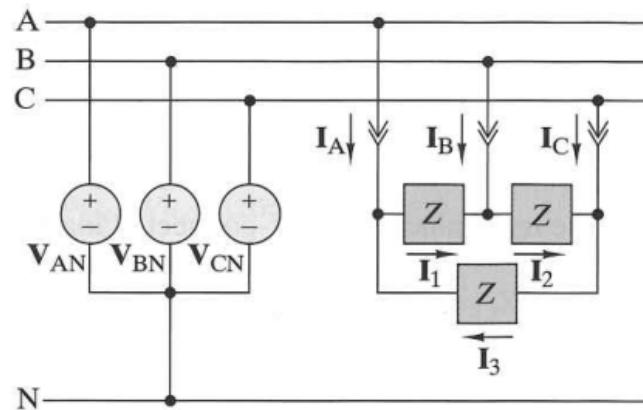


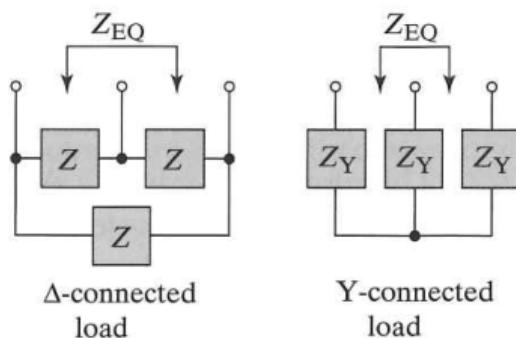
Figure 4. Three-phase system with a  $Y$  connected source and a  $\Delta$  connected load

The current to ground  $I_N$  is always 0 for a  $\Delta$  connected load.  $I_N$  is zero for a  $Y$  connected load if  $Y$  has no neutral connection and the load/source are balanced.

**Theorem 1**

**$Y - \Delta$  conversion**

Any  $\Delta$  load can be converted to an equivalent  $Y$  connected load

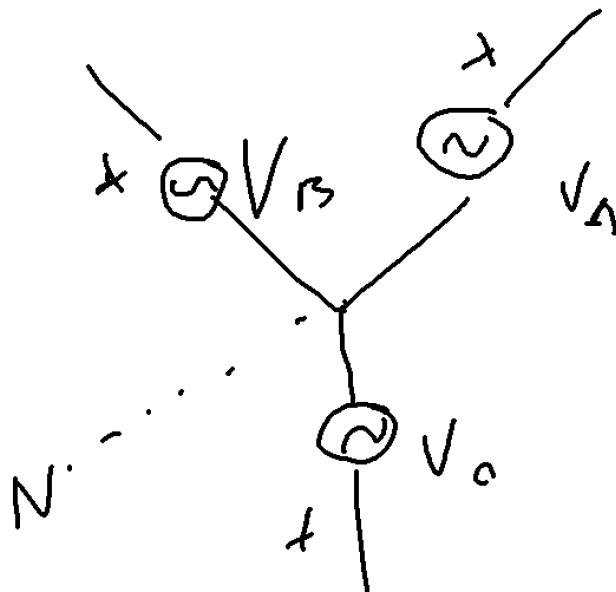


The derivation is in the textbook.

Sources can be connected in  $Y$  or  $\Delta$  configurations.

**Definition 9**

Line currents are the currents on the lines  $a, b, c$ . Phase currents are the currents immediately beside the sources.

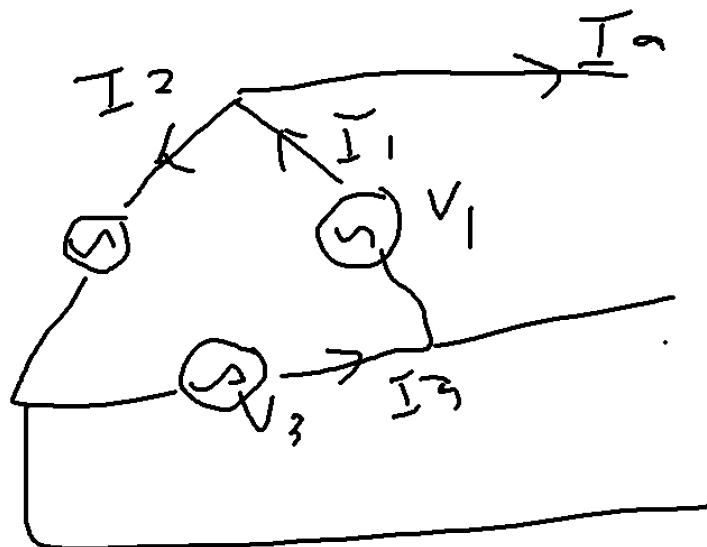


A  $Y$  connected source has the neutral line in the center.

$$\tilde{V}_{AB} = \tilde{V}_A - \tilde{V}_B = |\tilde{V}_A|(1 - 1[e^{-\frac{2\pi}{3}}]) = \sqrt{3}|\tilde{V}_A|e^{\frac{j\pi}{6}}$$

Angle differences are  
0, 120, 240 degrees

The line and phase current are the exact same in the case of a  $Y$  connected source. However, the line and phase voltages are different and are related by (3.24).



A  $\Delta$  connected source has no neutral point.

$$I_a = I_1 - I_2 = I_1(1 - 1|e^{-\frac{2\pi}{3}}) = \sqrt{3}I_1e^{\frac{j\pi}{6}} \quad (3.25)$$

The line and phase voltages are the exact same in the case of a  $\Delta$  connected source. However, the line current and phase currents are different and are related by (3.25).

Since neutral is not always available, we write  $3\phi$  systems based on their line-to-line voltages. For example, a 208V system has 120V<sub>rms</sub> on each phase.

*Example* A  $Y$  connected three-phase source has a line-to-line voltage, i.e.  $V_{A \rightarrow B}$  of 208V but each source has 120V since  $\frac{208}{\sqrt{3}} = 120V$ . A similar argument can be applied for the  $\Delta$  sources.

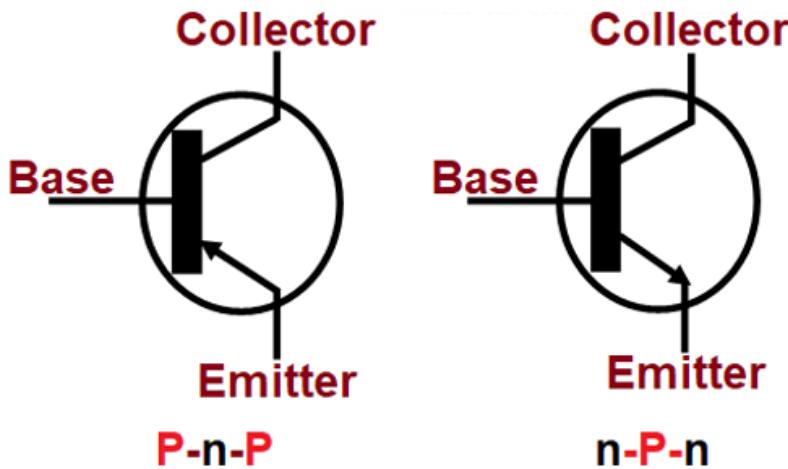
$$I = \frac{10}{\sqrt{3}} = 5.8A \quad (3.26)$$

#### SUBSECTION 3.4

### Lecture 7: DC-DC conversion

Let's say we want to supply 10V DC to a load but our supply is at 20.5V. We have a few options. We can use a resistor to drop the voltage via a voltage divide, which we learned about in ECE159. However this is not only not very efficient, but also non-responsive to changes in resistance load and will provide incorrect supply voltage if the load is not as anticipated. Another option is to use **transistors** to regulate the power instead.

semiconductors to process power  $\Leftrightarrow$  power electronics



**Figure 5.** Transistors have a Base, Collector, and Emitter. At this point in the course, we just need to know that we can change the  $V - I$  response by fiddling with the base current  $i_B$

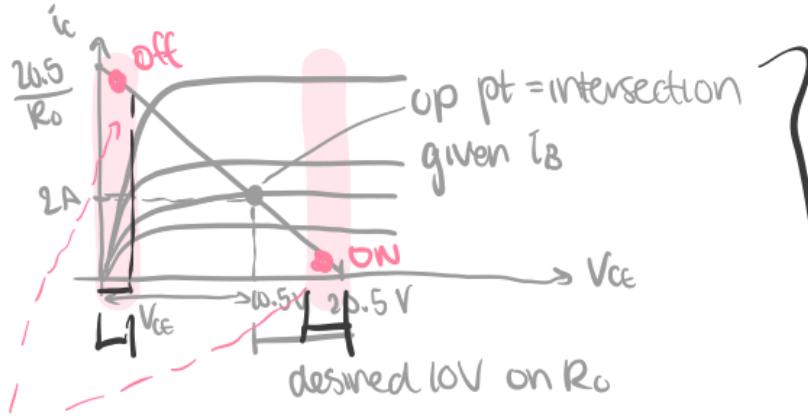


**Figure 6.** Thanks sherry for this figure

Finding the operating point that we want the transistor to be at can then be done graphically by solving a system of two equations with two unknowns, namely finding the intersection between the transistor response at a given  $i_B$  and the desired voltage/current of the load. Then the correct  $i_B$  can be selected.

However this is not very efficient since a lot of power is being wasted in the transistor. For a 20.5V source and a 10V load at 2A,  $V_{CE} = 10.5V$ , so  $P_{loss} = 10.5 * 2 = 21W$  being lost in the transistor which results in the rather poor  $\frac{20}{20.5*2=41} \rightarrow \eta = 49\%$  efficiency. So the transistor is a little bit better but we still lose half the power – which means that in a device this would halve the battery life and require a huge heat sink. How can we do better?

**Switch-mode power** is a way to maximize  $\eta$  by switching between two states that correspond to low power loss at a high frequency



The left state is the 'off' state and the right state is the 'on' state. Switching rapidly between 20.5V, 0A and 0.5V, 4A results in what is, on average what we want. This will also produce the same power output while increasing efficiency

What about the timing between the off/on states? This can be found by taking an integral to find the relationship between  $T_{on}$  and  $T_{off}$  with respect to power.

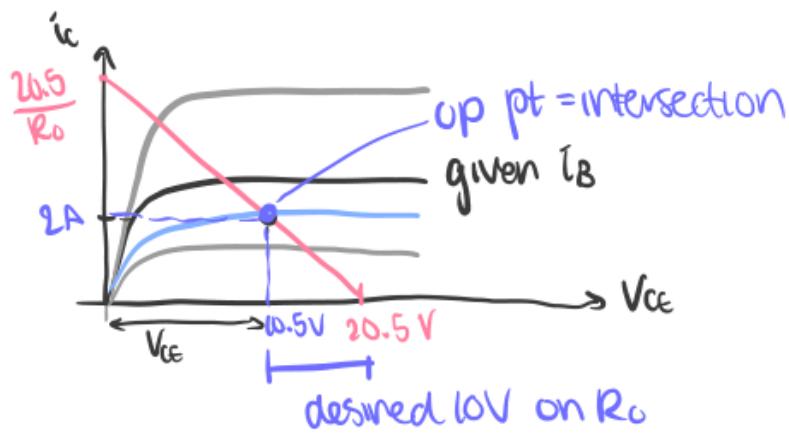
$$\begin{aligned}
 P &= \frac{1}{T} \int_0^T p(t) dt \\
 &= \frac{1}{T} \int_0^{T_{on}} i_c^2 R_o dt \\
 &= \frac{1}{T} \int_0^{T_{on}} 4^2 \cdot 5 dt \\
 &= 80 \frac{T_{on}}{T}
 \end{aligned} \tag{3.27}$$

So in order to transfer the same 20W to the  $5\Omega$  load, we would use  $T_{on}/T = \frac{1}{4}$ , which would correspond to  $P_{in} = V_i \cdot T_{on}/T = 20.5W$ . It then follows that  $\eta = \frac{20}{20.5} = 97.6\%$  which is much better<sup>1</sup>

<sup>1</sup> compared to the 49% before

A few points of caution:

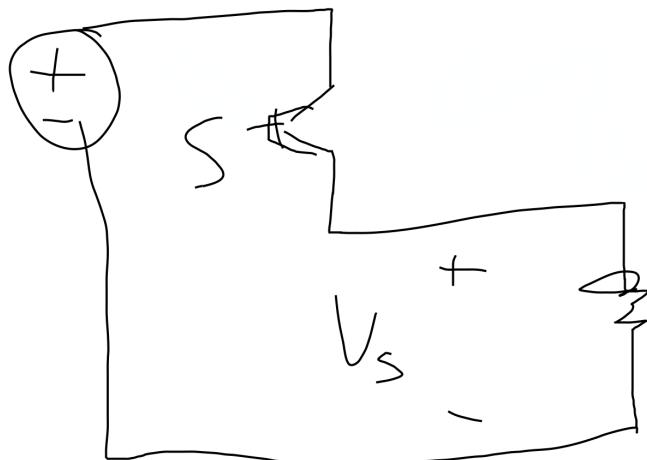
- A pulsing output voltage/power can be dangerous for some loads. For example motors aren't a fan of it. In these cases a filter can be applied to leave behind the average power.
- If we want to supply average voltage/power an on-time of 50% would be required.



## SUBSECTION 3.5

**Lecture 8: Average output voltage**

Consider a circuit with a transistor and a load.



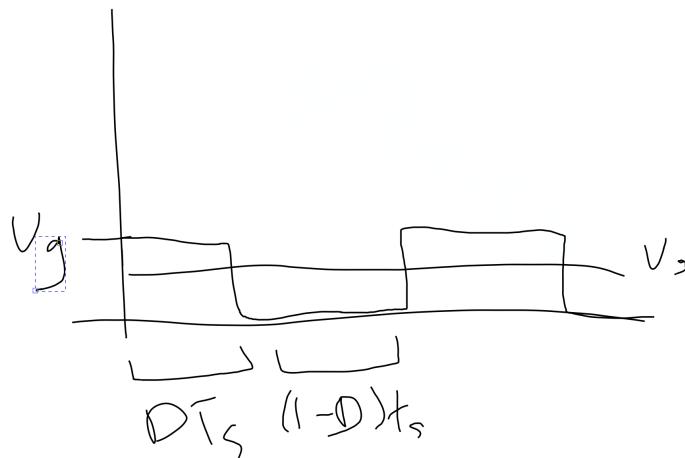
A switching signal  $S$  taking on a value from 0, 1 periodically over time is applied to the transistor

**Definition 10**

**Duty cycle:** the fraction of time that  $S$  is 1

$$D = \frac{T_{on}}{T_s} \quad (3.28)$$

The response of the circuit, where  $V_g$  is the source voltage and  $V_s$  being the voltage at the load would look like this:



One problem that we want to characterize is the ripple voltage, i.e. unwanted harmonics

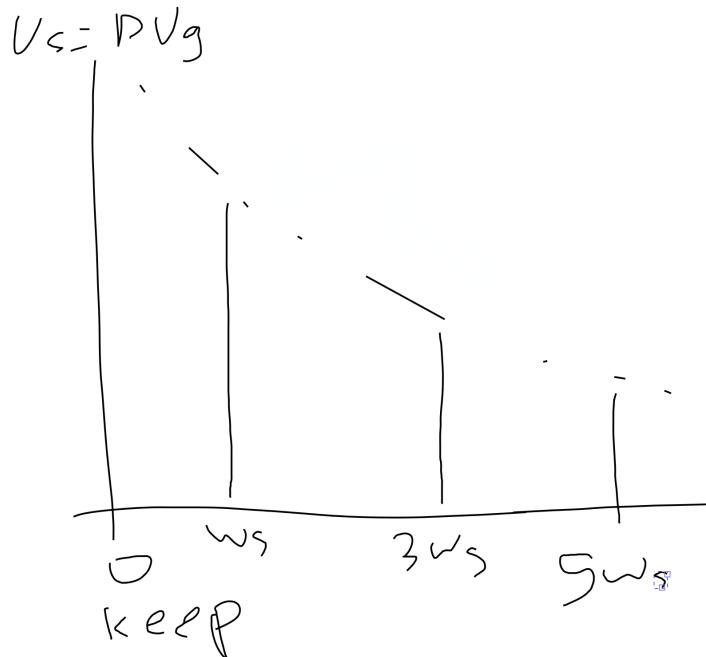
$$v_s(t) = V_s + \sum_{k=1}^{\infty} V_s^k \sin\left(k \frac{2\pi}{T_s} t\right) \quad (3.29)$$

The  $V_s$  term is desirable, and the harmonic terms  $V_s^k$  are undesirable.

$$V_s^k = \frac{2V_g}{\pi} \sin(k\pi D) \quad (3.30)$$

Basically all Fourier analysis for power electronics will basically always result in an expression with a bunch of terms on one end and then a sin or cos.  $\frac{1}{k} \frac{2V_g}{\pi}$  is the envelope of the harmonic terms. Note how the envelope decreases with  $k$ .

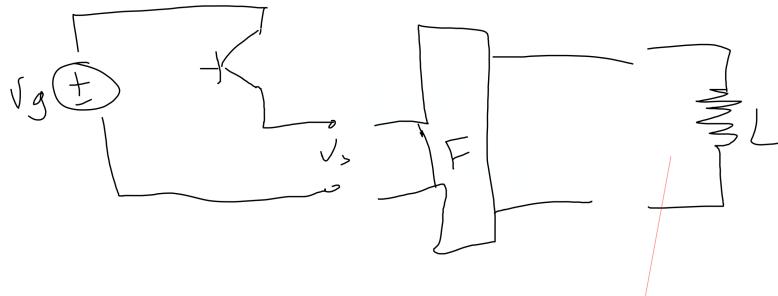
Plotting out the spectrum of signals we get:



**Figure 7.** Voltage vs frequency. We want to keep the  $\omega = 0$  term and then eliminate the rest with a filter. Note switching frequency  $\omega_s = \frac{2\pi}{T_s}$

### 3.5.1 Filtering

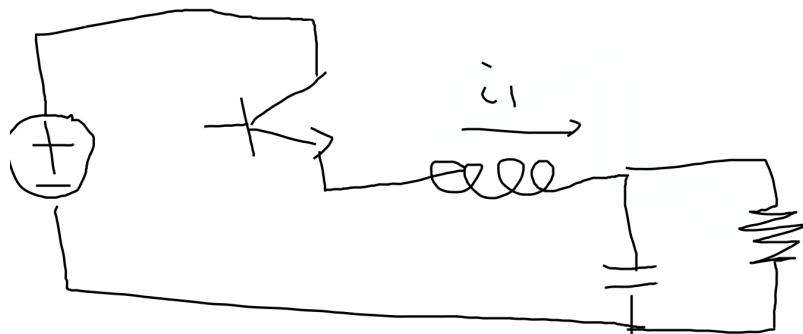
We'll need a "low pass" filter for dc/dc filtering.



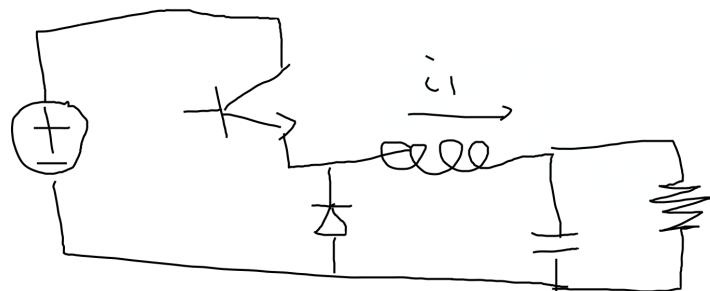
Choices for filters that may work for this system are the  $RC$ ,  $LR$ , and  $LC$  filters. The  $LC$  is most promising because it is second order and it's theoretically lossless because there is no resistor.

We are going to want to maintain continuity of capacitor voltages and inductor currents. Things work out pretty well while we're on our duty cycle, but once the duty cycle turns off and the switch is opened,  $i \rightarrow 0$ . This is a problem because this will cause us to lose all the energy stored in the inductor every time the switch is opened, which can be a lot given the frequency that these devices operate at.

$$E = \frac{1}{2} L i^2 \quad (3.31)$$



In order to handle this, we must give the current somewhere to go while off-duty cycle; an energy-recovery diode which ensures that  $i_L$  is continuous and therefore  $E_L$  is lost.



Comment

Switching should never yield discontinuity in  $i_L$  or  $v_c$ .

Some design considerations:

1. Filters all store energy during 'on' states and releasing it during 'off' states in the magnetic fields of inductors and electric fields of capacitors.
2. Faster switching  $\leftrightarrow$  less stored energy needed  $\leftrightarrow$  smaller capacitor and inductor needed
  - Switching too fast would cause the system to spend too much time during switching in undesirable high-loss states

Capacitors tend to be cheaper so designs are biased to store energy in them instead of inductors

Solving these DC-DC converters with Fourier analysis and superposition every time can be a pain, so we'll be looking at other methods next lecture. Plus, Fourier analysis requires an known  $v(t)$  or  $i(t)$  sources, which is not always the case. To motivate this, let's consider the case of an impulse being applied to the system or if the circuit contains diodes.

Periodic mapping using differential equations is a powerful method for solving transients and steady states from solving piecewise linear D.E.s. This is a whole ton of work so a third method was developed; small-ripple approximation. It only works for steady state<sup>2</sup> as it is an approximate solution to the differential equations based on periodic mapping

SUBSECTION 3.6

## Lecture 9

A periodic system is in steady state if

<sup>2</sup> Well, we only use it for steady-state

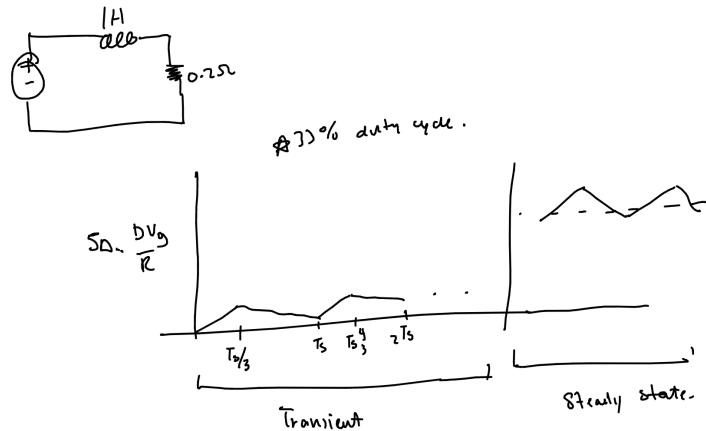
1. all input sources are periodic

2. All states are periodic

3. switching events are periodic

Consider a simple LR circuit with a 33% duty cycle.

Switching events being literally switching parts of the circuit on/off, i.e. with a transistor



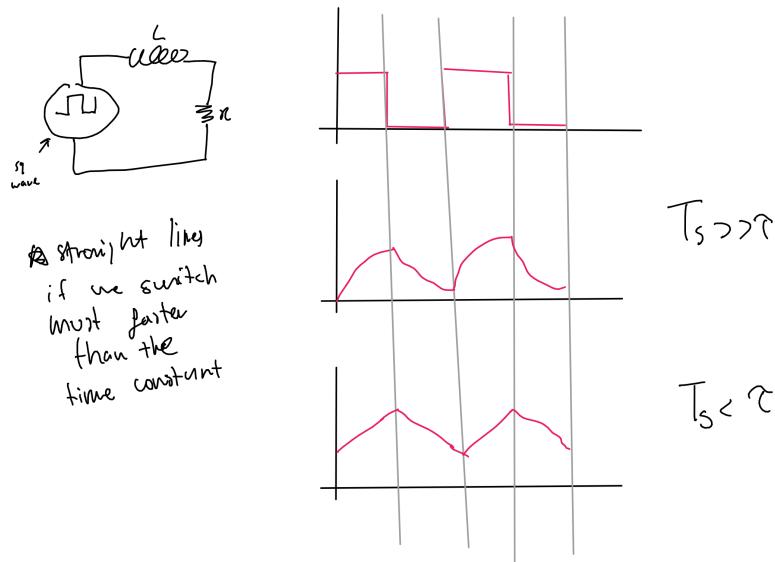
The system will have an initial transient response before settling down into a steady state. How can we find this steady state without having to solve for the transients until we get there?

Definition 11

**Steady state:** the net change of state variables over  $T_s$  is zero

An easier way to find the steady state is to use **state space averaging**. Let's look at a RL circuit and its' steady-state response.

In power electronics this is usually termed small ripple assumption



**Figure 8.** Note how the steady-state response becomes a bunch of straight lines if we switch faster than the time constant  $\tau$

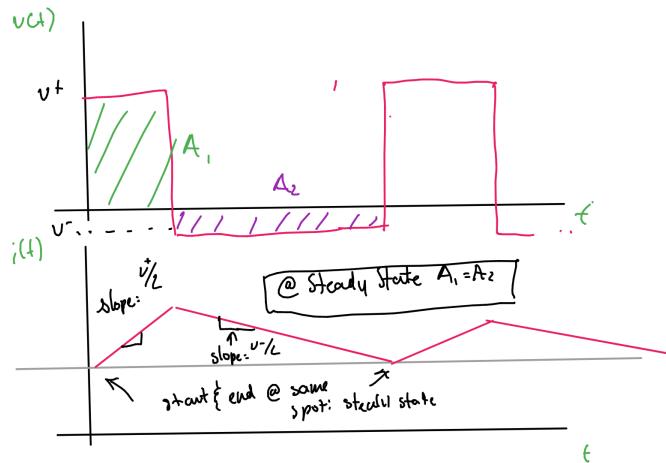
### 3.6.1 Inductor VoH-Seconds balance

$$\begin{aligned}
 v_l &= L \frac{di_l}{dt} \\
 i_l(t) &= \frac{1}{L} \int v_l(t) dt \\
 i_l(t + T_s) &= \frac{1}{L} \int_{t_0}^{t_0 + T_s} v_l(t) dt + i_l(t_0) \\
 \text{Note: } 0 &= \frac{1}{L} \int_{t_0}^{t_0 + T_s} v_l(t) dt
 \end{aligned} \tag{3.32}$$

So, in steady state  $i_l(t_0 + T_s) = i_l(t_0)$

$$0 = \frac{1}{L} \int_{t_0}^{t_0 + T_s} v_l(t) dt \tag{3.33}$$

This implies that, graphically, the area in the positive part of the  $v(t)$  plot is equal to that of the negative part.



An analogous relationship can be found for capacitors.

### 3.6.2 Capacitor charge balance

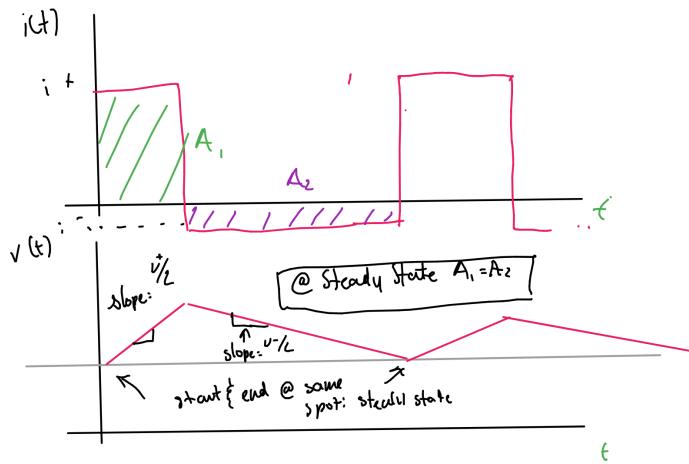
$$\begin{aligned}
 i_c(t) &= \frac{dv_c(t)}{dt} \\
 v_c(t_0 + T_s) &= \frac{1}{C} \int_{t_0}^{t_0 + T_s} i_c(t) dt + v_c(t_0)
 \end{aligned} \tag{3.34}$$

Capacitor balance gives current \* time which is charge

And at steady state

$$0 = \frac{1}{C} \int_{t_0}^{t_0 + T_s} i_c(t) dt \tag{3.35}$$

The relationship between the areas above/after the current curve and the voltage response returning to the same spots is the same as that for inductors, just flipped (i.e. voltage areas are equal for inductors but current areas are the same for capacitors).



**Figure 9.** This  $A_1 = A_2$  is the capacitor charge balance

As an aside it makes our lives easier to define the average current at steady state  $I_R$  so that instead of doing all the math relative to 0 we can take it to the center of the signal. What we really want to find is the average  $v_c$  and  $i_l$

$A_1$  is adding charge to the capacitor,  $A_2$  is removing charge

#### SUBSECTION 3.7

## Lecture 10

1. Assume all switches are ideal
  2. Apply small ripple assumption
- $$\begin{aligned} i_l(t) &\approx I_l \\ v_l(t) &\approx V_c \end{aligned} \quad (3.36)$$
3. Sketch waveforms for  $v_l(t)$  for inductors and  $i_c(t)$  for capacitors. These should generally be square in wave shape.
  4. Apply V-s balance and charge balance
  5. Validate ripple sizes and small current assumption

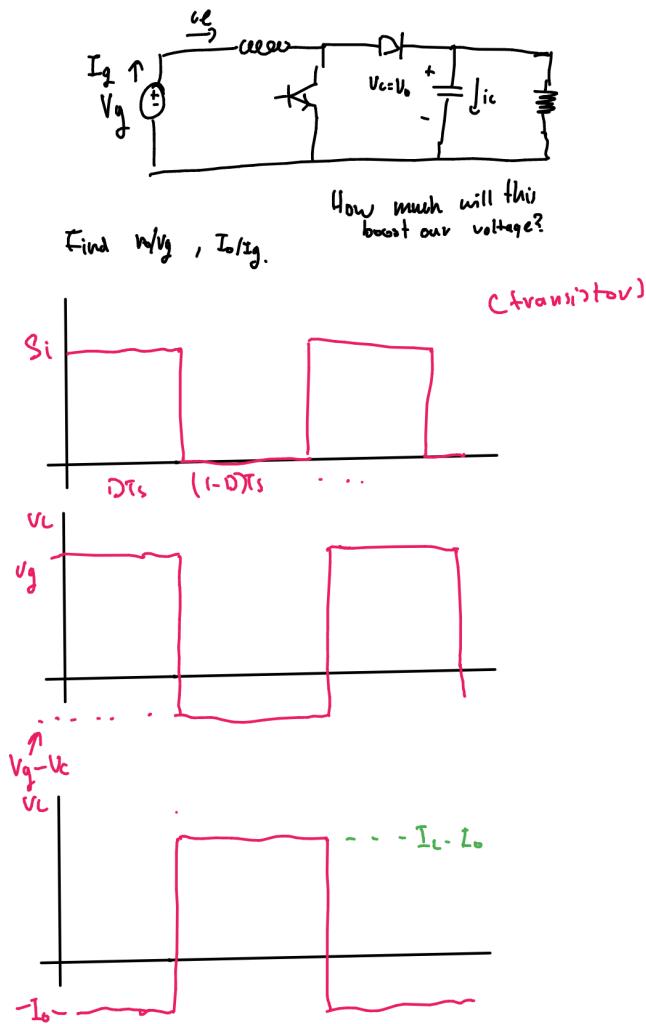


Figure 10. Example with a boost converter. Note: 3rd plot should be  $i_c$  vs  $t$

Applying the  $V - s$  balance:

$$\begin{aligned}
 \int_0^{T_s} v(t) dt &= 0 \quad \text{or: } A_1 = A_2 \\
 V_g DT_s + (V_g - V_c)(1 - D)T_s &= 0 \\
 V_g &= V_c(1 - D) \\
 \frac{V_o}{V_g} &= \frac{1}{1 - D}
 \end{aligned} \tag{3.37}$$

What this means is that we can boost  $v$  strongly by modifying the duty cycle  $D$ . In theory we can do this infinitely far but in practice after a while physical effects tend to make it really difficult to boost further.

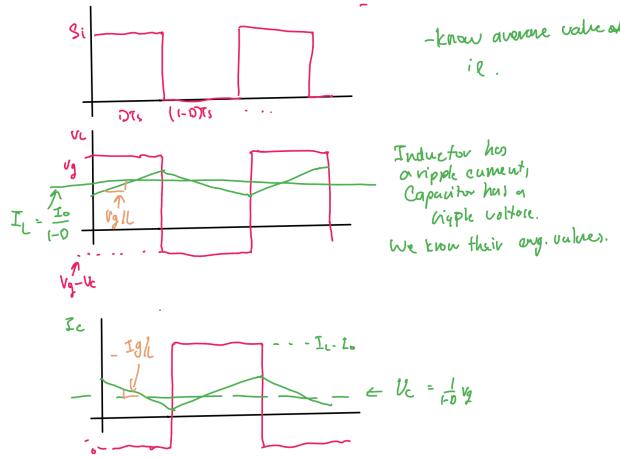
Applying a charge balance:

$$\int_0^{T_s} = i_c(t)dt$$

$$-I_o(DT_s) + (I_L - I_o)(1 - D)T_s = 0 \quad (3.38)$$

$$\frac{I_o}{(I_L = I_g)} = 1 - D$$

So the output current can similarly be modulated via the duty cycle.  
Let's check whether or not if the small ripple assumption is valid.



**Figure 11.** Inductor has a ripple current and the capacitor has a ripple voltage. The slope of the roughly saw-shaped waveform is inversely proportional to  $L, C$ . So we can pick  $L, C$  such that we get a small enough ripple.

So, what defines a 'small enough ripple'?

The amplitude of the ripple is defined from the peak to peak value of the waveform.<sup>3</sup>

$$\Delta I_{lp2p} = \frac{1}{L} \int_0^{DT_s} V_g dt = \frac{V_g}{L} DT_s \quad (3.39)$$

$$\Delta V_{op2p} = \left| \frac{1}{C} \int_0^{DT_s} -I_o dt \right| = \frac{I_o}{C} DT_s \quad (3.40)$$

Defining a small ripple is a little more hand-wavy but usually indicates the point at which our approximations start to break down. Generally it means within  $< 30\%$  of the rated current and  $< 30\%$  of the rate voltage. But this is highly dependent on the application; a computer PSU would want something a lot cleaner for example.

### 3.7.1 Design Process

1. The application will tell the load;  $I_o, V_o$
2. We will have to pick the switch. It'll have to meet the current spec and maybe some thermal design
3. Look at switch data sheet and then look at the switching frequency that it can run at<sup>4</sup>
4. Compute required duty cycle  $D$  for the desired output characteristics  $V_o/V_g$

These integrals can be found by looking at the graphs we drew with the square waves etc and the areas

<sup>3</sup> The textbook defines it as center to peak, but prof likes peak2peak

$V_o = V_c$ . Generally we don't care about  $I_{lp2p}$ ; it is an internal variable. User only cares about  $V_o$ , not  $V_c$  – they're the same in this case but not necessarily.

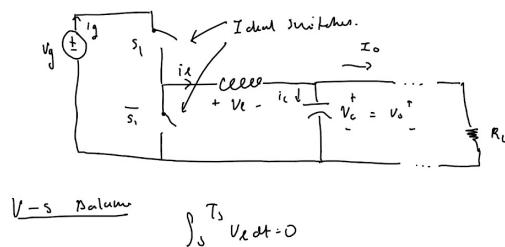
<sup>4</sup>Usually smaller switches can run faster than larger ones

- Pick inductor current to achieve the small ripple assumption to get reasonable  $\Delta I_{lp2p}$ . Inductors are more expensive than capacitors so most try to get it to within  $< 20\%$  of rated and do the rest using capacitors
  - Pick  $C$  to meet  $\Delta V_{cp2p}$  specification

### SUBSECTION 3.8

## Lecture 11

I slept in and was behind on notes so here are handwritten ones.



$$(V_q - V_c) D T_s - (1 - D) T_s V_c = 0$$

$$U_c = \rho V g$$

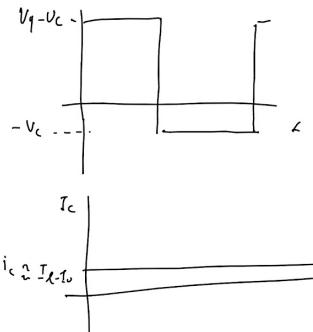
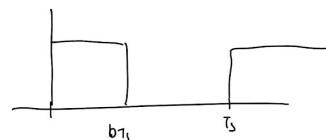
$$\frac{U_0}{Y_0} = D$$

$$\int_0^T i_c dt = 0$$

$$\int_{0}^{T_1} (I_L - I_{L0}) dt = 0$$

$$I_C - \frac{I_s}{2} = 0, \quad I_C \approx 0.$$

$$\frac{f_0}{T_1} =$$



→ Takes away

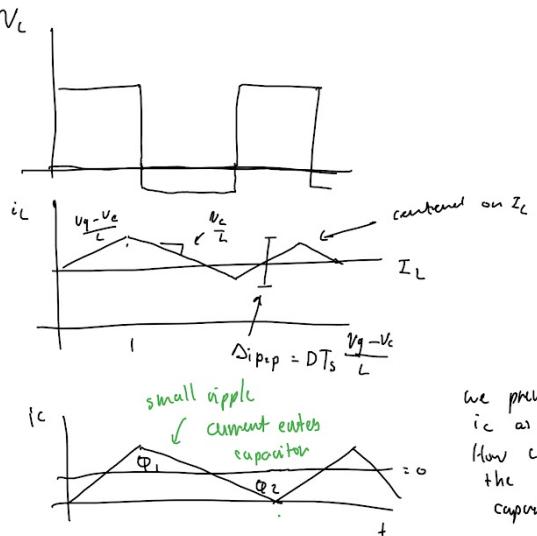
$$I_q = -\frac{1}{T_s} \int_0^{T_s} i_q dt \rightarrow \bar{I}_q = 0 I_C$$

$$\frac{I_0}{I_9} = \frac{1}{D}$$

$$\star I_0 - I_c \approx 0$$

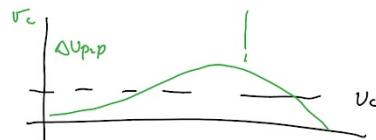
$$\therefore I_5 > I_6$$

Steady ripple to size  $L, C$



we previously treated  
iC as a straight line.  
How can we approximate  
the value of the  
capacitor current iC?

The difference between  $i_L, i_C$  should  
actually cause a small ripple  
in real world currents.



$$\Delta i_{p,p} = D T_s \frac{V_q - V_0}{L} \quad \text{Design eq #1}$$

$$\Delta V_{p,p} = \frac{Q_1}{C} =$$

$$Q_1 = \frac{1}{2} \frac{T_s}{2} \frac{\Delta i_{p,p}}{2}$$

$$= \frac{T_s}{8} D T_s \frac{V_q - V_0}{L}$$

$$\Delta V_{p,p} = \frac{T_s^2}{8} D \frac{V_q - V_0}{L C} \quad \text{Design eq #2}$$

$$V_0 = DVq \quad \therefore V_q = \frac{1}{D} V_0$$

$$(V_q - V_0) = \left( \frac{1}{D} - 1 \right) V_0$$

$$= \frac{1}{D} (1 - D) V_0$$

can multiply  
in terms  
of  
other...

$$\frac{\Delta V_{p,p}}{V_0} = \frac{T_s^2}{8} \frac{(1 - D)}{L C}$$

Alternative  
design eq

**Lecture 12**

PART

II

***ECE352: Computer Organization*****Admin stuff**

Taught by Prof. Andreas Moshovos

**Lecture 1**

- Lecture recordings on [YouTube](#)
- Online notes: <https://www.eecg.utoronto.ca/~moshovos/ECE352-2022/>
- Course will cover the following:
  - C to assembly
  - How to build a processor that works
  - Intro to processor optimizations
  - Peripherals
  - OS support (Maybe)
  - (Maybe) Arithmetic circuits
  - Use NIOS II and cover a little bit of RISC-V

**4.1.1 Mark breakdown**

- Labs 15%
- project 5%
- midterm 30%
- Final 50%
- All exams will be open notes/book/whatever except another person/service helping you.

**Preliminary****Lecture 2: Using binary quantities to represent other things**

Computers can represent information in bits; 0/1. Though they don't necessarily know or care what bits are, we may assign our own arbitrary meaning to them – usually numbers with the help of positioning; the LSB represents  $2^0$  and so forth.

C types

- int: 32b (word)

Or, just ; ; include  
<stdint.h>...

- char: 8b (byte)
- short: 16b (half word)
- long: 32b (word)
- long long: 64b

Signed numbers may be represented in a number of ways.

- Sign bit (make MSB represent positive or negative numbers and then the remaining  $n-1$  bits represent the number. Con: hardware impl sucks because requires if/else)
- Two's complement<sup>5</sup>. Pro: only need to implement adders on hardware and then negative numbers will work just like any other except must be interpreted differently. Positive numbers would always start with a 0 and negatives would start with 1. So the range of possible values becomes  $-(2^{n-1} - 1), +2^{n-1} - 1$

Adding together binary numbers can also cause overflow;  $(A + B) \geq A, (A + B) \geq B$  may not always be true. Also, when we work with these types we always use all the bits. This has implications when working with values of different lengths.

- `char b = -1 (1111 1111)`
- `short int c = -1 (0000 0000 0000 0001)`
- `a = b + c 0000 0001 0000 0000`
- In order to deal with this we must `cast` the `char` to a `short int`. This is done via sign extension which prepends 0s or 1s<sup>6</sup> to the `char` so that math can be done on it.

<sup>5</sup> Flip bits, add one. Intuition; in 3 bit system, adding 7 to 1 would result in 8 which would get truncated to 0.

<sup>6</sup> two's complement

### 5.1.1 Floating Point Numbers

Whereas fixed point numbers i.e. \$5.25 can be represented just as how an integer would be represented but with the understanding that the user would interpret it as having a decimal point somewhere that indicates the position of  $2^0$ . This decimal point would be the same for all numbers of that type, i.e. we could have a six bit number that has places  $2^2 2^1 2^0 2^{-1} 2^{-2}$ . This is common in embedded systems and how it is formatted isn't super clearly standardized.

**Lemma 1** | Reference: [What Every Computer Scientist Should Know About Floats](#)

**Definition 12**

#### IEEE 754 Floating Point

This is a single precision 32 bit float

S EEEEEEEE MMMM MMMM MMMM MMMM MMMM MMMM MMMM (5.1)

The most significant S bit is the sign bit, bits 30 through 23 E form the exponent which is an unsigned integer, and 22 through 0 form the (M)antissa. The number being represented can be found using the following:

$$(-1)^S \times 2^{(E-127)} \times 1.\text{Mantissa} \quad (5.2)$$

*Example*

For example, given the following float:

1 10000001 10000000000000000000000000000000

So S = 1, E = 10000001 = 129 and Mantissa = 10000000000000000000000000000000. The number is therefore

$$(-1^1) \times 2^{(129-127)} \times 1.1000000000000000000000000000000 = -6.0 \quad (5.3)$$

IEEE754 also defines 64 bit floating-point numbers. They behave the same except for now having an 11 bit exponent, the bias being 2047<sup>7</sup>, and the mantissa having 52 bits. A few special cases are also available to represent other quantities

- If E=0, M non-zero, value= $(-1)^S \times 2^{-126} \times 0.M$  (denormals)
- If E=0, M zero and S=1, value=-0
- If E=0, M zero and S=0, value=0
- If E=1...1, M non-zero, value=NaN 'not a number'
- If E=1...1, M zero and S=1, value=-infinity
- If E=1...1, M zero and S=0, value=infinity

Floating-point numbers are inherently imprecise. Addition and subtract are inherently lossy; the mantissa window may not be large enough to capture the decimal points. Multiplication and division just creates a ton of numbers.

Converting real numbers to IEEE754 floats, here using 37.64 as an example, can be done as follows

- Repeatedly divide the part of the number  $> 0$  by 2 and get the remainders, i.e.  $37/2 = 18$ , rem = 1  $\rightarrow 18/2 = 0$ , rem = 0  $\rightarrow 4/2 = 2$ , rem = 0,  $2/2 = 1$ , rem = 0,  $1/2 = 0$ , rem = 1. As a 2 bit number E is 100101. But we need to convert it to IEEE754 format with the exponent; E - 127 = 5, E = 132 = 1000 0100.
- Do the same for the part of the number past the decimal, but multiplying by two and checking if  $> 1$ :  $0.64 * 2 = 1.28 \rightarrow 1$ ,  $0.28 * 2 = 0.56 \rightarrow 0$ ,  $0.56 * 2 = 1.12 \rightarrow 1$  ... and so forth. At some point we will hit a cycle but we'll just take the  $N_{\text{mantissa}}$  of digits.

So the full number is 01000010000101101000111101011111

`float` is a 32 bit float and `double` is 64

There are more floating point formats introduced by nvidia and google such as a half-precision or 8-bit float designed to reduce memory use for machine learning

## SECTION 6

# Memory and NIOS II

### SUBSECTION 6.1

## Lecture 3: Behavioural Model of Memory

Computers can be described as a set of units, each of which interact with each other and the outside world in a specified way. For example, modern computers tend to have memory units, processing units, display units, and so forth. Each unit has a set of inputs and outputs, and a set of rules that govern how the unit behaves. This gives the manufacturer flexibility in how they want to implement a unit, as long as the unit behaves as specified. When designing these operational units it is important to strike a balance between functionality and specificity; if

the unit is too specific it will be difficult to implement, but if it is too general it will be difficult to use.

### 6.1.1 Memory

Memory is a unit that stores information and is usually represented as a vector of elements, usually a byte (8 bites). Each element, or memory location, contains a binary quantity and has an associated *address*. The address is a number that uniquely identifies the location of the element in the vector, and is **permanently fixed** at time of manufacture. Most systems are byte-addressable, meaning that there is an unique address for each byte in the memory. The collection of all addresses is called the **address space** of the memory, which is typically a power of two. Modern systems tend to be 32 or 64 bit, meaning that the address space is  $2^{32}$  or  $2^{64}$  elements long.

For each memory location there are two operations available

- **Read:** Read the value stored at a given address
- **Write:** Write a value to a given address

Typical memory behaviour models define that the order of memory operations matters, i.e.

1. **store** 0x10, 0xf0
2. **store** 0x20, 0xf0

We would see that 0xf0 contains the value 0x20 and not 0x10 due to the sequential execution model. Memory that adheres to the sequential model offers operations that are **atomic**; the operations are performed on its own with no interaction or overlap with anything else.

In this case it is convenient to draw memory as an array where each row comprises four consecutive bytes.

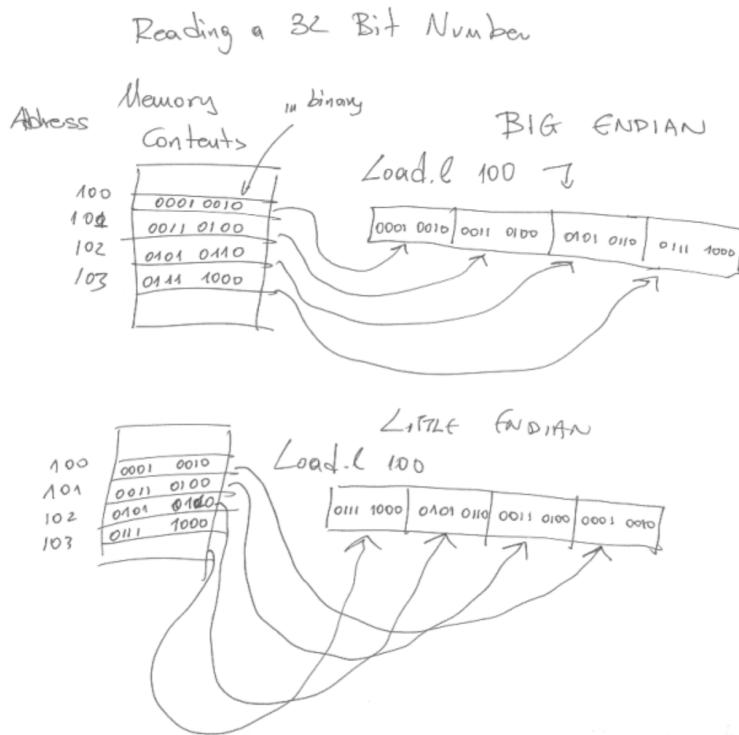
0x00 0000	0x11	0x22	0x33	0x44
0x00 0004	0xff	0x88	0x62	0x51
...				
0xff ffff				

Systems are generally also addressable by words, halfwords, and bytes. Different architectures have different constraints on allowing unaligned access<sup>8</sup>

**Endianness** refers to the order in which bytes are stored in memory. Though some processors are big-endian, most modern processors are little-endian. The NIOS II used for this course is little-endian.

**Specification** is the description of what an unit should do, and **implementation** is how it actually does it. For example, an OR gate can be specified as a truth table and then implemented via transistors or a person in a box.

<sup>8</sup> Aligned access only means to allow [only] reads or writes for a data size i.e. halfword to an address divisible by the size of said data type. For example an longword access on our development board would be at an address divisible by 4



### 6.1.2 Physical Interface

What physical interfaces would be necessary to implement this behavioural model?

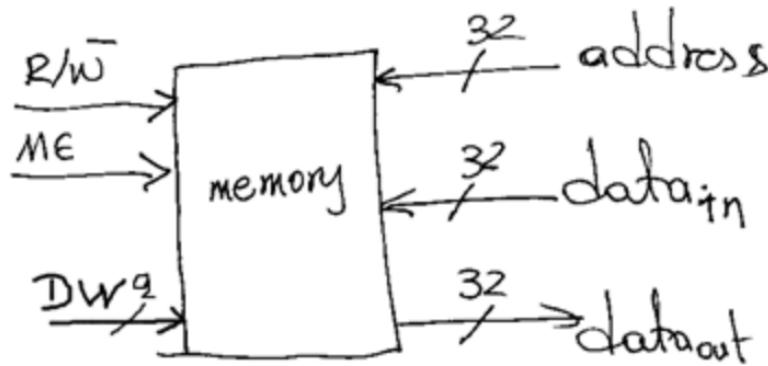
Given a summary of requirements as follows:

1. Read and write operations
2. Addressable by byte, word, longword
3. 24 bit address
4. 32 bit for writing
5. 32 bit for reading
6. signal for do nothing

A single bit signal can be used to indicate whether the memory is reading or writing, and a two bit signal can be used to specify if we're interested in addressing by byte, word, or longword. The address is 24 bits, so we need 24 address lines. As for reading/writing data, we have the option of having two 32 bit data lines, or multiplexing a single 32 bit line. A single bit signal can be used to indicate to do nothing or not.

One way of multiplexing the data lines is to use a tri-state buffer, which is a buffer that can be enabled or disabled. When enabled, the buffer acts as a normal buffer, but when disabled, the output is disconnected from the input. On the other hand this means that our memory chip would not support simultaneous reads or writes.

The use of a single bit signal to indicate 'do nothing' is necessary because a physical device won't be able to change all signals instantaneously, so we use it to tell the memory to wait until these transient effects die off



## SUBSECTION 6.2

**Lecture 4: NIOS II Programming Model**

The NIOS II assumes a 32-bit address space where each address holds a single byte. Each byte is addressable, and three data types are supported. Halfword and word accesses must be aligned.

- **Byte:** 8 bits
- **Halfword:** 16 bits
- **Word:** 32 bits

The NIOS II also has a set of registers

- 32 general purpose 32 bit registers
  - $r0$  is always zero
- 6 control registers, 32 bits each
- Program counter (PC), 32 bits

There are certain conventions for the use of registers, which are as follows:

Many operations can be synthesized using another operation involving zero, i.e. assignment  $A=B$  can be implemented as  $A = B + 0$

Register	Name	Function	Register	Name	Function
r0	zero	0x00000000	r16		
r1	at	Assembler Temporary	r17		
r2		Return Value	r18		
r3		Return Value	r19		
r4		Register Arguments	r20		
r5		Register Arguments	r21		
r6		Register Arguments	r22		
r7		Register Arguments	r23		
r8		Caller-Saved Register	r24	et	Exception Temporary
r9		Caller-Saved Register	r25	bt	Breakpoint Temporary (1)
r10		Caller-Saved Register	r26	gp	Global Pointer
r11		Caller-Saved Register	r27	sp	Stack Pointer
r12		Caller-Saved Register	r28	fp	Frame Pointer
r13		Caller-Saved Register	r29	ea	Exception Return Address
r14		Caller-Saved Register	r30	ba	Breakpoint Return Address (1)
r15		Caller-Saved Register	r31	ra	Return Address

*Notes to Table 3-1:*  
(1) This register is used exclusively by the JTAG debug module.

### 6.2.1 Adding Two Numbers

As an exercise, let's see how we can implement the following piece of code in NIOS II assembly

```

1  unsigned int a = 0x00000000;
2  unsigned int b = 0x00000001;
3  unsigned int c = 0x00000002;
4
5  a = b + c;

```

#### Register-only version

```

1  addi r9, r0, 0x1
2  addi r10, r0, 0x2
3  add r9, r10, r11

```

addi stands for 'add intermediate', the only difference being that the second operand is a number. It is used to set a constant

In general, most instructions take the form of **operation destination, source1, source2**.

Breaking it down even further we can see that these assembly instructions actually perform a number of steps

```

1 addi r9, r0, 0x1
2 ; 1. read r0
3 ; 2. Add value read in step 1 with 0x1
4 ; 3. Write result of step 2 to r9
5 ; 4. increment PC to next instruction
6 addi r10, r0, 0x2
7 ; 1. read r0
8 ; 2. Add value read in step 1 with 0x2
9 ; 3. Write result of step 2 to r10
10 ; 4. increment PC to next instruction
11 add r9, r10, r11
12 ; 1. read r10
13 ; 2. read r11
14 ; 3. Add values read in steps 1 and 2
15 ; 4. Write result of step 3 to r8
16 ; 5. increment PC to next instruction

```

What about 32 bit constants? An unfortunate quirk is that `addi` only supports 16 bit constants, so we need to use `ori` to set the upper 16 bits of the register.

```

1 movhi r9, 0x1122
2 ; Sets the upper 16 bits of r9 to 0x1122
3 ; and the lower 16 bits to zero
4 ori r9, r9, 0x3344
5 ; bitwise OR the value in r9 with 0x3344
6 ; which will set the lower 16 bits to 0x3344

```

This is a PITA so NIOS II offers a few pseudo-instructions to make this easier

```

1 movi rX, Imm16
2 ; sets rX to the sign-extended (signed) 16 bit
3 ; immediate
4 movui rX, Imm16
5 ; sets rX to a zero-extended unsigned 16 bit immediate
6 movia rX, Imm32
7 ; sets rX to a 32 bit immediate

```

- 
- Footnote1: `movia` does not use the `movhi` and `ori` instructions to create a 32-bit immediate but rather a `movhi` and a `addia`. `addi` will sign extend it's 16-bit field so some adjustment might be needed for whatever is being passed to `movhi`.
  - Footnote2: `movhi r9, %hi(0x11223344)` is equivalent to `movhi r9, 0x1122`. `Ori r9, %lo(0x11223344)` is equivalent to `ori r9, 0x3344`. That is, `%hi(Imm32)` returns the upper 16-bits of `Imm32` and `%lo(Imm32)` the lower 16 bits.
  - Footnote3: `movhi r9, %hiadj(0x11223344)` followed by `addi r1, %lo(0x11223344)` is the correct way of creating a 32-bit immediate using `movhi` and `addi`. `%hiadj(Imm32)` returns the upper 16 bits of the immediate as-is or incremented by 1 if bit 15 is 1. Think why this is necessary based on footnote 1.
  - Footnote4: `%hi()`, `%lo()`, and `%hiadj()` are macros supported by the assembler. They are not NIOS II instructions. They get parsed during compile time.

### 6.2.2 Adding two numbers using memory

NIOS II is a load/store architecture which means that all data manipulation happens only in registers.

```

1 ; read b from memory into r9
2 movhi r11, 0x0020
3 ori r11, r11, 0x0004
4 ldw r9, 0x0(r11)

5
6 ; read c from memory into r10
7 movhi r11, 0x0020
8 ori r11, r11, 0x0008
9 ldw r10, 0x0(r11)

10
11 ; add, then store into r8
12 add r8, r9, r10

13
14 ; store r8 into memory
15 movhi r11, 0x0020
16 ori r11, r11, 0x0000
17 stw r8, 0x0(r11)

```

The new instructions introduced here are

```

1 ldw rX, Imm16(rY) ;; 'load word' from memory
2 ;; rX, rY registers, Imm16 is a 16 bit immediate
3 ;; TLDR; Rx = mem[rY + sign-extended(Imm16)]
4 ; 1. read rY
5 ; 2. sign-extend Imm16 to 32bits
6 ; 3. adds the result of step 1 and 2
7 ; 4. reads from memory a word (32 bit) using the result of step
   ↳ 3 as the address
8 ; 5. write the result of step 4 to rX

```

```

1 stw rX, Imm16(rY) ;; 'store word' to memory
2 ;; rX, rY registers, Imm16 is a 16 bit immediate
3 ;; TLDR; mem[rY + sign-extended(Imm16)] = rX
4 ; 1. read rY
5 ; 2. sign-extend Imm16 to 32bits
6 ; 3. adds the result of step 1 and 2
7 ; 4. write to memory rX using the result of step 3 as the
   ↳ address

```

This can be simplified using the `movia` macro

```

1 movia r11, 0x200004
2 ldw r9, 0x0(r11)
3 movia r11, 0x200008
4 ldw r10, 0x0(r11)
5 add r8, r9, r10
6 movia r11, 0x200000
7 stw r8, 0x0(r11)

```

In this lecture so far we have seen three addressing modes

1. Register addressing, i.e.  $rX$
2. Immediate addressing, i.e.  $Imm16$
3. Register indirect addressing with displacement, i.e.  $Imm16(rY)$ . This is how we calculate the referenced memory address. Register indirect refers to using a register's value to refer to memory, and 'displacement' refers to adding a constant prior to using the register value to access memory. Register indirect addressing is where we use a displacement of 0.

We can exploit register indirect addressing with displacement.

```

1 movhi r11, 0x0020
2 ori r11, r11, 0x0004
3 ldw r9, 0x0(r11)
4 ; can be replaced with
5 movhi r11, 0x0020
6 ldw r9, 0x4(r11)

```

Note that the value of  $r11$  does not change since the subsequent operations use an offset to that value.

Generally when we want to read memory from A we can use

```

1 movhi r11, (upper 16 bits of A)
2 ori r9, r11, (lower 16 bits of A)

```

Care must be taken when the 16th bit of A is 1 since the addition that  $ldw$  performs will sign extend it to be a negative number, i.e.

```

1 movhi r11, 0x0020
2 ldw r9, 0x8000(r11)
3 ; this is incorrect because
4 ; will extend to 0xFFFF8000, which would result
5 ; in a final address of 0x001F800

```

This is where the macros `%hiadj(Imm32)` and `%lo(Imm32)` come in handy, since they will add 1 to the values if bit 15 of Imm32 is 1. This results in code that looks like this:

```

1 movhi r11, %hiadj(0x208000)
2 ldw r9, %lo(0x208000)(r11)
3 ;; will extend to 0xFFFF8000, which would result
4 ;; in a final address of 0x001F800

```

## SECTION 7

## Assembly Basics

---

## SUBSECTION 7.1

### Lecture 5: Simple Control Flow

---

We have prior worked with straight-line sequences. In this lecture we will look at how to add control flow to our programs, i.e if-then-else, etc.

A pseudo-c program will be rewritten in assembly to illustrate the concepts.

```

1 unsigned int a = 0x00000000;
2 unsigned int b = 0x11223344;
3 unsigned int c = 0x22334455;
4
5 if (b == 0)
6 then a = b + c;
7 else a = b - c;
8

1 .section .data
2 va: .long 0x0
3 vb: .long 0x11223344
4 vc: .long 0x55667788
5
6
7 main:
8     movia r11, va
9     ldw r9, 4(r11)
10    beq r9, r0, then
11 else:
12    ldw r10, 8(r11)
13    sub r8, r9, r10
14    stw r8, 0(r11)
15    beq r0, r0, after
16
17 then:
18    ldw r10, 8(r11)
19    add r8, r9, r10
20    stwio r8, 0(r11)
21 after:

```

The `data` section contains stuff that you want to be initialized for you before the entry point of the program is called, e.g. global variables. This segment as a fixed size. The `text` or `code` segment contains executable instructions (typically read-only, unless the architecture allows self-modifying code) and typically resides in the lower parts of memory. `bss` contains static and global variables which are zero-initialized; usually used for uninitialized data

## Definition 13

We encounter two new instructions in this snippet.

- `sub` is a subtraction instruction
- `beq`: a branch-if-equals instruction

The `beq` instruction takes the general form  
`beq RX, rY, label.`

This instruction will compare the values of  $rX$  and  $rY$  and if the condition is true then the program counter will jump to the destination label. When the branch changes the program counter it is called a taken branch, otherwise it is non-taken. Non-taken branches fall through to the next instructions.

## Comment

Note: whereas the assembly `beq` command is written as a comparison with a label to jump to, in the NIOSII instruction the destination is encoded relative to the instruction location with a 16-bit displacement constant. The displacement is therefore calculated as  $PC + 4 + \text{displacement}$ , meaning that we can jump at most  $+32774, -32772$  bytes<sup>9</sup> from the current program counter. The compiler will complain if the branch cannot be implemented.

Encoding is as follows:

AAAAAA BBBBB IIIIIIIIIIIIIII 0x26.

where  $A, B$  are register names (hence 5 bit),  $I$  is a 16 bit immediate value, and  $0x26$  is the branch type, in this case `beq`

Most branches tend to be branch backwards because of loops.

<sup>9</sup> 16 bit signed constant+4, with 4-byte alignment since instructions are 4 bytes long.

Other branch instructions include

- `br`: always/unconditional branch
- `bne`: branch if not equal
- `blt`: branch if less than, w/ signed comparison
- `bltu`: branch if less than, w/ unsigned comparison
- `bgt`: branch if greater than, w/ signed comparison
- `bgtu`: branch if greater than, w/ unsigned comparison

```

1      .data
2      .align 4 ; Align to word size addresses which are
3      ↳ faster to access
4      a: .word 0
5      b: .word 0x11223344
6      c: .word 0x55667788
7      .text
8      movia r11, a ; moves the address of a into r11
9      ldw r9, 4(r11) ; loads the value at address a+4 into
10     ↳ r9
11     ldw r10, 8(r11)
12     add r8, r9, r10
13     stw r8, 0(r11)

```

SUBSECTION 7.2

## Lecture 6, 7: For loops and arrays

How can we implement the following in assembly?

```

1 short arr[5] = { 1, 2, 3, 4, 5 }; // an array of word values
2   ↵ (16 bit)
3 short n = 5;           // the number of elements in the array
4 short sum = 0;
5
6 for (i = 0; i < n; i++)
  sum = sum + arr[i];

```

Arrays are typically implemented at the machine level as a contiguous block of memory.

```

1 .data
2   .align 1
3 arr .hword 1,2,3,4,5
4 n   .hword 5
5 sum .hword 0

```

Listing 1: Creating a static array in assembly

Multidimensional arrays are handled by having an array of pointers to arrays and dereferencing twice to arrive at the desired memory location. Alternatively we can allocate a  $1 \times (m \cdot n)$  chunk of memory and then address it as  $a[i][j] = a[i \cdot n + j]$ .

As for the for loop, let's look at what C does first.

```

1 for (init; cond; post)
2   body

```

Listing 2: General form of a C for loop

Breaking it down a little more into atomic steps

```

1 INIT
2 if (!COND), we are done
3 BODY
4 POST
5 GOTO line 2

```

Listing 3: General form of a C for loop

Let's now rewrite this in assembly

Generally element  $a[i]$  is at address  $\&a[0] + \text{sizeof}(\text{TYPE}) * i$

C and pascal and other modern programming languages store arrays in row-major order, i.e. consecutive elements in a row are placed together in memory. FORTRAN uses column-major.

```

1          .text
2 forloop:
3     add r8, r0, r0 ; INIT
4     movia r9, n ; set COND, i.e. the n that we compare i
5     → to. See: loop invariant :)
6     ldh r9, 0(r9)
7     ;; movhi r9, %hiadj(n)
8     ;; ldh r9, %lo(n)(r9) ; this code block is a shorter piece
9     → with the same effect
10    loop:
11        bge r8, r9, endloop ; test condition for when we are
12        → done
13        ;; body goes here
14        ;; let's use r10 to store the running sum at in the end
15        → we can load it to memory
16        movia r11, arr ; load the address of arr[0] into r11
17        ;; add the index to the address of arr[0]; &arr + i
18        add r11, r11, r8
19        ;; add again; &arr + i + i = &arr + 2i
20        add r11, r11, r8
21        ldhio r12, 0(r11) ; r12 = arr[i]
22        add r10, r10, r12 ; r[10] += arr[i]
23
24        addi r8, r8, 1 ; POST i.e. r8 += 1
25        br loop
26 endloop:
27     movia r11, sum
28     sth r12, 0(r11) ; write the sum to memory

```

A similar approach can be taken for while loops.

```

1  .text
2
3  add      r8, r0, r0 ; zero out r8
4  movia    r9, n ; assumes n >= 1
5  ldh      r9, 0(r9) ; loads condition
6
7  doloop:
8      → address of arr[0]
9
10     → grab addr of arr[i]
11     → load arr[i] into r12
12     → add arr[i] to sum
13
14     addi r8, r8, 1 ; increment i
15     blt r8, r9, doloop ; evaluate loop condition
16 endloop:
17     movia r11, sum
18     sth r12, 0(r11) ; write the sum into memory

```

Comment

gcc is not a compiler by itself: it actually orchestrates and calls out a lot of other things

1. **cpp**: the C preprocessor,  $f.c \rightarrow f.i$ ; dealing with  $;;$ , **defines**, etc. Inspect via `gcc -E`
2. **gcc -s f.i**: the **cc**, the c compiler  $f.i \rightarrow f.s$ : parse pure c from preprocessor and spew out assembly code
3. **as**: the assembler,  $f.s \rightarrow f.o$ : to turn assembly to machine code, creating objects ready for linkage
4. **ld**: the linker,  $f.o \rightarrow f$ : link together all the objects into a single executable. Looks through **LDPATH** to look for symbols to link.<sup>10</sup>

SUBSECTION 7.3

## Lecture 8: Subroutines

A subroutine (or how we more commonly understand them, function calls), is a way to break up a program into smaller pieces and is a core part of structured programming.

Let's look at how we can implement the following in assembly

```

1  int add3(int a, int b, int c){
2      return a+b+c;
3  }

```

First, for c subroutines to work as intended they must:

1. Be callable from anywhere in the program

<sup>10</sup> There is static and dynamic linking; static meaning that everything is inside. These days most are dynamically linked which can grab symbols from shared libraries (.so) right before/during runtime

2. Be able to pass unique parameters across different subroutine invocations
3. Be able to return a value to the caller
4. Must be able to change control flow such that it goes back to the point where it was called

This leads to a number of questions:

1. How does the subroutine return to the caller?
2. How does it return a value?
3. How do we pass arguments?
4. What about local memory?
5. What happens to registers when we call a subroutine?

These issues are addressed through a set of implementation-specific (though largely universal) rules that all valid subroutines must follow. These are called **calling conventions**.

#### Definition 14

A key concept in subroutine calling is the **stack frame**. As for how stacks work/are implemented that is a google search away.

A stack has the following operations defined:

1. push: put a value on the top of the stack
2. pop: remove the top value from the stack
3. peek (distance): look at the value at a certain distance from the top of the stack

The NIOS II stores its stack pointer, or the address to the top of the stack, in r27, i.e. the stack pointer **sp**. The **sp** starts at the top of the stack (highest address value) and then the stack grows downwards to lower addresses.<sup>11</sup> There is no way to represent an empty stack in NIOS II; r27 will always contain a valid pointer address.<sup>12</sup>

#### Definition 15

##### push

```

1      ;; push r9 -> stack
2      subi sp, sp, 4 ;; grow stack by a long word
3      ;; (recall: grow downwards)
4      stw r9, 0(sp) ;; save value of r9 to sp

```

##### pop

```

1      ;; pop top of stack and return to r9
2      ldw r9, 0(sp) ;; read top value
3      addi sp, sp, 4 ;; increment sp; remove top elem

```

##### top

<sup>11</sup> Conversely the **heap** (commonly used for dynamic memory allocation) grows upwards towards higher addresses. The stack and heap are commonly implemented such that they grow towards each other

<sup>12</sup> An element is in the stack if its address is greater than **sp**

```

1 // access ith elem, assuming index in r9
2 add r9, r9, r9 ;; assume r9 contains nedex
3 add r9, r9, r9 // r9 = 4 * i
4 add r9, sp, r9 ;; sp + 4*i
5 ldw r9, 0(r9) // return value of ith element into r9

```

### Calling a subroutine and having it return to the caller

The stack is used to implement this behaviour. If a function will be calling another it has to save the `ra` value on the stack in the beginning and restore it from the stack prior to returning. Consider the following pseudo C code

```

1 boo_calls(){
2     coo();
3     doo();
4     return;
5 }
6
7 coo(){
8     doo();
9     return;
10 }
11 doo(){
12     return;
13 }

```

The NIOS II code would be as follows

`r31` is commonly used as the return address pointer and is aliased as `ra`

Note that `ret` transfers execution to the address in `ra`

```

1      .text
2 boo:  ;; boo will be making calls, so it first pushes the ra
3      ← value on the stack
4      subi sp, sp, 4
5      stw ra,0(sp) ;; push the return address onto the stack
6
7      call coo ;; resume execution at coo, ra = PC + 4 =
8      ← boo_ret1
9 boo_ret1:
10     call doo ;; continue execution at doo, ra = PC + 4 =
11     ← boo_ret2
12
13 boo_ret2:
14     ldw ra, 0(sp) ;; pop return address from the stack
15     addi sp, sp, 4
16     ret ;; resume execution at ra, which would be the
17     ← return address from the
18     ;; stack we just popped
19 coo:
20     subi sp, sp, 4
21     stw ra,0(sp) ;; push the return address onto the
22     ← stack
23     call doo ;; resume execution at coo, ra = PC + 4 =
24     ← coo_ret
25
26 coo_ret:
27     ldw ra, 0(sp) ;; pop return address of boo from the
28     ← stack
29     addi sp, sp, 4
30     ret ;; resume execution there
31
32 doo: ;; doo will not be making any calls, no need to save ra
33      ← on the stack
34      ret ;; just return to whoever called

```



**Figure 12.** The call instruction accepts a label as the 2nd argument which is encoded as a 26 bit immediate. This limits the called function to be within 256 MB of the caller; the actual target is the immediate multiplied by 4 and then concatenating with the program counter, i.e.  $PC_{31\ldots28} = IMM26 * 4$

Note: we can look at executables with `objdump -d`.

Example:

```
1
2 #include <stdio.h>
3 int main () {
4     printf("hello world");
5 }
6
```

```

1
2 hello_world:      file format elf64-x86-64
3
4
5 Disassembly of section .init:
6
7 0000000000001000 <_init>:
8 1000: f3 0f 1e fa          endbr64
9 1004: 48 83 ec 08          sub    $0x8,%rsp
10 1008: 48 8b 05 c1 2f 00 00  mov    %rax,%rcx
11    ↳ 0x2fc1(%rip),%rax      # 3fd0 <_gmon_start__@Base>
12 100f: 48 85 c0          test   %rax,%rax
13 1012: 74 02          je    1016
14    ↳ <_init+0x16>
15 1014: ff d0          call   *%rax
16 1016: 48 83 c4 08          add    $0x8,%rsp
17 101a: c3          ret
18
19 Disassembly of section .plt:
20
21 0000000000001020 <printf@plt-0x10>:
22 1020: ff 35 ca 2f 00 00  push
23    ↳ 0x2fc0(%rip)      # 3ff0 <_GLOBAL_OFFSET_TABLE_+0x8>
24 1026: ff 25 cc 2f 00 00  jmp
25    ↳ *0x2fcc(%rip)      # 3ff8 <_GLOBAL_OFFSET_TABLE_+0x10>
26 102c: 0f 1f 40 00          nopl   0x0(%rax)
27
28 0000000000001030 <printf@plt>:
29 1030: ff 25 ca 2f 00 00  jmp
30    ↳ *0x2fc0(%rip)      # 4000 <printf@GLIBC_2.2.5>
31 1036: 68 00 00 00 00 00  push   $0x0
32 103b: e9 e0 ff ff ff          jmp    1020
33    ↳ <_init+0x20>
34
35 Disassembly of section .text:
36
37 0000000000001040 <_start>:
38 1040: f3 0f 1e fa          endbr64
39 1044: 31 ed          xor    %ebp,%ebp
40 1046: 49 89 d1          mov    %rdx,%r9
41 1049: 5e          pop    %rsi
42 104a: 48 89 e2          mov    %rsp,%rdx
43 104d: 48 83 e4 f0          and    %rdx,%rdx
44    ↳ $0xfffffffffffffff0,%rsp
45
46 1051: 50          push   %rax
47 1052: 54          push   %rsp
48 1053: 45 31 c0          xor    %r8d,%r8d
49 1056: 31 c9          xor    %ecx,%ecx
50 1058: 48 8d 3d da 00 00 00  lea
51    ↳ 0xda(%rip),%rdi      # 1139 <main>
52 105f: ff 15 5b 2f 00 00  call
53    ↳ *0x2f5b(%rip)      # 3fc0 <__libc_start_main@GLIBC_2.34>
54 1065: f4          hlt
55 1066: 66 2e 0f 1f 84 00 00  cs    nopw
56    ↳ 0x0(%rax,%rax,1)
57 106d: 00 00 00
58
59    ;;; and this goes on for a while loading in the
60    ↳ .so & printf

```

And more assembly later ...

```

1 000000000000001139 <main>:
2 1139: 55          push    %rbp
3 113a: 48 89 e5    mov     %rsp,%rbp
4 113d: 48 8d 05 c0 0e 00 00  lea
5           0x0(%rip),%rax    # 2004 <_IO_stdin_used+0x4>
6 1144: 48 89 c7    mov     %rax,%rdi
7 1147: b8 00 00 00 00  mov     $0x0,%eax
8 114c: e8 df fe ff ff  call    1030
9           <printf@plt>
10 1151: b8 00 00 00 00  mov     $0x0,%eax
11 1156: 5d          pop     %rbp
12 1157: c3          ret
13
14 Disassembly of section .fini:
15 000000000000001158 <_fini>:
16 1158: f3 0f 1e fa  endbr64
17 115c: 48 83 ec 08  sub    $0x8,%rsp
18 1160: 48 83 c4 08  add    $0x8,%rsp
19 1164: c3          ret
20
21

```

#### SUBSECTION 7.4

## Lecture 9

We've seen how to call and return from subroutines – but what about passing arguments and returning values? For this lecture we'll assume that only words are returned from and passed to subroutines, but other data types and structs can be used as well.

- Return value is passed in  $r2^{13}$
- First four parameters are passed in  $r4$ ,  $r5$ ,  $r6$ ,  $r7$
- Additional parameters are pushed onto the stack *in order*
  - We push the last argument to the stack first and so forth such that the first non-register argument (the 5th one) is the first to be popped off once we enter the function.

Consider a function which takes seven integers and adds them together.

The following calling convention is the one used by gcc for the NIOS II family

<sup>13</sup> only one return value can be given; multiple can be encoded via structures or ptrs etc

```

1          .data
2  sum: .word 0
3          .text
4  main:
5
6  addi sp, sp, -4
7  stw ra, 0(sp);
8  ;; return address ptr (ra) is pushed onto the stack
9
10 ;; fill first 4 args
11 movi r4, 1
12 movi r5, 2
13 movi r6, 3
14 movi r7, 4
15
16 ;; allocated for arguments 5-7
17 addi sp, sp, -12 ;; 3 words * 4 byte
18
19
20 ;; push arguments 5-7 onto the stack
21 ;; note order; <top> 5, 6, 7
22 movi r2, 7
23 stw r2, 8(sp)
24
25 movi r2, 6
26 stw r2, 4(sp)
27
28 movi r3, 5
29 stw r2, 0(sp)
30
31 call add7
32
33 add7:
34     add r2, r4, r5      # add the first two arguments and
35     ← place the sum into r2
36     add r2, r2, r6      # add the third argument to r2
37     add r2, r2, r7      # add the fourth argument to r2
38     ldw r7, 0(sp)       # read the fifth argument from the
39     ← stack
40     add r2, r2, r7      # add to r2
41     ldw r7, 4(sp)       # read the sixth argument from the
42     ← stack
43     add r2, r2, r7      # add to r2
44     ldw r7, 8(sp)       # read the seventh argument from the
45     ← stack
46     add r2, r2, r7      # add to r2 (return value in r2 by
47     ← convention)
48     ret
49
50 ;; and then do the cleanup etc

```

Note that in reality `gcc` will actually preallocate all the memory needed by the function

before the function is called. So instead of allocating 12 bytes (3x 1 word arguments) as it does on line 17, it will actually allocate 16 bytes because we need to store the return address.

The allocated stack frame for the function will be the largest of any function being called from it.

For example,

```

1 int main(){
2     foo(1,2,3);
3     boo(1,2,3,4,5,6,7,8);
4 }
```

Boo has the maximum number of arguments, so we'll need to allocate  $8 - 4 = 4$  words on the stack for the arguments and the return address  $- 5 * 4 = 20$  bytes. This results in

```

1 ;; prologue
2 addi sp, sp, -20
3 stw, ra, 16($0) ; note little-endian and remaining 16 bytes
4     ← for 4 argument words
5
6 ;; epilogue
7 ldw ra, 16(sp) // pop return addr from stack
8 addi sp, sp, 20;
```

#### SUBSECTION 7.5

## Lecture 10: Recursive Subroutines

---

Consider this code block that computes the Ackerman function

```

1 int Ackerman(unsigned int x, unsigned int y)
2 {
3     if (x==0) return y+1;
4     if (y==0) return Ackerman(x-1, 1);
5     return Ackerman(x-1, Ackerman(x, y-1));
6 }
```

This function is interesting to implement because of its recursive nature.  
Breaking it down a little more,

```

1 int Ackerman(unsigned int x, unsigned int y)
2 {
3     if (x==0) return y+1;
4     if (y==0) return Ackerman(x-1, 1);
5     int tmp = Ackerman(x, y-1)
6     return Ackerman(x-1, tmp);
7 }
```

The return address and the value of `x` must be stored on the stack, so we will need space for 2 words.

```

1      .text
2 Ackerman:
3      addi sp, sp, -8
4      stw ra, 4(sp)
5
6      bne r4, r0, Xnot0;
7 Xis0:
8      addi r2, r5, 1
9      br epilogue
10 Xnot0:
11     bne r5, r0, Ynot0
12 Yis0:
13     ; pass arguments
14     addi r4, r4, -1 ;First one is x-1, i.e. r4-1
15     addi r5, r0, 1 ; second is 1
16     call Ackerman
17     br epilogue
18 Ynot0:
19     stw r4, 0(sp) ; preserve value of x on the stack
20     ; x is already at the right place for fn call
21     addi r5, r5, -1 ; decrement y
22     add r5, r5, 0
23     call Ackerman
24 epilogue:
25     ldw ra, 4(s0)
26     addi sp, sp, 8
27     ret

```

## SUBSECTION 7.6

## Lecture 11: Structs and recursive structures

---

Consider a binary tree with a left and right child and a value. We can represent this as a struct in C.

```

1 struct node{
2     int value;
3     struct node *left;
4     struct node *right;
5 };

```

The memory layout of the struct is word-aligned and *exactly* like that of the struct definition. In this case `value` lives at `addr+0`, `left` at `addr+4`, and `right` at `addr+8`.

Now, consider the following recursive function to perform binary search on a BST

When this gets compiled note that in memory the struct will be word-aligned, i.e. on a 4-byte word machine the beginning of the struct will be at an address which is a multiple of the word size

```
1 int findv(int d, struct node) {
2     struct node * m;
3     m = root;
4     if (root == NULL) {
5         return 0;
6     }
7     if (root->val == d) {
8         return 1;
9     }
10    if (d <= root->v) {
11        return findv(d, root->left);
12    }
13    else {
14        return findv(d, root->right);
15    }
16    // Note that this is a tail-recursive function, i.e.
17    // nothing is done after the recursive call
18    // some compilers may unfurl this into a loop
19 }
```

How can we represent this in assembly?

```

1 // int d is in r4, root is in r5 (by convention)
2         .text
3
4 findv:
5     addi sp, sp, -4
6     stw R4, 0(sp);
7 isrootnull:
8     bne r5, r0, isdv
9 rootisnull:
10    movi r2, 0
11    br epilogue
12 isdv:
13    ldw r2, 0(r5) ; r2 <- root->val
14    bne r2, r4, tryagain
15 found:
16    mov r2, 1
17    br epilogue
18 tryagain:
19    ble r4, r2, goleft
20 goleft:
21    ldw r5, 4(r5) ; offset struct addr pointer to right
22    → member
23    call findv
24    br end
25 goright:
26    ldw r5, 8(r5) ; offset struct addr pointer to right
27    → member
28    call findv
29    br end
30 notfound:
31    add r2, r0, r0
32 epilogue:
33    ldw ra, 0(sp)
34    addi sp, sp, +4
35    ret

```

Another example of a recursive function is the following routine which detects whether a string is a palindrome.

Note that strings in C are represented as null-terminated array of characters

```
1 char s[] = "abba";
```

Which in assembly looks like:

.datas : .byte'a','b','c','a',0;; Or, can write with double quotes i.e. string literals : .string"abba"  
(7.1)

The C implementation of this function is as follows:

```
1 int palindrome(char *a)
2 {
3     char * e;
4     if (a == NULL) return 1; // empty string is a
5     → palindrome
6     e = a;
7     while (*e != 0) e++; // find end of string
8     if (e == a) return 1; // string of length 1 is a
9     → palindrome
10    e--;
11    while ((*a != 0) && (*a == *e)) {
12        a++;
13        e--;
14    }
15    if (*a == 0) return 1; // is a palindrome
16    return 0;
17 }
```

In assembly this would be as follows:

As an example this palindrome function can be called as follows

```
1           .data
2           s: .string "abba"
3
4           .text
5           .global main
6
7           main:
8           movia r4, s
9           call palindrome
```

## SECTION 8

# GPIO

---

## SUBSECTION 8.1

## Lecture 12: Devices

---

A computer is great and all, but for it to be useful it must communicate with the outside world via devices. A simple device interface is the Parallel Port Interface, a common implementation of which is the GPIO (General Purpose Input/Output) interface<sup>14</sup>

There are two registers used to communicate with the NIOSII GPIO interface which live in memory;

- `dr` (data register) at `0xFF200060`
- `dir` (data register) at `0xFF200064`

`dr` is the data word to communicate in and out of the GPIO interface, and `dir` denotes if the signal is an input or an output. The PIT provides 32 bits of data which can be independently mapped to be input or output depending on the value in DIR. If bit 0 of DIR is 1, then D0 becomes an output. And so forth for the rest of the bits. Note that the `stwio` and `ldwio` variants of the `stw` and `ldw` instructions must be when interfacing with IO devices

<sup>14</sup> Of which the NIOS II board has two

The NIOSII has two GPIO interfaces. The other, GPIO2, is at `0xFF200070`

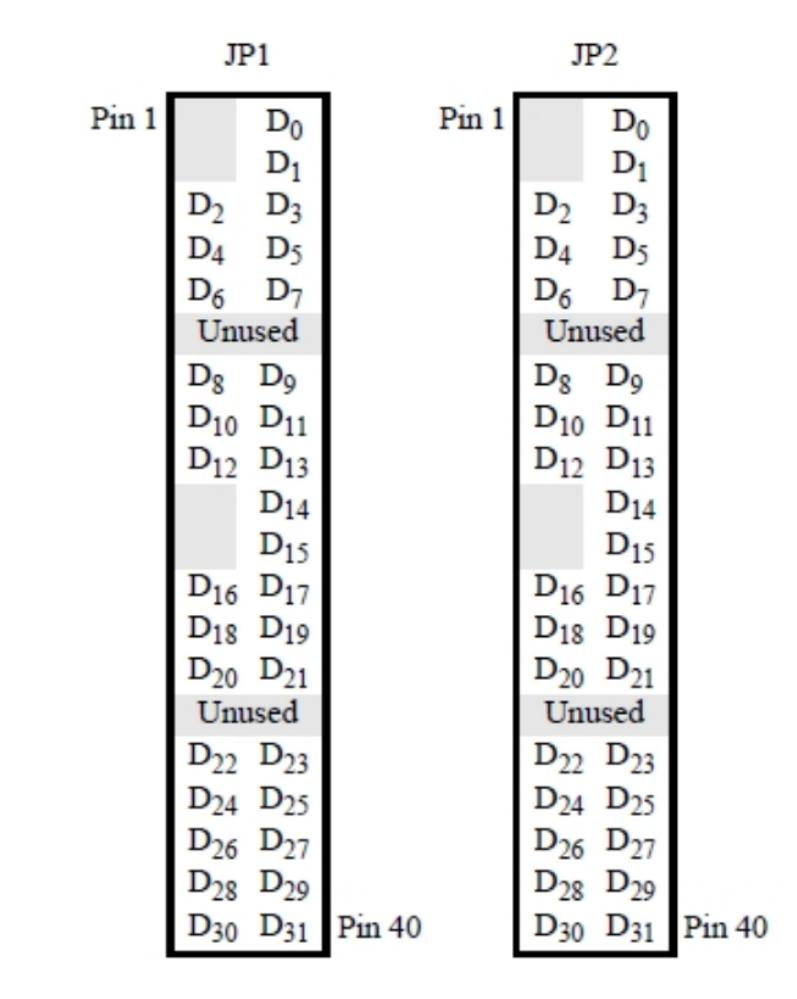


Figure 13. GPIO pinout

Two exercises we will cover include building a thermostat with the GPIO pins and building a keyboard.

```

1  ;; configure all pins as outputs
2  addi r8, r0, 0xFFFFFFFF
3  movia r9, 0xFF200064
4  stwio r8, 0(r9)
5
6  ;; configure pins D0-D3 as outputs, D4-D31 as inputs
7  addi r8, r0, 0x0000000F
8  movia r9, 0xFF200064
9  stwio r8, 0(r9)

```

Given that  $D_0$  is connected to the thermostat's output and  $D_3$  is connected to the heat fan motor,

A write to an input pin will be silently ignored.

```

1  PIT1_BASE equ, 0xFF200060
2  PIT1_DR_OFFSET equ, 0x0
3  PIT1_DIR_OFFSET equ, 0x4
4
5      .text
6  heat:
7      movia r9, PIT1_BASE
8      addi r8, r0, 8
9      stwio PIT1_DIR_OFFSET(r9) ;; configure all pins but d3
10     ← as inputs (0x00000008)
11     stwio r0, PIT1_DR_OFFSET(r9) ;; turn off the motor
12  fever:
13      ldwio r8, PIT1_DR_OFFSET(r9) ;; read all pins
14      andi r8, r8, 0x1 ;; mask off all but d0
15      beq r8, r0, fanon ;; turn fan on because it's cold (d0
16      ← == 9)
17  fanoff:
18      stwio r0, PIT1_DR_OFFSET(r9) ;; turn off the motor
19      br fever
20  fanon:
21      addi r8, 8
22      stwio r8, PIT1_DR_OFFSET(r9) ;; turn on the motor
23      br fever

```

Something similar can be implemented in c

In the lecture notes he uses 0x00000004 but I think that's a typo.

```

1
2  #define DR ((unsigned int*) 0xFF200060)
3  #define DIR ((unsigned int*) 0xFF200064)
4
5  void heat(void)
6  {
7      unsigned int t;
8      *DIR = 0x8;
9      *DR = 0x0;
10     while (1)
11     {
12         t = *DR;
13         if (t & 0x1)
14             *DR = 0x0;
15         else
16             *DR = 0x8;
17     }
18 }

```

A nicer way that this can be implemented is to use a struct to represent the PIT

```

1
2 struct PIT_t {
3     unsigned int DR;
4     unsigned int DIR;
5 };
6
7 struct PIT_t *pitp = (struct PIT_t *) 0xFF200060;
8
9 void heat(void)
10 {
11     unsigned int t;
12     pitp->DIR = 0x8;
13     pitp->DR = 0x0;
14     while (1)
15     {
16         t = pitp->DR;
17         if (t & 0x1)
18             pitp->DR = 0x0;
19         else
20             pitp->DR = 0x8;
21     }
22 }

```

This chunk of code exploits the fact that the DR element appears first and is immediately followed by DIR. This allows us to elegantly map the fields of the struct to the actual memory layout (line 7)

Another example let's consider debouncing the metal switch used inside the thermostat. Whenever metal contacts are used there is a short period of time where the switch will rapidly change state. This may not be what the user intends and as such we'd like to debounce the values to make it more sane. To avoid this we take a number of samples and assume that the value read is 1 only if the number of samples that was 1 is greater than  $\frac{N}{2}$

```

1  PIT1_BASE equ, 0xFF200060
2  PIT1_DR_OFFSET equ, 0x0
3  PIT1_DIR_OFFSET equ, 0x4
4
5      .text
6  heat:
7      movia r9, PIT1_BASE
8      addi r8, r0, 8
9      stwio PIT1_DIR_OFFSET(r9) ;; configure all pins but d3
10     ← as inputs (0x00000008)
11     stwio r0, PIT1_DR_OFFSET(r9) ;; turn off the motor
12
13 fever:
14     addi r10, r0, 1000 ;; take 1000 samples
15     add r11, r0, r0 ;; count of samples that are 1
16
17 sampling:
18     ldwio r8, PIT1_DR_OFFSET(r9) ;; read all pins
19     andi r8, 0x1 ;; mask out r0 (thermostat)
20     add r11, r11, r8 ;; increment count if r8 is 1
21     subi r10, r10, 1 ;; decrement sample count
22     bne r10, r0, sampling ;; loop if we still have samples
23     ← to take
24
25     addi r10, r10, 500 ;; check if we have more than 500
26     ← samples that are 1
27     bgt r11, r10, fanon
28
29 fanoff:
30     stwio r0, PIT1_DR_OFFSET(r9) ;; turn off the motor
31     br fever
32
33 fanon:
34     addi r8, 8
35     stwio r8, PIT1_DR_OFFSET(r9) ;; turn on the motor
36     br fever

```

### 8.1.1 The PIT implementation

This example is a PIT with 8 connections with *DR* at 0xFF1110 and *DIR* at 0xFF1114. A 32 bit PIT should follow.

At a physical level the PIT presents the following signals

1. Data: 8 wires that supply the value to be written to or read from the PIT
2. DIR write signal: when this is 1 the value in the data wires is written to the DIR
3. DR write signal: when this is 1 the value in the data wires is written to the DR
4. DR read signal: when this is 1 the data signals are placed into output mode and take on values that currently appear on external port connections
5. External port: 8 wires that connect to the external world

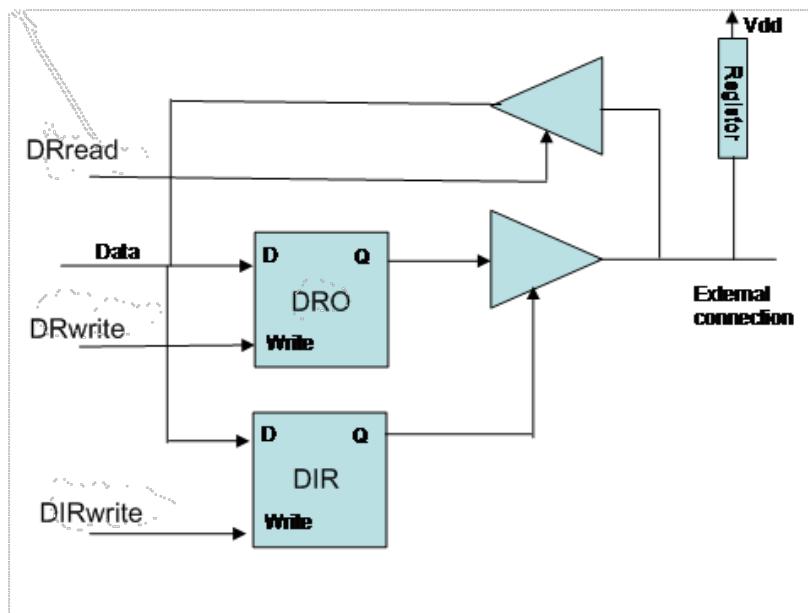


Figure 14. One PIT bit

The DRO and DIR boxes are latches whose values change using the Data and Write signals. The triangle looking things are **tri-state buffers**.

- DIR = 1 -> tri-state latch is closed -> passes value of DRO to external connection
- DIR = 0 -> tri-state latch is open -> external source determines value on the connection
  - resistor helps drive high voltage from external source
- DRread = 1 -> opens upper tristate buffer, puts external value into Data line

Eight of these can then be chained together to form a 8 bit PIT.

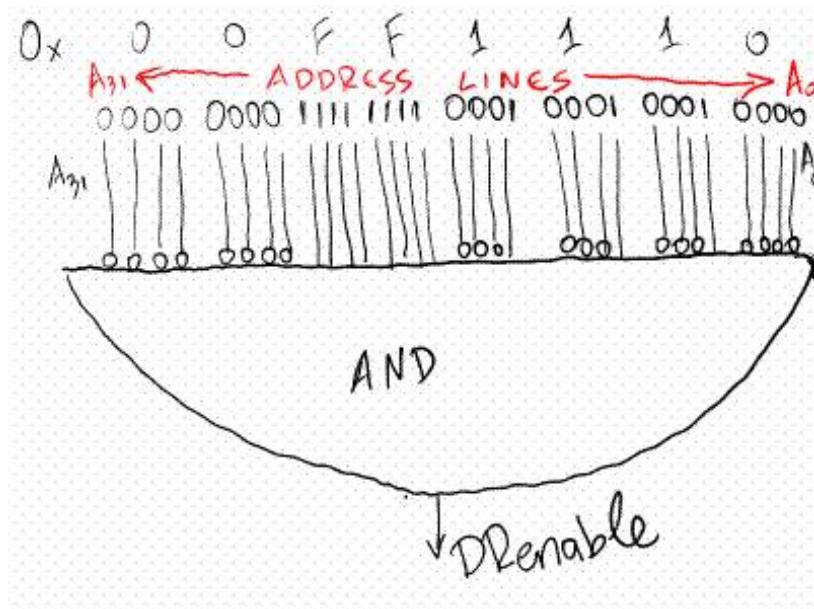
### 8.1.2 Memory

Our memory needs to meet these requirements

1. Load/store operations
2. Byte, halfword, or word types
3. Addr of 32 bits
4. data value of at most 32 bits for read/write values
5. A do-nothing signal

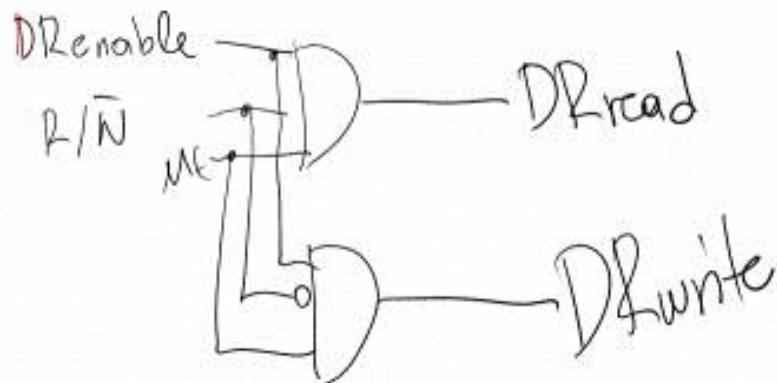
1) can be implemented with a single read-write signal. 2)) can be done with 2 signals. Let's use 00 for byte, 01 for half-word, and 11 for long-word. 10 is not used. 3) and 4) need 32 signals each, and 5) requires a master-enable signal. This is useful to prevent the memory from taking on transient values.

Now comes the task of connecting the PIT to the memory interface. First we connect the PIT data lines to the lower 8 bits of the memory interface. So DIR, DRO, DRI can be accessed via load/store operations



**Figure 15.** We'll also need to check if the address being accessed is the address of the PIT. The easiest way to do this is to just use an AND across the address lines and pass that signal to DREnable

DRenable can now be combined with R/W to form a single signal that generates the DR-read and DRwrite signals



Something similar can be done for DIRwrite.

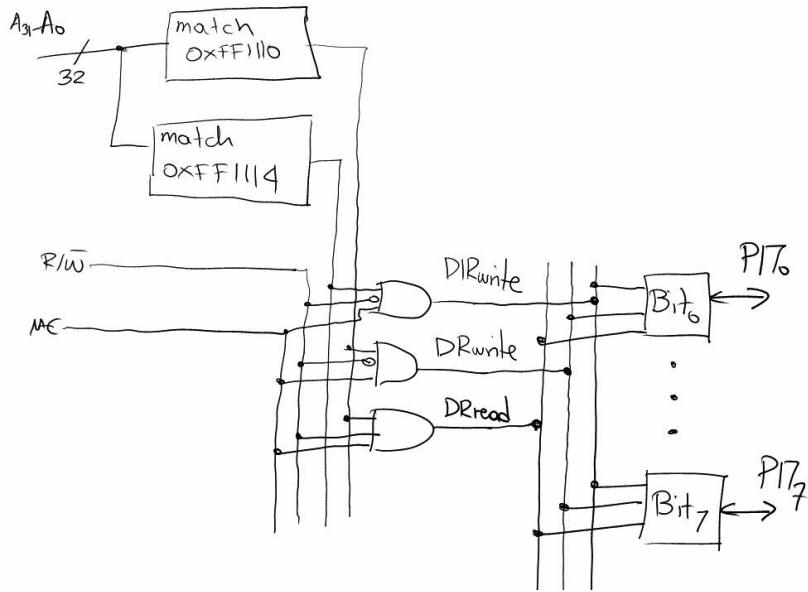


Figure 16. The complete design

#### SUBSECTION 8.2

## Lecture 12: UART

The (JTAG) UART device is a UART implemented over the JTAG interface, which for the NIOSII is implemented over USB. UART stands for Universal Asynchronous Receiver Transmitter. It is a communication protocol which allows for non-instantaneous communication between two devices.

- Starts at address 0xFF201000
- Receiver Register RR at an offset of 0
- Transmitter Register at an offset of 0 (same as RR)
- Control and status register, CSR at an offset of 4

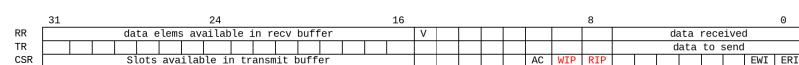


Figure 17. Register format

There are two transactions that can be asked of the UART: (1) send a character and (2) receive a character. They are implemented by placing bits into the data to send/receive into the transmitter/receiver register. However, in real life communication is more difficult than just that; sending characters takes considerably longer than the CPU's speed so we must be able to wait while the UART is sending a character before an attempt to send another one is made. Also, we should only read a character from the RR only if one has been received from the external source. This reading may also need to be buffered due to the variable nature of RR receive speed.

### sending a character

Here's a subroutine for sending the character in r4.

```

1  .equ JTAG_UART_BASE, 0xFF201000
2  .equ JTAG_UART_RR, 0
3  .equ JTAG_UART_TR, 0
4  .equ JTAG_UART_CSR, 4
5
6      .text
7  putchar:
8      movia r8, JTAG_UART_BASE
9  wait:
10     ldwio r2, JTAG_UART_CSR(r8) # read csr
11     srl r2, r2, 16 # keep only upper 16 (shift right
12     ← logical immediate)
13     ;; The csr's bits 16-31 contains a number which
14     ;; is the number of slots in the queue available
15     ;; to accept requests. If the number is 0, then
16     ;; the queue is full and we must wait.
17     beq r2, r0, wait # wait until upper 16 bits are nonzero
18
19     stwio r4, JTAG_UART_TR(r8) # place it into the FIFO
20     ret
21
22
23

```

### receiving a character

To receive a character we must wait until a character is received. This information is in RR

- 0-7 contain the character, if one has been received
- 15 is 1 if a character has been received. If it is 0, then no character has been received and 0-7 are meaningless
- 31-16 contain additional information e.g. number of additional characters received and are waiting in the incoming queue.

```
1 .equ JTAG_UART_BASE, 0xFF201000
2 .equ JTAG_UART_RR, 0
3 .equ JTAG_UART_TR, 0
4 .equ JTAG_UART_CSR, 4
5
6     .text
7 getchar:
8     movia r8, JTAG_UART_BASE
9
10    wait:
11        ldwio r2, JTAG_UART_RR(r8) ;; read RR in r2
12        andi r10, r2, 0x8000 ;; extract bit 15
13        beq r10, r0, wait ;; wait until bit 15 is 1
14        andi r2, r2, 0xff ;; keep only character (bit 0-7) in
15        r2
16
17
18
```

Both of these subroutines busy waits are used. This is slow and inefficient. A better way is to use interrupts, which will be covered in a later lecture

At the communication link level the device represents the characters as a stream of bits. Each bit is communicated by setting the line to the corresponding voltage level for a pre-specified duration; this is the bit-cell shown.

### Definition 16

**baud rate** : number of bit cells within one second. For example 9600 baud means a bit time of  $\frac{1}{9600} = 104.16$  microseconds, so it would take 1.0416 milliseconds at least to send a full byte. Note that each byte has a start and stop bit so a byte takes 10 bit cells to transmit.

Ideally the transmitter and receiver would use identical time references for the bit cells, which would make life easy because it would mean the receiver can simply just take a single sample at the center of each bit cell to reconstruct the byte. However, life is not so easy and real-life devices have phase differences.

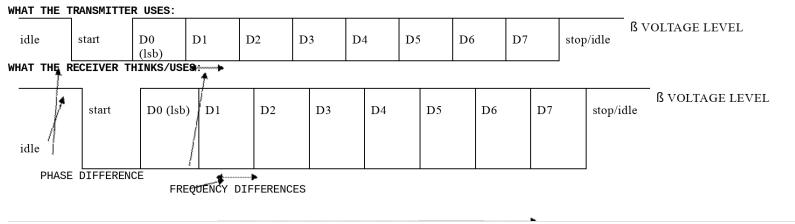
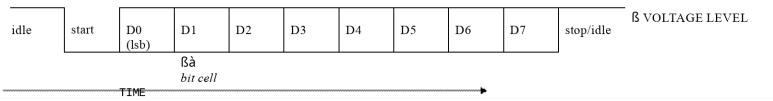


Figure 18. Phase differences

15

The common solution for this problem is to use over-sampling, i.e. taking several samples to detect the 0->1 transition for the stop/idle to start bits, then making carefully chosen single samples to fall at the center of bit time. For a receiver that takes 16 samples per bit cell, generally the  $i$ th sample should be taken at  $(24 + i * 16)$  cycles, i.e. pass 16 samples to get past start bit and then another 8 to go to the middle of the first bit cell and so forth.

Even with these measures communication errors<sup>16</sup> can still occur. Further reduction of errors can be achieved by using an additional parity bit which can be used to detect single errors.



<sup>15</sup> In practice there shouldn't be a difference of more than 20% between the stop bit and where the receiver thinks the stop bit is. This is because of RS-232C standard imposing a requirement imposing a maximum of 2% difference between device baud rates, and the fact that there are start/stop bits which allow for the receiver to sync to sender time.

<sup>16</sup> FRAME Errors

#### SUBSECTION 8.3

## Lecture 13: Interrupts & UART

In the previous lecture we used busy polling to communicate with the serial device, i.e. probing the status register until the desired condition was met. Not only is this a wasteful use of device resources, it is also difficult to scale to multitasking scenarios.

At a high level, interrupts work as follows:

1. The processor is executing instructions
2. A device requests an interrupt
3. The processor decides to grant that request, i.e. finishing current instruction and then inducing a call to a special interrupt handler routine
4. Program flow is restored to previous instruction

Therefore there are a few requirements that must be met for interrupts to work

1. Devices must be able to request interrupts
2. Must be able to signal to device that its interrupt has been handled
3. Must have a way to map interrupts to the correct interrupt handler
4. Need to know where the interrupt handler lives in memory
5. Must be able to save and restore state affected by an interrupt call

Requirements 1 and 2 are handled by having two directed wires between the CPI and the device through which request and response 'handshake's can be implemented. The NIOS II provides 32 interrupt request wires IRQ0 to IRQ31, and notifying a device is done in software

i.e. writing to a specific location in memory. Exact process is device-specific. Requirements 3 and 4 i.e. inducing an interrupt handler. One way that this can be done is to have a subroutine at a hard-coded address for all interrupts which then queries which device requested the interrupt and then inducing the call to the appropriate handler. This is generally implemented as some sort of lookup table. The NIOS II implements this by having all interrupt requests start executing code at `0x00000020`; the `.section exceptions, "ax"` directive can be used to assign code to that part of memory. That code must interrogate all devices to figure out which devices are requesting interrupts and then fire off the appropriate ISRs. Requirement 5 can be met by just pushing everything relevant onto the stack.

Definition 17

- **ctl0**: Master interrupt enable switch. Bit 0 is 1 means interrupts will be serviced; they will be ignored otherwise.
- **ctl1**: Preserves current state of **ctl0** upon accepting an interrupt
- **ctl3**: Enable/disable specific IRQ lines. Interrupts are accepted for  $\text{IRQ}_i$  if bit  $i$  of **ctl3** is 1 and bit 0 of **ctl0** is 1

Reading and writing from control registers is done through two specialized instructions; `rdctl ctrlx, ry`: reads value of a control register `ctrlx` and stores in `ry`, and `wrctl ctrlx, ry` which writes the value of `ry` into `ctrlx`. Also, `r29` also participates by storing the PC of the instruction following that of the interrupt and commonly is aliased to be `ea`, or exception address.

The more detailed sequences is therefore as follows

- Device assert an IRQ line to request an interrupt
- Processor finishes current instruction
- Processor saves the current interrupt enable bit (storing **ctl0** into **ctl1**), then disables **ctl0** to disable further interrupts
- Set  $ea = PC + 4$ ; the PC immediately following the instruction that had just been executed
- Set  $PC = 0x20$  so that execution continues in interrupt handler
- Handler terminates with `eret` which will restore **ctl0** from **ctl1** and PC from `ea`

PART

III

# *ECE355: Signal Analysis and Communication*

SECTION 9

Taught by Prof. Sunila Akbar

## **Admin and Preliminary**

SUBSECTION 9.1

### **Lecture 1**

- CT and DT signals

- A ton of LTI (Linear time invariant) systems
- Processing of signals via LTI systems
- Fourier transforms
- Sampling

### 9.1.1 Mark Breakdown

**Table 1.** Mark Breakdown

Homework	20
MT1	20
MT2	20
Final	40

- Continuous enclose in  $(\cdot)$ , independent is  $t$
- Discrete: enclose in  $[\cdot]$ , independent is  $n$

Theorem 2

### Energy for Complex Signals

$$E_{[t_1, t_2]} = \int_{t_1}^{t_2} |x(t)|^2 dt \quad (9.1)$$

$$E_{[t_1, t_2]} = \sum_{n=n_1}^{n_2} |x(n)|^2 \quad (9.2)$$

### Average Power for Complex Signals

$$P_{avg, [t_1, t_2]} = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} |x(t)|^2 dt \quad (9.3)$$

$$P_{avg, [t_1, t_2]} = \frac{1}{n_2 - n_1 + 1} \sum_{n=n_1}^{n_2} |x(n)|^2 \quad (9.4)$$

In many systems we are interested in power and energy of signals over an infinite time interval;  $-\infty < \{t, n\} < \infty$

## SECTION 10

# Transformations

## SUBSECTION 10.1

### Lecture 2

Most of this lecture was review. When applying transforms just note to always scale, *then* shift, i.e.

1.  $y(t) = x(\alpha t)$
2.  $y(t) = x(\alpha t + \frac{\beta}{\alpha})$

**Definition 18** **Fundamental Period**

$$x_t = x(t + mT), m \in \mathbb{Z} \quad (10.1)$$

The fundamental period,  $T_o$  is the smallest positive value of  $T$  for which this holds true

**Definition 19** **Even signals**

$$x(t) = x(-t) \quad (10.2)$$

**Definition 20** **Odd signals**

$$x(t) = -x(-t) \quad (10.3)$$

**Theorem 3** Any signal can be broken into an even and odd component

$$\begin{aligned} x(t) &= Ev \{x(t)\} = \frac{1}{2} [x(t) + x(-t)] \\ x(t) &= Od \{x(t)\} = \frac{1}{2} [x(t) - x(-t)] \end{aligned} \quad (10.4)$$

## SUBSECTION 10.2

**Lecture 3**

Again, most of this lecture was review from the waves portion of PHY293 from last year or some other course prior.

A complex exponential and sinusoidal system can be represented as

$$x(t) = Ce^{at} \quad (10.5)$$

Where  $C, a$  are complex numbers.

Two cases may occur.

If  $a$  imaginary and  $C$  is real we have, depending on  $\omega$ , either a constant signal or a periodic sinusoidal system.

$$x(t) = e^{j\omega_0 t} \quad (10.6)$$

- Important property: this is periodic, i.e.  $Ce^{j\theta_0 t} = Ce^{j\theta_0(t+T)}$
- Implies that  $e^{j\omega_0 T} = 1$
- Implies that for  $\omega \neq 0 \rightarrow T_0 = \frac{2\pi}{|\omega_0|}$

On the other hand, if  $a$  imaginary and  $C$  complex, we have a periodic signal with  $T = \frac{2\pi}{\omega_0}$

$$x(t) = Ce^{j\omega_0 t} = |C|e^{j\omega_0 t + \phi} = |C| \cos(\omega_0 t + \phi) + j|C| \sin(\omega_0 t + \phi) \quad (10.7)$$

The energy of the signal is given by (9.1), or

$$E_{period} = \int_0^{T_0} |e^{j\omega_0 t}|^2 dt = \int_0^{T_0} 1 dt = T_0 \quad (10.8)$$

$$P_{period} = \frac{E_{period}}{T_0} = 1 \quad (10.9)$$

Recall: implication that  $e^{j\omega_0 T} = 1$ , therefore the quantity inside the integral evaluates to 1

### 10.2.1 General Continuous Complex Exponential Signals

The most general case of a complex exponential can be represented as a combination of the real exponential and the periodic complex exponential;

$$C = Ce^{at} \quad (10.10)$$

and

$$a = r + j\omega_0 \quad (10.11)$$

can be combined to give

**Definition 21**

$$Ce^{at} = |C|e^{rt}e^{j\omega_0 t + \theta} \quad (10.12)$$

Euler's relation can be used to simplify this to

$$Ce^{at} = |C|e^{rt} (\cos(\omega_0 t + \theta) + j \sin(\omega_0 t + \theta)) \quad (10.13)$$

By inspection we can see that the signal has the following properties:

1.  $r = 0$ : real and imaginary parts of sinusoidal
2.  $r > 0$ : sinusoidal signal with exponential growth
3.  $r < 0$ : sinusoidal signal with exponential decay

I will be skipping notes on the discrete case as it is essentially the same as the continuous case, but with the following differences

**Table 2.** Comparison of continuous and discrete complex exponential signals

$e^{j\omega_0 t}$	$e^{j\omega_0 n}$
Distinct signals for distinct $\omega_0$	identical signals for distinct $\omega_0 \in \{\omega_0 \pm 2\pi i, i \in \mathbb{Z}\}$
Periodic for any $\omega_0$	Periodic only if $\omega_0 = \frac{2\pi m}{N}$ for integers $N > 0, m$
Fundamental frequency $\omega_0$	Fundamental frequency $\frac{\omega_0}{m}$
Fundamental period $\omega_0 = 0 \rightarrow$ undefined, otherwise $T_0 = \frac{2\pi}{\omega_0}$	Fundamental period $\omega_0 = 0 \rightarrow$ undefined, otherwise $T_0 = m \frac{2\pi}{\omega_0}$
	Since unique $\omega$ does not mean unique signal, pick $0 \leq \omega_0 \leq 2\pi$ or $-\pi \leq \omega_0 \leq \pi$

SECTION 11

## Basic Signals

SUBSECTION 11.1

### Lecture 4: Step and Impulse Functions

One of the simplest discrete-time signals is the **unit impulse**<sup>17</sup> function,  $\delta[n]$

<sup>17</sup> or unit sample

Definition 22

$$\delta[n] = \begin{cases} 0 & n \neq 0 \\ 1 & n = 0 \end{cases} \quad (11.1)$$

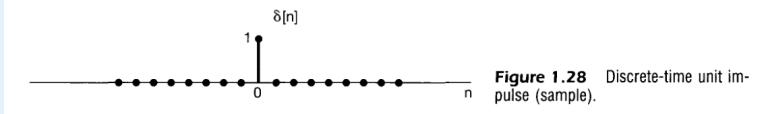


Figure 1.28 Discrete-time unit impulse (sample).

Another basic signal is the **unit step** function,  $u[n]$

Definition 23

$$u[n] = \begin{cases} 0 & n < 0 \\ 1 & n \geq 0 \end{cases} \quad (11.2)$$

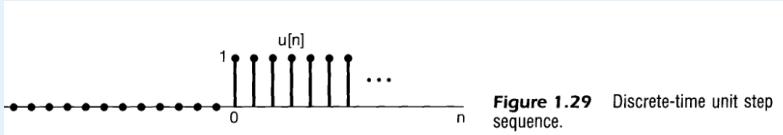


Figure 1.29 Discrete-time unit step sequence.

The unit impulse function is the first difference of the discrete time step function, i.e.

$$\delta[n] = u[n] - u[n - 1] \quad (11.3)$$

And the unit step function is the running sum of the unit impulse function, i.e.

$$u[n] = \sum_{m=-\infty}^n \delta[m] \quad (11.4)$$

This can be rewritten with  $k = n - m$  to make a more convenient expression for moving the function along  $-\infty \dots 0 \dots \infty$

$$u[n] = \sum_{k=0}^{\infty} \delta[n - k] \quad (11.5)$$

Theorem 4

The unit impulse function  $\delta[n - n_0]$  can be used to sample a function at a specific  $n = n_0$  since the impulse function will take on the value 0 for all values of  $n \neq n_0$

$$x[n]\delta[n - n_0] = x[n_0]\delta[n - n_0] \quad (11.6)$$

The continuous equivalents of the unit impulse and unit step functions are defined similarly

Definition 24

$$u(t) = \begin{cases} 0 & t < 0 \\ 1 & t > 0 \end{cases} \quad (11.7)$$

Likewise, the continuous unit step function is a running sum integral of the continuous unit impulse function

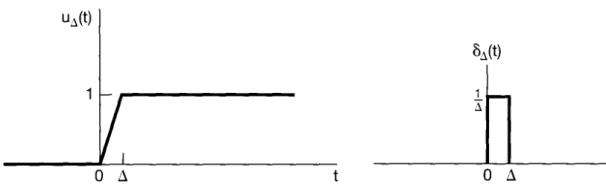
$$u(t) = \int_{-\infty}^t \delta(\tau) d\tau \quad (11.8)$$

A relationship analogous to the discrete case can be found for the continuous case; the continuous unit impulse function can be thought of as the first derivative of the continuous-time unit step function

**Definition 25**

$$\delta(t) = \frac{du(t)}{dt} \quad (11.9)$$

(11.9) is discontinuous at  $t = 0$  so it is non-differentiable. We can address this by considering an approximation of (11.9) for a  $\Delta$  short enough to not matter for any practical purpose



**Figure 1.33** Continuous approximation to the unit step,  $u_D(t)$ .

**Figure 1.34** Derivative of  $u_D(t)$ .

(11.8) can be rewritten as follows to make it more convenient to use along  $\sigma \in -\infty \dots 0 \dots \infty$ .

$$u(t) = \int_0^\infty \delta(t - \sigma) d\sigma \quad (11.10)$$

**Theorem 5**

And by the same argument as for the discrete case, the continuous impulse function has an important sampling property.

For any arbitrary point  $t_0$ ,

$$x(t)\delta(t - t_0) = x(t_0)\delta(t - t_0) \quad (11.11)$$

#### SUBSECTION 11.2

## Lecture 5

**Definition 26**

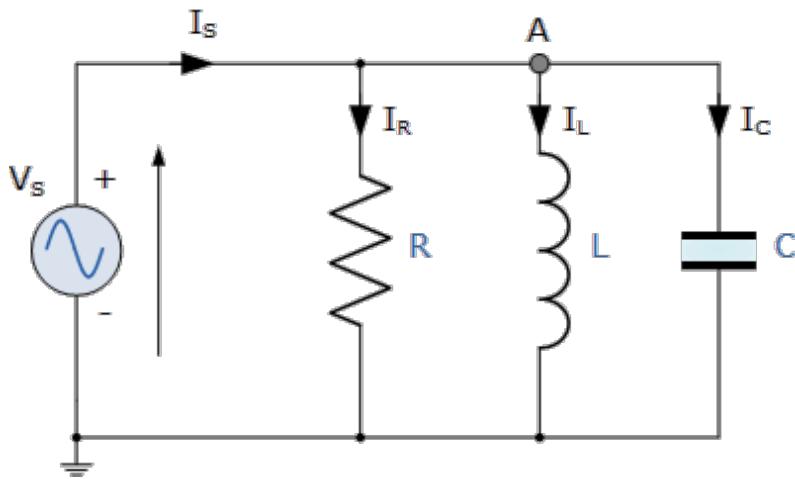
A **system** is a process that transforms a signal, i.e.

$$x(t) \xrightarrow{\text{sys}} y(t) \quad (11.12)$$

Some examples of the relationship between signals and physical systems include circuits, i.e. RLC circuits and spring-mass systems.

*Example* For example a RLC circuit can be modeled as a system that transforms a voltage signal into a current signal;

A similar definition can be applied for the discrete case



$$v(t) \xrightarrow{\text{RLC}} i(t) \quad (11.13)$$

Solving and modelling this system was covered in the other circuit classes.

Systems can be combined, i.e. making a mobile call

$$a(t) \xrightarrow{\text{mic}} y(t) \xrightarrow{\text{antenna}} z(t) \xrightarrow{\text{tower}} u(t) \xrightarrow{\text{antenna}} w(t) \xrightarrow{\text{speaker}} b(t) \quad (11.14)$$

### 11.2.1 Types of systems

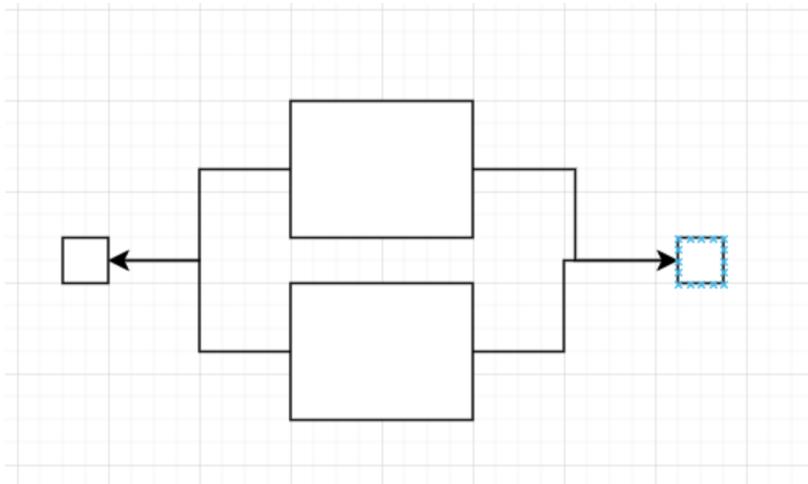


Figure 19. Parallel systems

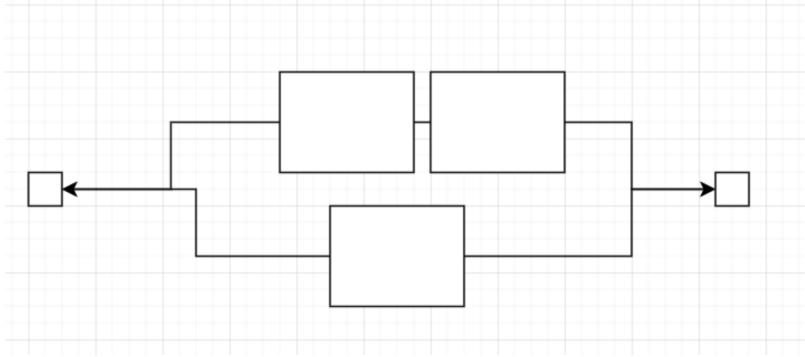
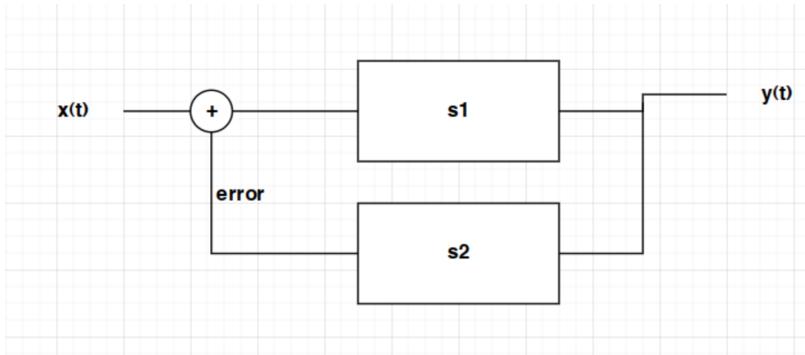


Figure 20. Series-parallel



The feedback control system will be discussed in depth in the control systems class

Figure 21. Feedback control system

### 11.2.2 System properties

Definition 27

**Memoryless** systems are systems where its output of the independent variable at a given time is dependent only on the input at the same time.

For example,

$$y[n] = 2x[n] - x^2[n] \quad (11.15)$$

is a memoryless system, but

$$y[n] = 2x[n] - x^2[n-1] \quad (11.16)$$

is not.

Other simple memoryless systems include the identity system  $x(t) = y(t)$

A system with memory is the *accumulator*

$$y[n] = \sum_{k=-\infty}^n x[k] \quad (11.17)$$

A capacitor is an example of a continuous-time system with memory

$$v(t) = \frac{1}{C} \int_{-\infty}^t i(\tau) d\tau \quad (11.18)$$

There can also be systems that are dependent on future values of the input and output<sup>18</sup>. <sup>18</sup>PID go brr

**Definition 28** A system is *invertible* if distinct inputs lead to distinct outputs<sup>19</sup> For example, the identity system is invertible, but the accumulator is not.

<sup>19</sup>Recall: MAT185 and matrix invertability

**Definition 29** A system is *casual* if its output at a given time is dependent only on the input at the current time and in the past.

Then it follows that

**Lemma 2** | All memoryless systems are causal

Though causal systems are useful, non-causal systems are also of great utility in modelling systems in which the independent variable is not time, or in *anticipative* models that account for the future values of the input or output, i.e a controller.

**Definition 30** A system is *stable* is if the output is bounded if the input is bounded.

**Definition 31** A system is *time invariant* if the behaviour and characteristics are fixed over time. For example, a *RC* circuit is time-invariant if the circuit  $R, C$  values are constant over time. More formally, a system is time invariant if a time shift in the input signal results in an identical time shift in the output.

If

$$x[n] \xrightarrow{\text{sys}} y[n] \quad (11.19)$$

Then, for a time invariant system,

$$y[n - n_0] \xrightarrow{\text{sys}} x[n - n_0] \quad (11.20)$$

**Definition 32** A system is *linear* if it possess the property of superposition, i.e. it possess the additive property and the scaling, or homogeneity property.

More formally, a linear system is one such that

$$ax_1(t) + bx_2(t) \xrightarrow{\text{sys}} ay_1(t) + by_2(t) \quad (11.21)$$

Where  $y_1$  is the output of a system with input  $x_1$  and  $y_2$  is the output of a system with input  $x_2$ .

## SECTION 12

### LTI systems

#### SUBSECTION 12.1

#### Lecture 6: Discrete LTI systems

Many physical systems can be modelled as linear time-invariant (LTI) systems. This is useful because there exists a large body of theory that can be applied to LTI systems, in part due to LTI systems possessing the superposition property.

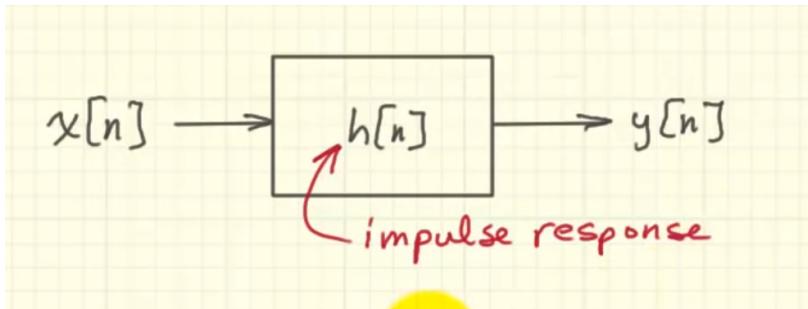


Figure 22. In this lecture the idea of LTI systems being a sum of  $h[n]$  impulse responses will be

**Theorem 6**

Discrete signals can be represented as a summation of impulses.

```

x[n] = \sum_{k=-\infty}^{\infty} x[k] \delta[n-k] \quad \text{Weighted sum of unit impulses}
\begin{aligned}
&\text{Load definition} \\
&\text{Begin definition} \\
&\text{A system is linear if it} \\
&\text{is formally a linear system is} \\
&\text{the definition} \\
&\text{where } y, z \text{ is the output of a} \\
&\text{and definition} \\
&\text{Addition} \\
&\text{Subtraction} \\
&\text{This is useful because there exists} \\
&\text{discrete signals can be represented} \\
&\text{as a sum of weighted shifted} \\
&\text{unit impulses} \\
&\text{End definition} \\
&\text{Begin theorem} \\
&\text{A discrete signal} \\
&\text{can be represented as a sum of} \\
&\text{shifted unit impulses} \\
&\text{End theorem} \\
&\text{End of lecture 1} \\
&\text{End of document}
\end{aligned}

```

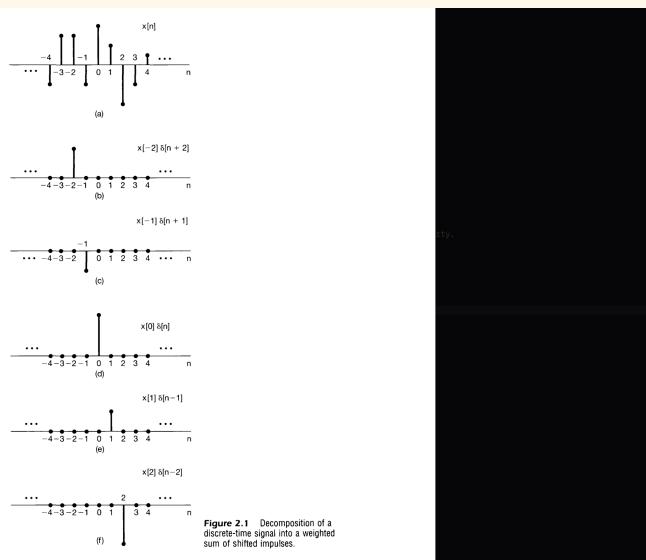


Figure 2.1 Decomposition of a discrete-time signal into a weighted sum of shifted impulses.

More generally, a discrete signal can be rewritten as a sum of shifted unit impulses with weights  $x[k]$

$$x[n] = \sum_{n=-\infty}^{\infty} x[k] \delta[n-k] \quad (12.1)$$

*Example*

As an example we can rewrite the unit step function as

$$u[n] = \sum_{n=0}^{\infty} \delta[n-k] \quad (12.2)$$

A useful result of the properties of LTI systems (i.e. the superposition property) is that the output of a system is the convolution of the input and the impulse response of the system, i.e.

**Definition 33**

**Convolution sum**

The discrete unit impulse also possesses a *sifting* property; since  $\delta[n-k]$  is nonzero only when  $k = n$ , this summation will sift through the values of  $x[k]$  and preserves only the values corresponding to  $k = n$

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k] \quad (12.3)$$

**Theorem 7**

An LTI system can be modelled as a **convolution** sum, or a sum of scaled and shifted impulse responses. So, if we know the response of the linear system to the set of shifted unit impulses – we can determine the response of the system to any input signal!

Symbolically convolution can be written as

$$y[n] = x[n] * h[n] \quad (12.4)$$

SUBSECTION 12.2

## Lecture 7: Continuous LTI systems

The concept of a LTI system responding to unit impulses and taking advantage of the sifting property may be extended to the continuous systems by idealizing the pulse as one so short that its duration is inconsequential for any real system.

**Definition 34**

**Sifting property** of continuous-time impulse

$$x(t) = \int_{-\infty}^{\infty} x(\tau)\delta(t-\tau)d\tau \quad (12.5)$$

Can interpret  $x(\tau)\delta(t-\tau)$  equals  $x(t)\delta(t-\tau)$ , i.e. that it is a *scaled impulse*

$$x(t) = \int_{-\infty}^{\infty} x(\tau)\delta(t-\tau)d\tau = x(\tau) \int_{-\infty}^{\infty} \delta(t-\tau)d\tau \quad (12.6)$$

If we let  $h_\tau(t)$  denote the response at time  $t$  to an unit impulse  $\delta(t-\tau)$  at time  $\tau$ , then

$$y(t) = \lim_{\Delta \rightarrow 0} \sum_{k=-\infty}^{\infty} x(k\Delta)\hat{h}_{k\Delta}(t)\Delta \quad (12.7)$$

As  $\Delta \rightarrow 0$  the summation becomes an integral, i.e.

$$y(t) = \int_{-\infty}^{\infty} x(\tau)h_\tau(t)d\tau \quad (12.8)$$

For notational convenience the  $\tau$  subscript can be dropped and generalizing to all  $t$

$$y(t) = \int_{-\infty}^{\infty} x(\tau)h(t-\tau)d\tau \quad (12.9)$$

Symbolically the convolution is represented  $y(t) = x(t) \cdot h(t)$

SUBSECTION 12.3

## Lecture 8

PART

IV

# ECE358: Foundations of Computing

SECTION 13

Taught by Prof. Shurui Zhou

## Admin and Preliminary

SUBSECTION 13.1

### Lecture 1

Topics covered will include:

- Graphs, trees
- Bunch of sorts
- Fancy search trees; red-black, splay, etc
- DP, Greedy
- Min span tree, single source shortest paths
- Maximum flow
- NP Completeness, theory of computation
- Blockchains??
- $\Theta$

Solutions will be posted on the window of SF2001. Walk there and take a picture.

#### 13.1.1 Mark Breakdown

**Table 3.** Mark Breakdown

Homework x 5	25
Midterm (Open book)	35
Final (Open book)	40

SECTION 14

## Complexities

SUBSECTION 14.1

### Lecture 2

This lecture we talked about big O notation. For notes on this refer to my tutorial notes for ESC180, ESC190: <https://github.com/ihasdapie/teaching/>

**Definition 35**

Big O notation (upper bound)  
 $g(n)$  is an asymptotic upper bound for  $f(n)$  if:

$$O(g(n)) = \{f(n) : \exists c, n_0 \text{ s.t. } 0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0\} \quad (14.1)$$

**PROOF**

**What is the big-O of  $n!$ ?**

$$n! \leq n \cdot n \cdot n \cdot n \dots n = n^n \Rightarrow n! \in O(n^n) \quad (14.2)$$

□

**Definition 36**

Big  $\Omega$  notation (lower bound)  
 $h(n)$  is an asymptotic lower bound for  $f(n)$  if:

$$\Omega(h(n)) = \{f(n) : \exists c, n_0 > 0 \text{ s.t. } 0 \leq c \cdot h(n) \leq f(n), \forall n \geq n_0\} \quad (14.3)$$

**PROOF**

**Find  $\Theta$  for  $f(n) \sum_i^n i$ .**

For this we will employ a technique for the proof where we take the right half of the function, i.e. from  $\frac{n}{2} \dots n$  and then find the bound

$$\begin{aligned} f(n) &= 1 + 2 + 3 \dots + n \\ &\geq \lceil \frac{n}{2} \rceil + (\lceil \frac{n}{2} \rceil + 1) + (\lceil \frac{n}{2} \rceil + 2) + \dots n \quad n/2 \text{ times} \\ &\geq \lceil \frac{n}{2} \rceil + \lceil \frac{n}{2} \rceil + \lceil \frac{n}{2} \rceil + \dots \lceil \frac{n}{2} \rceil \\ &\geq \frac{n}{2} \cdot \frac{n}{2} \\ &= \frac{n^2}{4} \end{aligned} \quad (14.4)$$

And therefore for  $c = \frac{1}{4}$  and  $n = 1$ ,  $f(n) \in \Theta(n^2)$

□

**Definition 37**

Big  $\Theta$  notation (asymptotically tight bound)

$$\Theta(g(n)) = \{f(n) : \exists c_1 c_2, n_0 \text{ s.t. } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\} \quad (14.5)$$

**PROOF**

Prove that

$$\sum_{j=1}^n i^k = \Theta(n^{k+1}) \quad (14.6)$$

First, prove  $O(f(n)) = O(n^{k+1})$

$$\begin{aligned} f(n) &= \sum_{j=1}^n i^k = 1^k + 2^k + \dots n^k \\ &\leq n^k + n^k + \dots n^k \\ &= n^{k+1} \end{aligned} \quad (14.7)$$

Next, prove  $\Omega(f(n)) = \Omega(n^{k+1})$

$$\begin{aligned}
 f(n) &= \sum_{j=1}^n i^k = 1^k + 2^k + \dots n^k \\
 &= n^k + (n_1)^k + \dots 2^k + 1^k = \sum_{i=1}^n (n - i + 1)^k \\
 &\geq \frac{n^k}{2} * n \geq \frac{n^{k+1}}{2^k} = \Omega(n^{k+1})
 \end{aligned} \tag{14.8}$$

Therefore  $f(n) = \Theta(n^{k+1})$  □

Note that we may not always find both a tight upper and lower bound so not all functions have a tight asymptotic bound.

**Theorem 8**

**Properties of asymptotes:**

Note:  $\wedge$  means AND

**Transitivity**<sup>20</sup>

$$(f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n))) \Rightarrow f(n) = \Theta(h(n)) \tag{14.9}$$

**Reflexivity**<sup>21</sup>

$$f(n) = \Theta(f(n)) \tag{14.10}$$

**Symmetry**

$$f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n)) \tag{14.11}$$

**Transpose Symmetry**

$$\begin{aligned}
 f(n) = O(g(n)) &\iff g(n) = \Omega(f(n)) \\
 f(n) = o(g(n)) &\iff g(n) = \omega(f(n))
 \end{aligned} \tag{14.12}$$

<sup>20</sup> The following applies to  $O, \Theta, o, \omega$

<sup>21</sup> The following applies to  $O, \Theta$

Runtime complexity bounds can sometimes be used to compare functions. For example,  $f(n) = O(g(n))$  is like  $a \leq b$

- $O \approx \leq$
- $\Omega \approx \geq$
- $\Theta \approx \approx$
- $o \approx <$ ; an upper bound that is **not** asymptotically tight
- $\omega >$  a lower bound that is **not** asymptotically tight

Note that there is no trichotomy; unlike real numbers where we can just do  $a < b$ , etc, we may not always be able to compare functions.

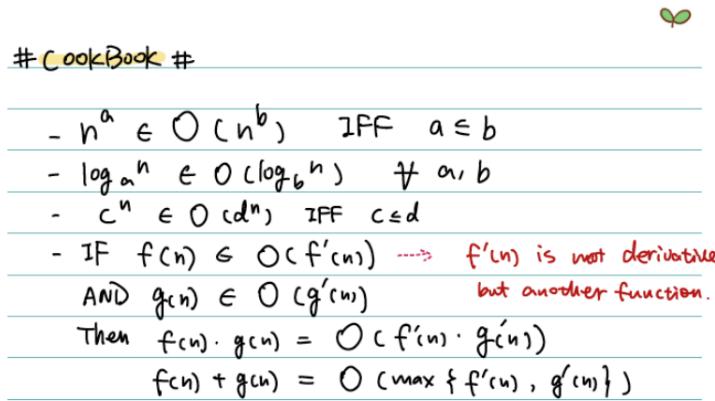


Figure 23. Complexity Cookbook

## SUBSECTION 14.2

## Lecture 3: Logs &amp; Sums

## 14.2.1 Functional Iteration

Recall:

 $f^{(i)}(n)$  denotes a function iteratively applied  $i$  times to value  $n$ .

$$a = b^c \Leftrightarrow \log_b a = c \quad (14.13)$$

For example, a function may be defined as:

$$f^{(i)}(n) = \begin{cases} f(n) & \text{if } i = 0 \\ f(f^{(i-1)}(n)) & \text{if } i > 0 \end{cases} \quad (14.14)$$

Given (14.14) we see that

1.  $f(n) = 2n$
2.  $f^{(2)}(n) = f(2n) = 2^2 n$
3.  $f^{(3)}(n) = f(f^{(2)}(n)) = 2^3 n$
4.  $f^{(i)}(n) = 2^i n$

As an exercise we may look at an iterated logarithm function, 'log star'

$$\lg^*(n) = \min\{i \geq 0 : \lg^{(i)} n \leq 1\} \quad (14.15)$$

This describes the number of times we can iterate  $\log(n)$  until it gets to 1 or smaller.

- $\log^* 2 = 1$
- $\log^* 4 = 2 = \log^* 2^2 = 1 + \log^* 2 = 2$
- for practical reasons  $\log^*$  doesn't really get bigger than 5. This is one of the slowest growing functions around.

## Summations &amp; Series

PROOF | Proof for a finite geometric sum:

$$\begin{aligned}
 \sum_{k=0}^n x^k &= S \\
 S &= 1 + x + x^2 \dots x^n \\
 xS &= x + x^2 + x^3 \dots x^{n+1} \\
 S &= \frac{1 - x^{n+1}}{1 - x}
 \end{aligned} \tag{14.16}$$

□

$$\sum_{i=1}^{\infty} x^i = \frac{1}{1 - x} \quad \text{if } |x| < 1 \tag{14.17}$$

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1 - x)^2} \quad \text{if } |x| < 1 \tag{14.18}$$

PROOF Begin by differentiating both sides over  $x$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{(1 - x)} \quad \text{if } |x| < 1 \tag{14.19}$$

$$\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1 - x)^2} \tag{14.20}$$

And then multiply both sides by  $x$ , therefore (14.18) follows. □

### Telescoping Series

$$\sum_{k=1}^n a_k - a_{k-1} = a_n - a_0 \tag{14.21}$$

PROOF Write it out and cancel out terms

$$(a_1 - a_0) + (a_2 - a_1) \dots (a_n - a_{n-1}) = a_n - a_0 \tag{14.22}$$

Therefore the sum telescopes □

Another telescoping series may be proved similarly:

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} \xrightarrow{\text{math}} \sum_{k=1}^{n-1} \left( \frac{1}{k} - \frac{1}{k+1} \right) = \left( 1 - \frac{1}{2} \right) + \dots + \left( \frac{1}{n-1} - \frac{1}{n} \right) = a_o - a_n \tag{14.23}$$

#### SUBSECTION 14.3

## Lecture 4: Induction & Contradiction

### 14.3.1 Induction

The general steps for proving a statement by induction are:

1. Basis
2. Hypothesis
3. Inductive step

I.e. if the basis holds for some  $i$ , i.e.  $0, 1, 2, 3, 12, \dots$  AND if we assume that the hypothesis holds for an arbitrary number  $k$ , then we just need to prove that the inductive step follows, or that  $P(n + 1)$  holds.

*Example* | Prove that  $P(n) = 1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$

*PROOF* | 1. Basis:  $P(0) = 0 = \frac{0(0+1)}{2}$

2. Hypothesis (assume that it is true):  $P(k) = \frac{k(k+1)}{2}$

3. Inductive step (need to prove  $P(k + 1) = \frac{(k+1)(k+2)}{2}$ ):  $P(k + 1) = \underbrace{1 + 2 + \dots + n}_{\frac{n(n+1)}{2}} + (n + 1) = \dots = \frac{n^2 + 3n + 2}{2} = \frac{(n+1)(n+2)}{2}$

□

*Example* | Show that for any finite set  $S$ , the power set  $2^S$  has  $2^{|S|}$  elements (that is, there are  $2^{|S|}$  distinct subsets of  $S$ )

*PROOF* | 1. Basis:

$$n = 0, |S| = 0, |2^S| = 1 = 2^0 \quad (14.24)$$

$$n = 1, |S| = 1, |2^S| = 2 = 2^1 \quad (14.25)$$

2. Hypothesis: Assume that  $2^S$  has  $2^n$  elements when  $|S| = n$

3. Inductive step: need to prove that when  $|S| = n + 1$ ,  $|2^S| = 2^{n+1}$

Let  $B = S \setminus \{a\}$  for some  $a \in S$ . Now there are two types of subsets of  $S$ ; those that include  $a$  and those who do not include  $a$

For subsets that do *not* include  $a$ ,  $|2^B| = 2^{|B|} = 2^n$ , by the hypothesis.

For subsets that do include  $a$ , these sets are of size  $2^B \cup \{a\}$ , which is  $2^n$ .

Therefore the total number of subsets of  $S$  is  $2^n + 2^n = 2^{n+1}$ , as desired.

The power set of a set  $S$  is the set of all subsets of  $S$

The same kind of argument can be applied to problems such as the [Towers of Hanoi](#) and the tiling problem.

### 14.3.2 Contradiction

1. Assume the theorem is false
2. Show that the assumption is false (leads to a contradiction)
  - Therefore the theorem is true

*Example* | Prove that  $\sqrt{2}$  is irrational

PROOF

Assume that  $\sqrt{2}$  is rational.  
Therefore we can write  $\sqrt{2}$  as

$$\sqrt{2} = \frac{a}{b} \quad (14.26)$$

Where  $a, b$  **have no common factors**.

We can square both sides

$$2 = \frac{a^2}{b^2} \rightarrow a^2 = 2b^2 \quad (14.27)$$

Therefore  $a^2$  is even.

Let  $a = 2c$

$$2^2c^2 = 2b^2 \rightarrow b^2 = 2c^2 \quad (14.28)$$

Therefore  $b$  is even as well.

This results in a contradiction since we assumed that  $a, b$  have no common factors, but our analysis shows that both would have to be even (and share a common factor of 2).  $\square$

#### SUBSECTION 14.4

## Lecture 5: recurrences

Many recursive algorithms can be thought of as a divide-and-conquer approach where we break the problem into subproblems that are similar to the original but smaller in size, solve them recursively, then combine them to create a solution to the original problem.

Definition 38

A recurrence is a function defined in terms of:

- 1+ base cases
- Itself, with smaller arguments

For example, finding a Fibonacci number is a recurrence;

$$T(n) = T(n-1) + T(n-2) \text{ with some base cases.}$$

Example

### Mergesort

Sorting  $[3, 1, 7, 5]$

1. Divide: break into partitions:  $[[3, 1], [7, 5]]$
2. Sort partitions:  $[[1, 3], [5, 7]]$
3. Create result array
4. Compare: have two pointers to front of array
  - compare 1, 5. 1 is smaller;  $result = [] \leftarrow 1$
  - Move ptr to left array (1, 3) ahead one. Compare 3, 5. 3 is smaller, so  $result = [3] \leftarrow 3$
  - One of the arrays is now empty so we can just append the rest
5.  $result = [1, 3, 5, 7]$

**Definition 39**

Pseudocode for mergesort is given by:

```

1   mergesort(A, p, r)
2       if p < r
3           q = [(p+r)/2]
4           mergesort(A, p, q) // N/2
5           mergesort(A, q+1, r) // N/2
6           merge(A, p, q, r) // merge the sorted
    ↵    subarrays

```

This mergesort partitions in half each time<sup>22</sup>.In the worst case we will compare  $N - 1$  times, so  $O(N)$  worst case.Proving that merge sort is  $\Omega(N)$ <sup>22</sup>binary partitioning(?)

Here we're discussing not time complexity but rather the number of times we call mergesort

How much time does MergeSort take?

The time of mergesort is defined recursively as:

$$T(N) = \begin{cases} O(1) & n = 1 \\ T(N) = 2T(\frac{N}{2}) + \Theta(N) & \text{otherwise} \end{cases} \quad (14.29)$$

More generally we can find the runtime of a recurrence algorithm via

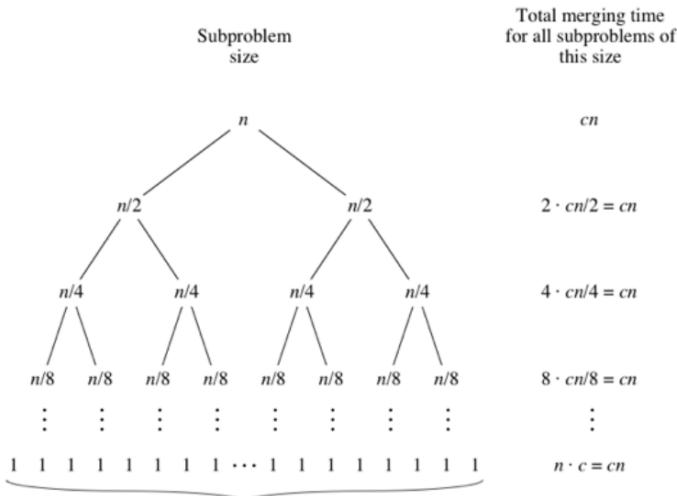
- Recurrence trees
- Substitution
- Master theorem

Note that  $\Theta(N)$  is for the merge operation

#### 14.4.1 Recurrence Trees

*Example*

Recurrence trees can be used to find the time complexity of mergesort.

The height of the tree is  $\log N$ 

The total cost is the total cost per level times the number of levels, which is

$$N \cdot \log N \quad (14.30)$$

| So the complexity is  $O(N \log N)$

#### 14.4.2 Substitution

1. Guess the answer
2. Apply induction

Example | Determine an asymptotic upper bound on  $T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + \Theta(n)$

PROOF | This expression can be simplified since we don't care too much about floors or ceilings for asymptotic behaviour.

$$T(n) = 2T\left(\frac{n}{2}\right) + N \quad (14.31)$$

Let's guess that the upper bound is  $O(n \log n)$

Then, we need to prove that  $T(n) < C \cdot n \log n$  for some  $C > 0$ . Let's apply induction.

1. Basis: this is tricky since if  $n = 1$  we end up with  $T(1) \leq C \cdot 1 \cdot \log 1 = 0$  which cannot hold since that would just not make sense. Instead, observe that  $T(1) = 1$ ,  $T(2) = 2T(1) + 2 = 4$ ,  $T(3) = 2T(1) + 3 = 5$ ,  $T(4) = 2T(2) + 4 \dots$   
So  $T(n)$  is therefore independent of  $T(1)$ , so we can use two bases,  $T(2), T(3)$ . Since  $T(2) \leq C * 2 \log 2 = 2C$ ,  $T(3) \leq C \cdot \log 3$
2. Hypothesis: Assume that the upper bound holds for all possible  $m < n$ , let  $m = \lfloor \frac{n}{2} \rfloor$ . This yields  $T(\lfloor \frac{n}{2} \rfloor) \leq C \cdot \lfloor \frac{n}{2} \rfloor \cdot \log \lfloor \frac{n}{2} \rfloor$
3. Inductive step: substitute hypothesis into recurrence yields

$$T(N) \leq C \cdot (C \cdot \left\lfloor \frac{N}{2} \right\rfloor \cdot \log \left\lfloor \frac{N}{2} \right\rfloor) + N = cN \log N - (1-c)N \leq Cn \log n \quad (14.32)$$

□

A few pitfalls to avoid is guessing  $T(n) = O(n) = c \cdot n$  and so forth we would get

$$T(N) \leq 2C \cdot \left\lfloor \frac{n}{2} \right\rfloor + n = cn + n = (c+1)n \quad (14.33)$$

This would be wrong since we cannot change the constant to  $c+1$ ; we have to prove it with exactly the hypothesis given.

#### 14.4.3 Master Theorem

Definition 40 | The master method applies to recurrences of the form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad \text{where } a \geq 1, b > 1, f \text{ asymptotically positive} \quad (14.34)$$

It distinguishes 3 common cases b comparing  $f(n)$  with  $n^{\log_b a}$

1. If  $f(n) = O(n^{\log_b a - \varepsilon})$  for some  $\varepsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$
2. If  $f(n) = \Theta(n^{\log_b a})$  for some  $\varepsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$
3. If  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for some  $\varepsilon > 0$  and  $af\left(\frac{n}{b}\right) \leq cf(n)$  for some  $c < 1$ , then then  $T(n) = \Theta(f(n))$

Proof is out of scope for the course

There are a few technicalities to be aware of. In each example we compare  $f(n)$  with  $F = n^{\log_b a}$  and take the larger of each as the solution to the recurrence. For the first case we

note that  $f(n)$  must be *polynomially* smaller than  $F$ ; i.e. it must be asymptotically smaller than  $F$  by some factor of  $n^\varepsilon$ . In the third case  $f(n)$  must be greater than  $F$  as well as being polynomially larger and satisfy the regularity condition  $af(\frac{n}{b}) \leq cf(n)$ . There are areas where the master theorem does not cover, for example a gap between cases 1, 2 where  $f(n) > F$  but is not polynomially larger. If  $f(n)$  falls in one of these gaps or the regularity condition does not hold, the master method cannot be used to solve the recurrence.

*Example* What is the closed form of  $T(n) = T(\frac{2n}{3}) + 1$ ?

*PROOF*  $a = 1, b = 2/3, f(n) = 1$ .

$$\log_b a = \log_{\frac{2}{3}} 1 = 0 \quad (14.35)$$

$$f(n) = \Theta(n^0) \quad (14.36)$$

So

$$T(n) = O \log(n) \quad (14.37)$$

□

#### SUBSECTION 14.5

## Lecture 6

### 14.5.1 Graphs

**Definition 41**

A **graph** is a data structure comprised from set of vertices  $V$  and a set of edges  $E$ , where each edge connects a pair of vertices. A **directed graph (digraph)** is a graph where edges  $E$  have a *direction*, i.e. an edge  $(u, v)$  is different from  $(v, u)$ . Conversely, an **undirected graph** is a graph where edges  $E$  do not have a direction.

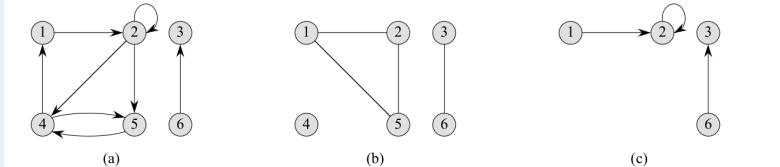


Figure 24. (a) directed graph, (b) undirected graph, (c) a subgraph of (a)

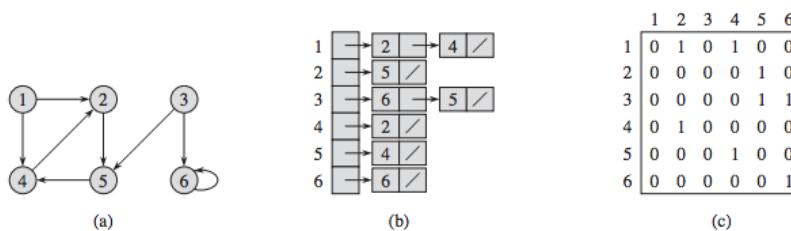
Some conventions:

- Edges are denoted by  $(u, v)^{23}$  where  $u, v \in V$
- If  $(u, v)$  is a edge in a directed graph, then  $(u, v)$  is incident from or leaves  $u$ , and is incident to or enters  $v$
- If  $(u, v)$  is a edge a graph, then  $u, v$  are adjacent
- The **degree** of a vertex is the number of edges incident to it
- A **path** is a sequence of vertices  $(v_0, v_1, \dots, v_k)$  from vertex to another such that each vertex is incident<sup>24</sup> to the ones prior and after.
  - If there exists a path from  $a$  to  $b$  then  $b$  is **reachable** from  $a$  and  $a$
  - A path is **simple** if no vertex is repeated
  - A path forms a cycle if the first and last vertices are the same

<sup>23</sup>not  $\{u, v\}$   
<sup>24</sup>with the exception of start/end vertices

- A directed graph with no self-loops is **simple**
- A graph with no simple cycles is **acyclic**
- An undirected graph is **connected** if there exists a path between any two vertices
- A directed graph is **strongly connected** if every vertex is reachable from every other vertex
- Two graphs  $V, V'$  are **isomorphic** if there exists a bijection<sup>25</sup> between the vertices of the two graphs such that the edges are preserved
- Given graph  $G, G' = (V', E')$  is a **subgraph** of  $G$  if  $V' \subseteq V$  and  $E' \subseteq E$
- Given a set  $V' \subseteq V$ , the subgraph of  $G$  **induced** by  $V'$  is the graph  $G' = (V', E')$  where  $E' = \{(u, v) \in E \mid u, v \in V'\}$
- Given an undirected graph, the **directed version** of  $G$  is  $G = (V, E')$  where  $(u, v) \in E'$  if and only if  $(u, v) \in E$ . In other words we replace all undirected edges and replace them with their directed counterpart.
- The corollary can be applied to a directed graph to get the **undirected version** of  $G$ .
- A **neighbor** of  $u$  in a directed graph is any vertex  $v$  such that  $(u, v) \in E$  where  $E$  is the set of edges for the undirected counterpart of the graph
- A **complete** graph is a graph where every pair of vertices are connected by an edge
- A **bipartite graph** is an undirected graph  $G$  in which it's  $V$  can be partitioned into two disjoint sets  $V_1, V_2$  such that every edge  $(u, v) \in E$  connects a vertex in  $V_1$  to a vertex in  $V_2$  or vice-versa
- An acyclic undirected graph is a **forest**
- A connected acyclic undirected graph is a **free tree**.
  - A directed acyclic graph is often termed a DAG
- A multi-graph is a graph where edges can be repeated and self-loops are allowed
- A hyper-graph is a graph where edges can connect more than two vertices
- The **contraction** of an undirected graph  $G$  by an edge  $e = (u, v)$  is a graph  $G'$  where  $V' = V - \{u, v\} \cup \{x\}$ , where  $x$  is a new vertex. Then, for each edge connected to  $u, v$  the edges are deleted and then reconstructed with  $x$ , effectively 'contracting'  $u, v$  into a single vertex

In code graphs are commonly represented as adjacency lists or adjacency matrices. This was covered in ESC190, but for reference:



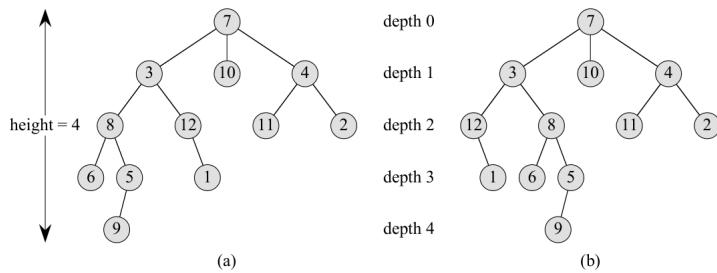
<sup>25</sup> $f : V \rightarrow V'$ , i.e. we can relabel the vertices of  $V$  to be those of  $V'$  and the two graphs would be identical

**Table 4.** Time complexities of graph representations

	adjacency list	matrix
Time	$O(n)$	$O(1)$
Memory	$O(E)$	$O(n^2)$

**14.5.2 Trees**

A **tree** is a common and useful subset of graphs



**Figure B.6** Rooted and ordered trees. (a) A rooted tree with height 4. The tree is drawn in a standard way: the root (node 7) is at the top, its children (nodes with depth 1) are beneath it, their children (nodes with depth 2) are beneath them, and so forth. If the tree is ordered, the relative left-to-right order of the children of a node matters; otherwise it doesn't. (b) Another rooted tree. As a rooted tree, it is identical to the tree in (a), but as an ordered tree it is different, since the children of node 3 appear in a different order.

**Definition 42**

A tree is a common subset of graphs, i.e. ones that are **connected, acyclic, and undirected**. This gives a few useful properties, i.e. the existence of a *root* node, the parent-child relationship, and the existence of a unique path between any two nodes.

Some conventions for trees

- Depth of node: length from root to node
- Height length of longest path from node to leaf
- Degree of node: number of children. Binary trees have degree 2, n-ary tree has degree n

**Theorem 9**

All of the following statements are equivalent for a tree  $T = (V, E)$ :

1.  $\forall v \in V, v$  is a tree; all nodes in a tree are trees unto themselves.
2. Every two nodes are connected by a unique path
3.  $T$  is connected by if any edge is removed the resulting graph is disconnected  $T$  is connected and  $|E| = |V| - 1$   $T$  is acyclic and  $|E| = |V| - 1$   $T$  is connected but if a edge is added the resulting graph has a cycle

## SUBSECTION 14.6

**Lecture 7, 8: Probability and Counting**

Most of the probability stuff is review from ECE286 so I'll be omitting most notes.

**Definition 43**

Probability distribution  $Pr \{ \cdot \}$ : mapping from events of  $S$  to real numbers where

1.  $Pr \{ \emptyset \} = 0$

2.  $Pr\{A\} \geq 0 \forall A \in S$
3.  $Pr\{S\} = 1$
4.  $Pr\{A \cup B\} = Pr\{A\} + Pr\{B\} \forall A, B \in S$

For any two events  $A, B$  we can define the triangle inequality

**Definition 44**  $Pr\{A \cup B\} \leq Pr\{A\} + Pr\{B\}$

The complement of an event  $A$  is  $\bar{A} = S - A$ , and the probability is  $Pr\{\bar{A}\} = 1 - Pr\{A\}$

**Definition 45** Baye's theorem

$$Pr\{A | B\} = \frac{Pr\{B | A\} Pr\{A\}}{Pr\{B\}} = \frac{Pr\{A\} Pr\{B | A\}}{Pr\{A\} Pr\{B | A\} + Pr\{\bar{A}\} Pr\{B | \bar{A}\}} \quad (14.38)$$

The expected value of a random variable is

$$E[X] = \int_{-\infty}^{\infty} x Pr\{X = x\} dx \quad (14.39)$$

And in the discrete case,

$$E[X] = \sum_{x \in \mathbb{Z}} x Pr\{X = x\} \quad (14.40)$$

The variance is

$$Var[X] = E\{(X - E[X])^2\} = E[X^2] - E^2[X] \quad (14.41)$$

#### SUBSECTION 14.7

## Lecture 9: Heapsort

The **standard deviation**,  $\sigma$ , is the positive square root of the variance.

Heapsort is a sorting algorithm that runs in  $O(n \log(n))$  and sorts its elements in place by leveraging a *heap*.

**Definition 46**

A heap is a data structure that satisfies the *heap property*. For a max heap, the value of a mode is at most the value of its parent and vice-versa for a min-heap. Heaps are commonly used to implement a priority queue because it offers a `pop{min, max}` operation in  $O(1)$  time and insertion in  $O(\log(n))$  time. Another reason for its use is its ability to be represented as an array with `Parent(i) = floor(i/2)`, `Left(i) = 2i`, `Right(i) = 2i+1`

This should be review from ESC190

The **MAXHEAPIFY** method is used to coerce heaps back into fulfilling the heap property by 'bubbling down' nodes until the heap property is satisfied. The intuition for this is that as long as the heap property is not satisfied<sup>26</sup>, node  $i$  is exchanged for the largest of its' children. Since the heap property is now satisfied for  $i$  but not necessarily its' children, this heapify operation is then recursively called on the larger of  $i$ 's children.

<sup>26</sup> which we can tell by looking at the node's children

```

1  MAXHEAPIFY(A, i)
2      l = LEFT(i)
3      r = RIGHT(i)
4      if l <= A.size and A[l] > A[i]
5          largest = l
6      else largest = i
7      if r <= A.size and A[r] > A[largest]
8          largest = r
9      if largest != i
10         swap A[i], A[largest]
11         MAXHEAPIFY(A, largest)

```

The child sub-trees can have size of at most  $\frac{2n}{3}$  and so we can describe MAXHEAPIFY by the recurrence relation

$$T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1) \quad (14.42)$$

This heapify operation can then be used to build a max heap from an array by calling heapify on all nodes that are not leaves. This is done by starting at the last node that has children and calling heapify on it. This is repeated until the root is reached. It can likewise be used to sort an array.

```

1  HEAPSORT(A)
2  for i = A.length -> 2,
3      swap A[1], A[i]
4      A.heapsize = A.heapsize - 1
5      HEAPIFY(A, 1)

```

SUBSECTION 14.8

## Lecture 10, 11: Quicksort

---

SUBSECTION 14.9

## Lecture 12: lower bound, counting and radix sort

---

SUBSECTION 14.10

## Lecture 13: Select & BST

---

PART

V

# ECE360: Electronics

SECTION 15

## Admin and Preliminary

---

Taught by Prof. Khoman Phang

SUBSECTION 15.1

## Lecture 1

---

### 15.1.1 Mark Breakdown

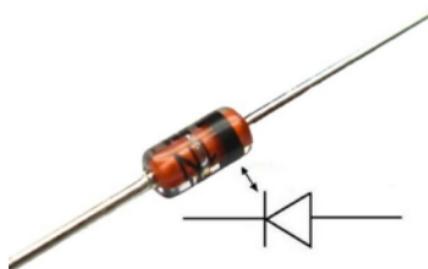
**Table 5.** Mark Breakdown

Test 1	15
Test 2	20
Homework	10
Labs	12
Final	43

### 15.1.2 Diodes

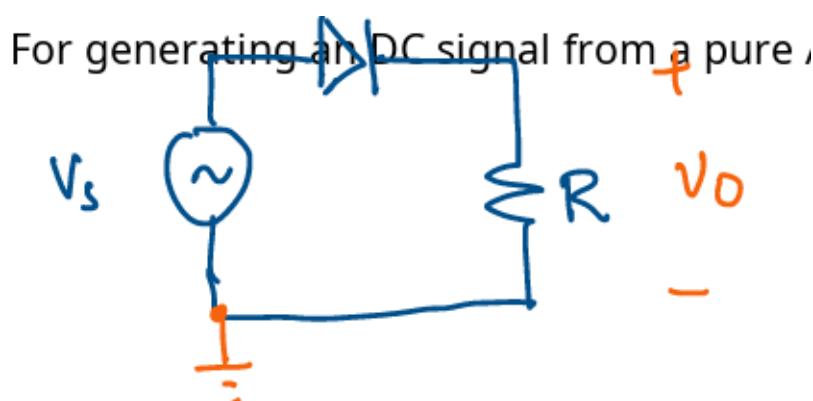
Diodes are an electronic valve which causes current to only flow in one direction. An ideal diode is an open circuit in the closed direction and a closed circuit in the other, so the current is always in the direction of the arrow (+'ve @ arrow base, -'ve at arrow point)<sup>27</sup>.

<sup>27</sup> recall: passive sign convention

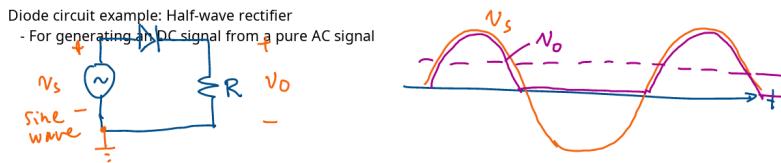


**Figure 25.** A diode and its symbol

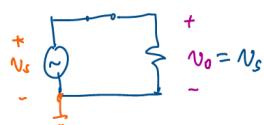
An example of a diode circuit is the half-wave rectifier which turns an AC signal to a DC signal



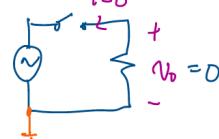
Can take oscilloscope over resistor to see that a pure DC signal has been generated



If  $N_s > 0$ , diode is "ON"



If  $N_s < 0$ , diode is "OFF"



## SECTION 16

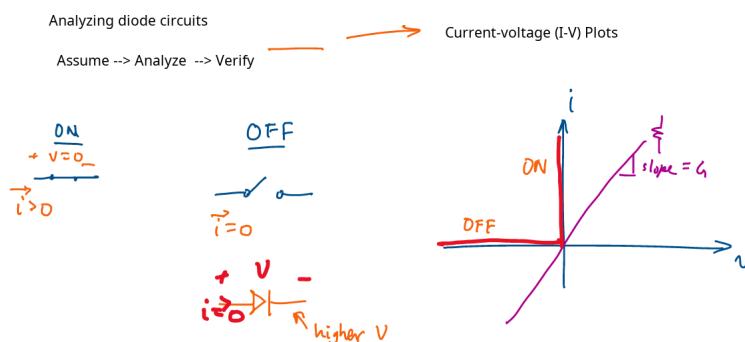
## Diodes

### SUBSECTION 16.1

## Lecture 2

More formally, off/on for diodes should be referred to as:

- Off  $\leftrightarrow$  reverse bias
  - On  $\leftrightarrow$  forwards bias



**Figure 26.** General steps for analyzing non-linear circuits. Note plotting out expected response

An example of how this is used in circuit design is to manage two power sources. Consider an Arduino that could be powered by an AC adapter or by a computer's USB port. This circuit would choose the higher voltage source and prevent back-flow into the other power source due to any potential power differentials. It is also effectively an OR gate

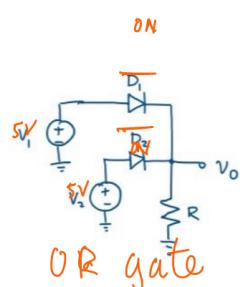
## Analysis Examples

'0' '1'

## Example 1:

Find output voltage  $V_o$  assuming input voltages  $V_1$  and  $V_2$  are either 0V or 5V.

What is the function of this circuit?

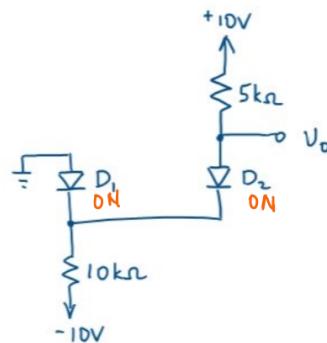


$V_1$	$D_1$	$V_2$	$D_2$	$V_o$
5V	ON	5V	ON	5V
5	ON	0	OFF	5V
0	OFF	5	ON	5V
0	OFF	0	OFF	0V

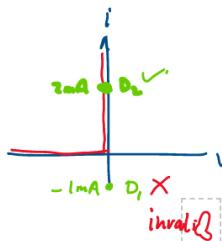
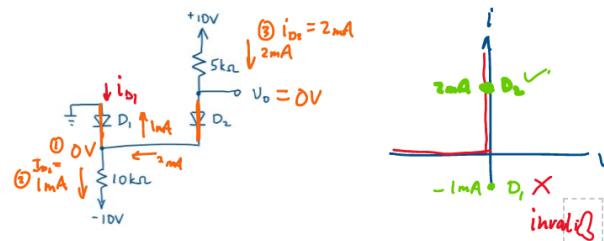
## Example 2:

Find output voltage  $V_o$ 

Assume  $\rightarrow$  Analyze  $\rightarrow$  Verify



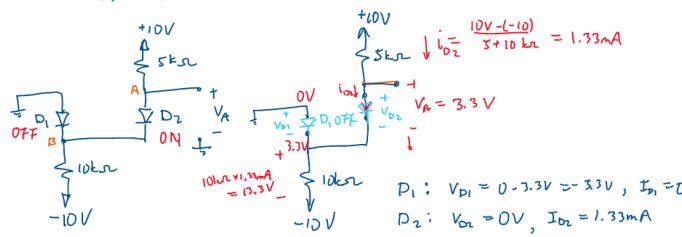
Assume  $D_1$  ON  
 $D_2$  ON



In this example the initial assumption was incorrect.  
 Let's try another analysis with  $D_1$  off and  $D_2$  on:

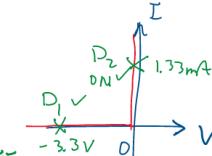
Second attempt

Step 1: Assume  
 $D_1$  OFF,  $D_2$  ON  $\Rightarrow$  Step 2: Analyze circuit

Step 3: verify assumptions

assume  $I \rightarrow$  check  $V < 0$   
 $D_1$  OFF  $\rightarrow$  open  $\rightarrow I_1 = 0$   
 $D_1 \not\rightarrow V_{D1} = -3.3V < 0 \checkmark$   
 Diode  $D_1$  is OFF

$D_2$  ON  $\rightarrow$  short  $\rightarrow V_{D2} = 0V$   
 $D_2 \not\rightarrow I_{D2} = 1.33mA > 0 \checkmark$   
 Diode  $D_2$  is ON



If we were to do this brute force we'd have to consider 4 cases, so it's important to build up some sort of intuition for the circuit.

## SUBSECTION 16.2

**Lecture 3**

Today we're going to look at the characteristics of real diodes.

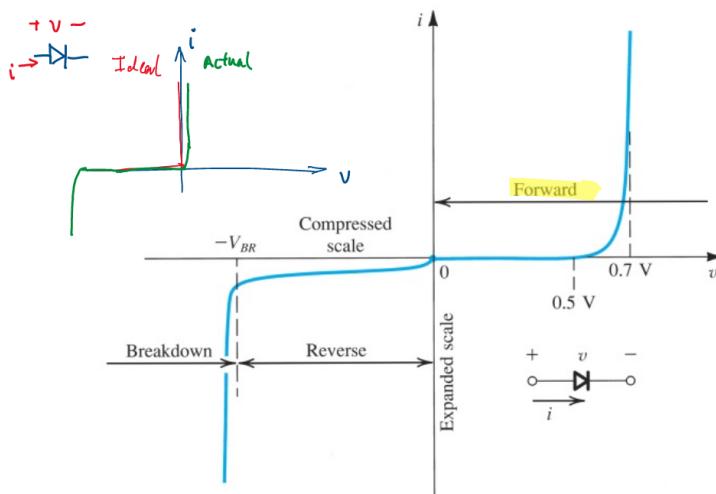


Figure 4.8 The silicon diode  $i$ - $v$  relationship with some scales expanded and others compressed in order to reveal details.

Real diodes have a little bit of leakage current and also encounter a breakdown point where they're no longer able to block the current.

Theorem 10

**Forward Bias**

$$i = I_s (e^{\frac{V}{V_T}} - 1) \quad (16.1)$$

Where:

$$V_T = \frac{kT}{q} \quad [V] \quad (16.2)$$

Most of the time we can assume that the circuit is at room temperature and that  $v_T = 25mV$ . Note that this value explodes when  $V > V_T$  which is the breakdown point. When encountering a reverse bias  $V_s < 0$ , the  $-1$  term comes in and causes  $i \approx I_s$ . The scale current is just a general constant which varies in range from  $10^{-9}$  to  $10^{-15} A$  and scales with temperature, doubling with every approximately  $5^\circ C$  increase in temperature. Note: the ideal diode equation can be rearranged to find an expression for voltages

$$V = V_T \ln \left( \frac{i}{I_s} \right) = \ln(10) V_T \log_{10} \left( \frac{i}{I_s} \right) \quad (16.3)$$

These expressions turns out to be quite reliable for reasonable diodes to reasonable voltages.

$k$  is Boltzmann's constant,  $T$  is temperature in Kelvins,  $q$  is the charge of an electron.  $I_s$  is the scale current which is usually  $\approx 1pA$ , which doesn't change much until the breakdown point.

Using the ideal diode equation we can find the relationship between voltages and currents as they pass through the diode.

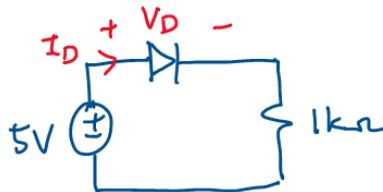
$$\frac{i_2}{i_1} = \frac{I_s e^{\frac{V_2}{V_T}}}{I_s e^{\frac{V_1}{V_T}}} = e^{\frac{V_2 - V_1}{V_T}} \quad (16.4)$$

$$V_2 - V_1 = V_T \ln \left( \frac{i_2}{i_1} \right) \xrightarrow{\text{room temperature}} 60mV \log_{10} \frac{i_2}{i_1} \quad (16.5)$$

Example:

Calculate the diode voltage and current in the circuit below.

Assume that the diode voltage is 0.7V at 1mA and  $V_T = 25mV$ .



Example

Recall (16.1). Plugging in the given values gives us the scale current.

$$1mA = I_s e^{\frac{0.7V}{25mV}}, I_s = 6.9 \cdot 10^{-16} A \Rightarrow I_o = I_s e^{v_o/v_T} \quad (16.6)$$

Ohm's law can then be applied at the resistor

$$V_r = IR = I_o R = 5V - V_D \Rightarrow 5 - V_D = I_o R \quad (16.7)$$

So we have two equations and two unknowns (since we know  $v_T = 25mV$  but  $v_o$  was used at first just to find  $I_s$ ) Solving for the unknowns gives us:

- $V_o = 0.736V$

$V_D$  is the voltage across the diode

- | •  $I_D = 4.264mA$

## SECTION 17

## Lecture 4 & 5: Forward conducting diodes

The exponential model accurately describes the diode outside of the breakdown region, though its nonlinear behaviour makes it difficult to use.

For  $V_{DD} > 0.5V$

$V_{DD}$  is DC voltage,  $v_d$  is small signal voltage,  $V_D$  is the diode voltage

$$I_D = I_S e^{V_D/V_T} \quad (17.1)$$

Where

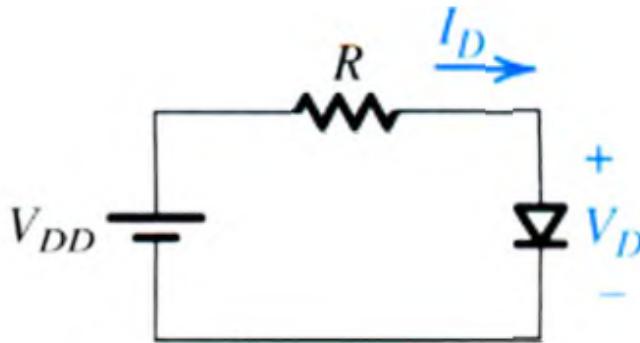
- $I_S$  is the diode parameter
- $V_T$  is the thermal voltage

Another equation may be produced via Kirchhoff's law

$$I_D = \frac{V_{DD} - v_d}{R} \quad (17.2)$$

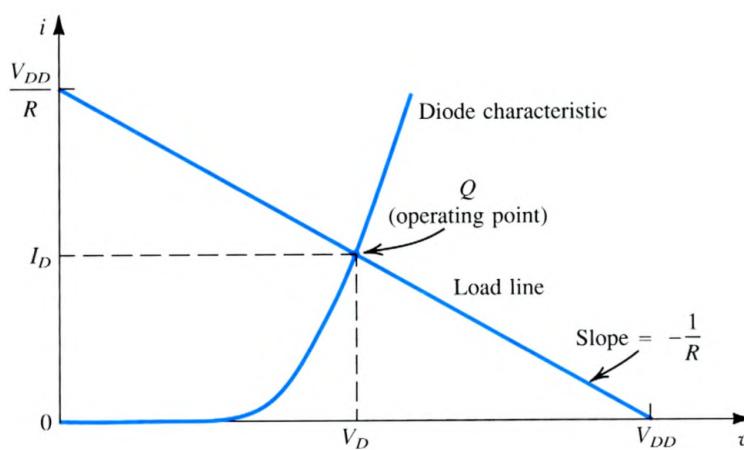
The unknown quantities  $I_D$  and  $v_d$  may be solved for via graphical analysis or iteration.

*Example* This simple circuit is used to demonstrate the exponential model of the diode.



**Figure 27.** Simple example circuit with diode

Plots of the diode characteristics and Kirchhoff's relation are plotted, the intersection of which gives the solution.



An iterative procedure may also be applied to solve for the unknowns, the procedure for which will be illustrated through an example

*Example* Find  $I_D$ ,  $V_D$  for the circuit in the previous example (Fig. 27).  $V_{DD} = 5V$ ,  $R = 1k\Omega$ , and at  $V_D 0.7V$ ,  $I_D = 1mA$

1. Assume  $V_D = 0.7V$ , then use (17.2) to find  $I_D$ .

$$I_D = \frac{5V - 0.7V}{1k\Omega} = 4.3mA \quad (17.3)$$

2. Use the diode equation (17.1) to get a better estimate for  $V_D$ .

$$V_2 - V_1 = 2.3V_T \log \frac{I_2}{I_1} \Rightarrow V_2 = V_1 + 0.06 \log \frac{I_2}{I_1} \quad (17.4)$$

substituting  $V_1 = 0.7V$ ,  $I_1 = 1mA$ ,  $I_2 = 4.3mA$ ,

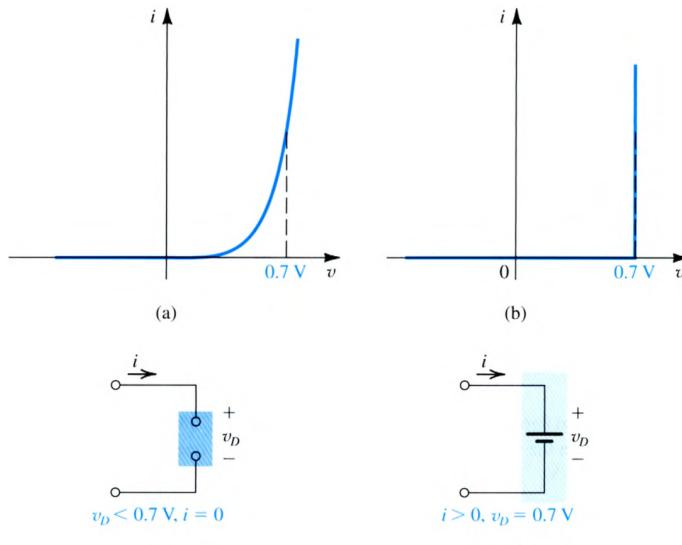
$$V_2 = 0.738V \Rightarrow I_D = 4.3mA, V_D = 0.738V \quad (17.5)$$

- This states that for a decade<sup>28</sup> change in current the diode voltage drop changes by  $2.3(V_T \approx 60mV)$  which is negligibly small for  $v < 0.5V$ . The voltage at which this behaviour becomes significant is called the **cut-in voltage**

<sup>28</sup>Factor of 10

3. Repeat steps 1 and 2 with the new values until the values more or less become stable

This iterative model is powerful and yields accurate results, but can be computationally expensive especially when calculating by hand. To address this we employ other models such as the *constant-voltage-drop* model which approximates the exponential characteristics via a piecewise linear model. The reason why this is possible is because forward conducting diodes exhibit a voltage drop that varies in a relatively narrow range.



Using the constant voltage drop model in our analysis looks the same as before, but with  $V_D$  directly taking on the value of  $0.7V$  (as per the prior example) instead of being solved for with the diode equation.

In applications that involve voltages greater than the voltage drop (i.e. usually  $\approx 0.6 - 0.8V$ ) we can neglect the diode voltage drop altogether while calculating the diode current.

$$V_D = 0V$$

$$I_D = \frac{5 - 0}{1} = 5mA \quad (17.6)$$

This is generally good enough for a first estimate, though the previous model isn't that much more work and gives more accurate results. The primary use of this model is to determine which diodes are on or off in a multi-diode circuit

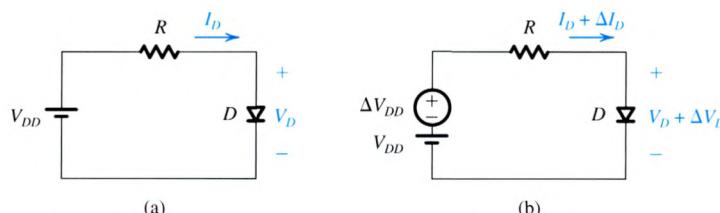
#### SECTION 18

## Small-Signal Model

The small signal method is an alternative model used to describe the nonlinear diode's characteristics with greater accuracy than piecewise linear models.

Consider a small  $\Delta V_{DD}$  applied to the diode, which would cause a small  $\Delta I_D, \Delta V_D$ . We want to find a quick way of determining the values of these incremental changes.

Similar methods will be applied to transistors in later chapters



Skipping a bunch of math<sup>29</sup> the results are as follows: <sup>30</sup>

<sup>29</sup> It is 11:17pm and I have two more lectures to catch up to today

<sup>30</sup> Small signal analysis can be performed separately from the dc bias analysis because of the linearization of diode characteristics in the small-signal ap-

Definition 47

**Small signal approximation**

$$i_D(t) \approx I_D \left( 1 + \frac{v_d}{V_T} \right) \quad (18.1)$$

This is valid for when variations in diode voltage  $|v_d| \lesssim 5mV$ .

$$r_d = \frac{V_T}{I_D} \quad (18.2)$$

From this we can define the small signal resistance as the resistance relating  $i_d$  to  $v_d$

The steps for calculating the small signal model are as follows:

1. Perform a dc analysis using the exponential, constant-voltage-drop, or piecewise-linear model.
2. Linearise the circuit. For a forward-based diode, find  $r_d$  by substitution  $I_D$  into (18.2). The small-signal equivalent circuit is found by eliminating all independent dc sources<sup>31</sup> and replacing the diode with its small-signal resistance  $r_d$
3. Solve the linearised circuit. In particular we would want to find  $\Delta I_D$ ,  $\Delta V_D$  and check to see if it is consistent with our approximation, i.e. that  $\Delta V_D \lesssim 5mV$

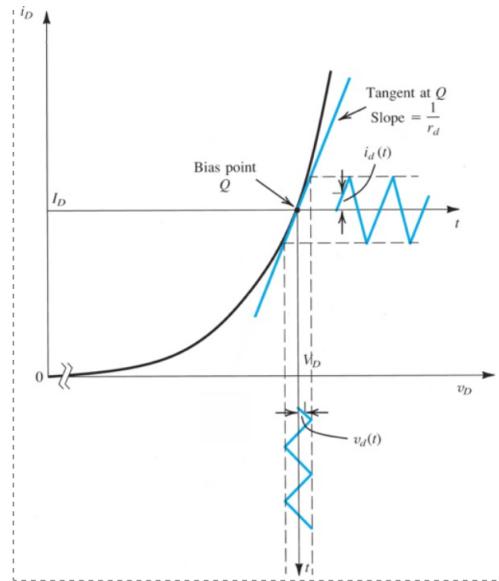
<sup>31</sup> since we already accounted for them in step 1

The reason why we linearise these non-linear systems, we, as engineers, try to linearise them because it is convenient to be able to use superposition, phasors, Fourier series, Laplace transforms, and so forth.

SECTION 19

**Lecture 6: Small signal model, cont'd**

$V_D$  can be thought of an input to a transfer function that is the diode, with  $I_D$  being the output. If the input signal is more or less a triangular wave the output will be as well.



Since we are applying a linear approximation, superposition may apply;

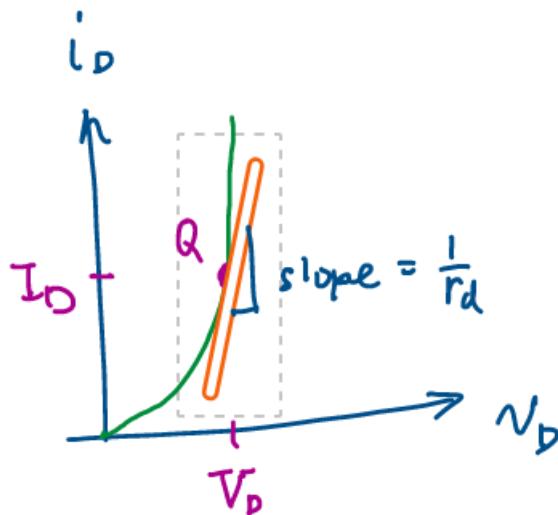
$$f(a + b + c \dots) = f(a) + f(b) + f(c) + \dots \quad (19.1)$$

Here we can think of the function as

$$f(V_o + v_d) = f(V_o) + f(v_d) \quad (19.2)$$

Small signal analysis works because, by superposition, we can zero out the other sources ( $V_o$ , etc) and inspect only the effect of the small signal voltage on the system.

### 19.0.1 Deriving small-signal resistance



Resistance can be found as the slope of the  $I - V$  relationship. The following is a proof of (18.2).

PROOF

$$i_D = I_s e^{\frac{V_D}{V_T}} \quad (19.3)$$

$$\text{slope} = \frac{di_D}{dv_o} = I_s \left( \frac{1}{V_T} \right) e^{\frac{V_D}{V_T}} \quad (19.4)$$

Then, substitute values at the operational point  $Q$ , i.e.  $I_D, V_{DD}$

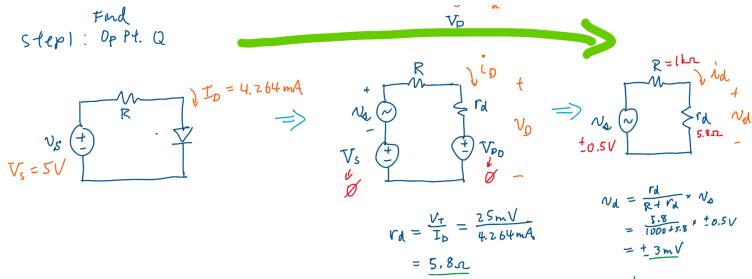
$$\text{slope} = I_s e^{V_D/V_T} \left( \frac{1}{V_T} \right) \Rightarrow \frac{I_D}{V_T} = \frac{1}{r_d} \quad (19.5)$$

And therefore

$$r_d = \frac{V_T}{I_D} \quad (19.6)$$

□

1. Calculate DC bias point  $Q$
2. Derive small-signal circuit
3. Analyze small-signal circuit
4. If required, recombine to arrive at final result



The textbook skips the middle step where we actually apply the small signal approximation and all the values have not been offset by the bias point yet. Note that voltages/values in the 3rd step (small signal approx) are all relative to the bias point  $Q$ .

Applying this to our circuit we find

$$r_d = \frac{V_T}{I_D} = \frac{25mV}{4.264mA} = 5.8\Omega \quad (19.7)$$

This was not as accurate as the exponential model but it is more than close enough 99% of the time. But also the  $0.7V$  constant voltage drop model is usually sufficient as well.

PROOF The small-signal approximation is valid for voltage variations up to  $\pm 5mV$

$$\begin{aligned} i_D &= I_S \exp(V_D/V_T) \\ &= I_S \exp \frac{V_D D + v_d}{V_T} \\ &= I_D \times e^{v_d/V_T} \end{aligned} \quad (19.8)$$

$e^x$  can be expanded to a power series

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \\ &\approx 1 + x \quad \text{if } \frac{x^2}{3!} \ll x \rightarrow x \ll 2 \end{aligned} \quad (19.9)$$

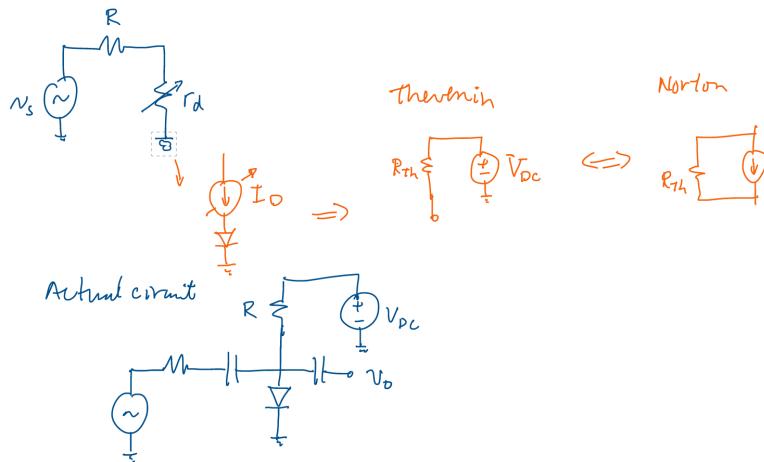
Therefore  $\frac{v_d}{V_T} \ll 2$ , so  $v_d \ll 2V_T = 50mV$   
So make

$$|v_d| < 5mV \quad (19.10)$$

□

In the lab we'll be using a variable attenuator circuit, which involves a voltage source. Can make a current source with a voltage source and a large resistor.

variable attenuator circuit



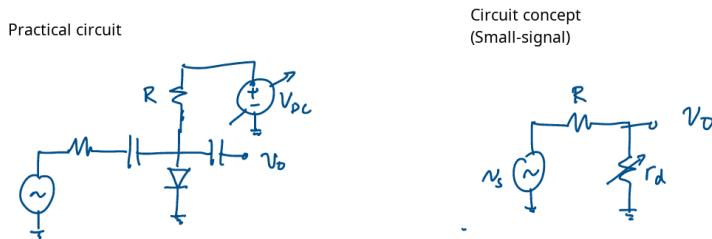
This will be covered more next lecture. TLDR: can use small signals and then build up a circuit to get the intended  $Q$  using biases.

## SECTION 20

## Lecture 7

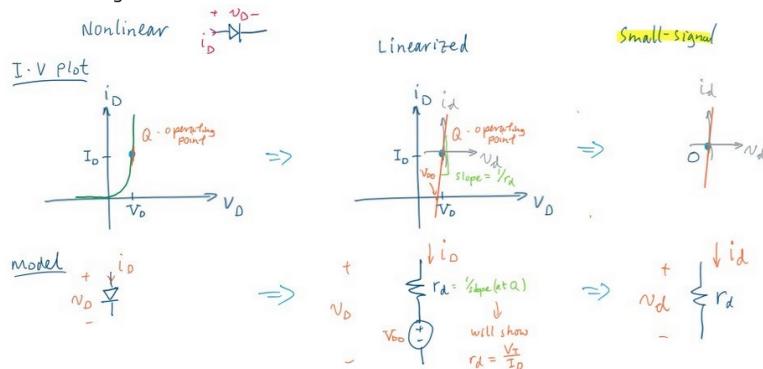
---

Let's take a deeper look into the variable attenuator circuit discussed last class.



- The capacitors in this circuit can be treated like short circuits because they are large (which is why we don't see them in the small-signal circuit).
- The constant voltage offset source gets zeroed out in the small-signal circuit because it is a constant, thereby becoming a ground.
- Current sources become open circuits
- Negative currents

What does a negative current mean?

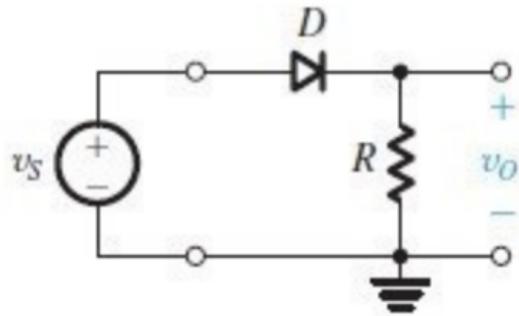


## SECTION 21

## Rectifiers

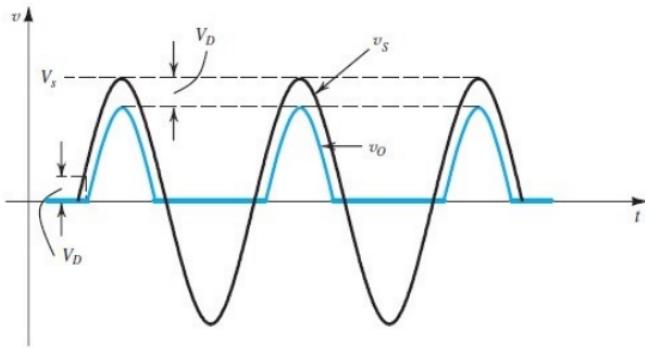
We have previously talked about the half-wave rectifier circuit, which used a single diode to rectify a waveform to only the positive values

recall half-wave rectifier:



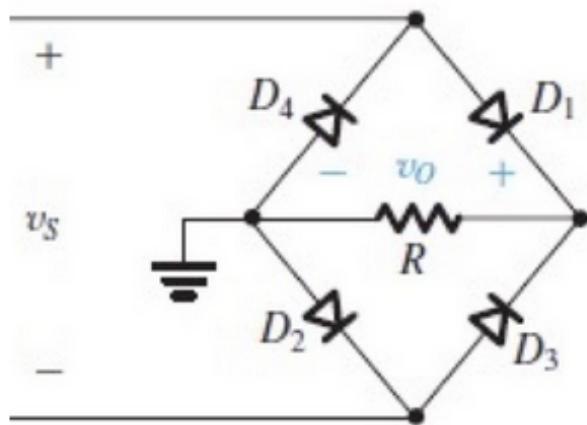
This isn't very efficient because it only uses half of the waveform, so it is desirable to build a *full wave* rectifier

Most of the homework, etc we will be practicing working from the practical circuit to the small-signal design, whereas as a designed we will usually be going the other way around.



What we want with a full-wave rectifier is to automatically interchange the wires whenever the input signal goes from negative to positive and vice-versa as to provide the load a wholly positive signal.

A full wave rectifier can be built as follows



**Figure 28.** When the signal is positive it will flow from diode 1 to the load and then to diode 2. When it is the negative polarity it will go through diode 3 to the load and then return through diode 4. Note how the direction remains the same, and how we selectively connect the ground to the top or bottom terminal depending on the input polarity.

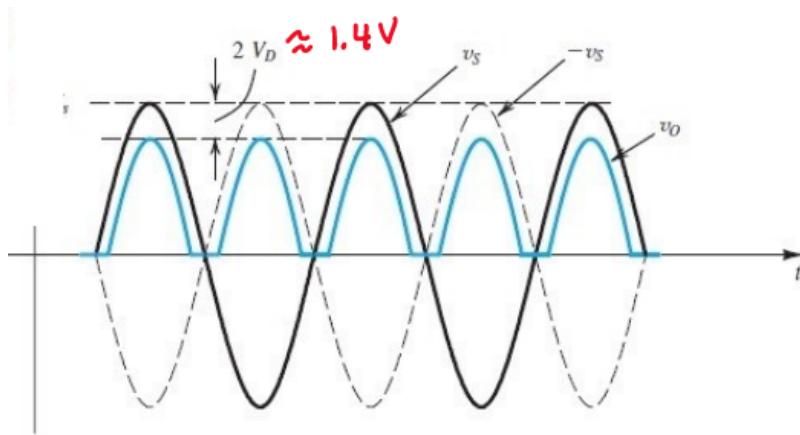


Figure 29. Comparing the two there is a little more loss but at least we get the full wave now

A **peak rectifier** uses a capacitor as an energy storage device to hold the voltage at the peak value.

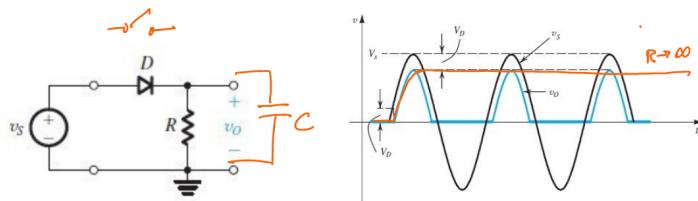
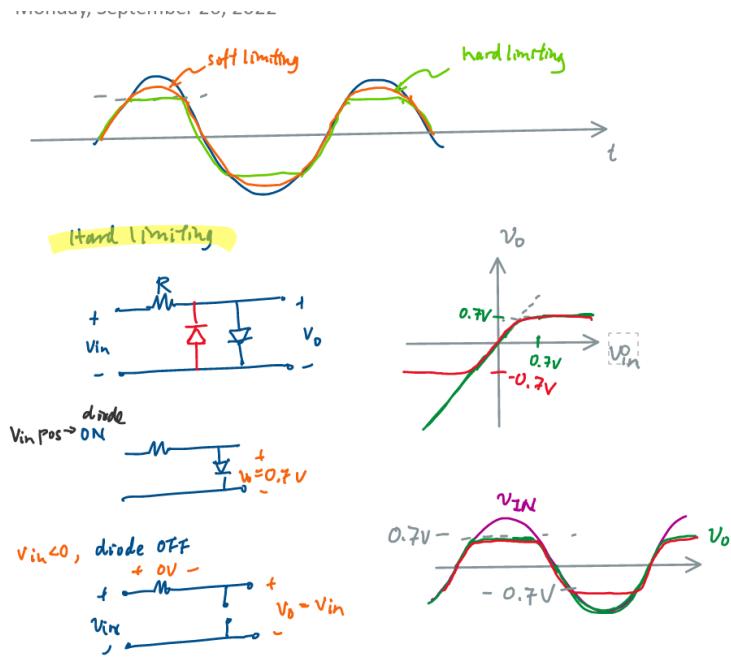


Figure 30. Capacitor size and frequency must be tuned as to avoid voltage drop (recall RC circuits. Note that the time constant  $\propto RC$ , so a bigger resistor (slower rate of discharge) or bigger capacitor (bigger energy storage) can reduce voltage drop)

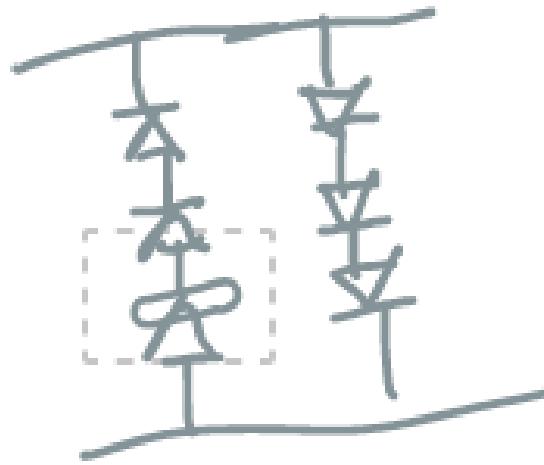
#### SUBSECTION 21.1

## Lecture 8

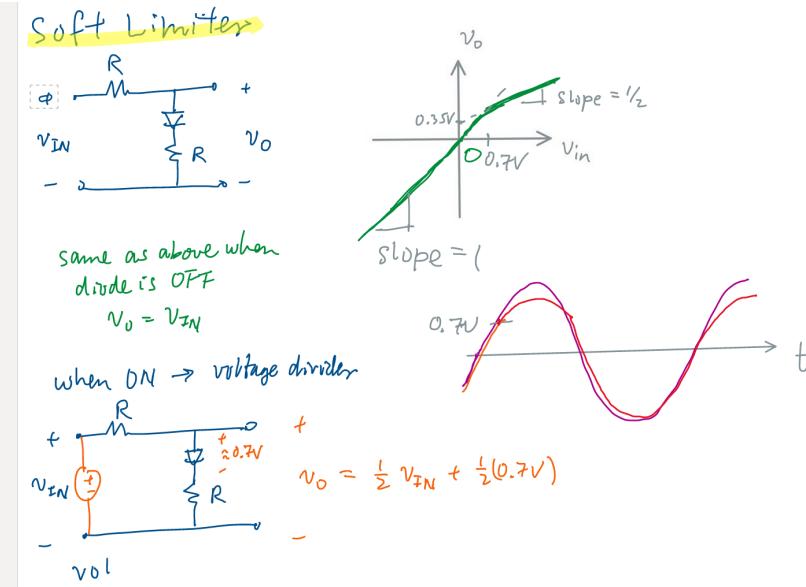
Hard limiting circuits include a pair of diodes which cause only a maximum of  $0.7V$  signal to pass through.



Increasing the hard limiting voltage can be done by putting a bunch of diodes in series, i.e.



The hard, squared-off signal response (and multiple of 0.7V) may not be desirable, so we can use a soft limiter instead.



This will act like nothing is happening when the diode is off, i.e.  $v_o = v_{in}$ . When the diode is on the soft limiting circuit acts like a voltage divider. For this circuit there are two resistors with the same  $R$  so the output voltage from just the current divider is

$$v_o = \frac{R}{R+R} v_{in} = \frac{1}{2} v_{in} \quad (21.1)$$

We will also have to account for the voltage of the diode, so

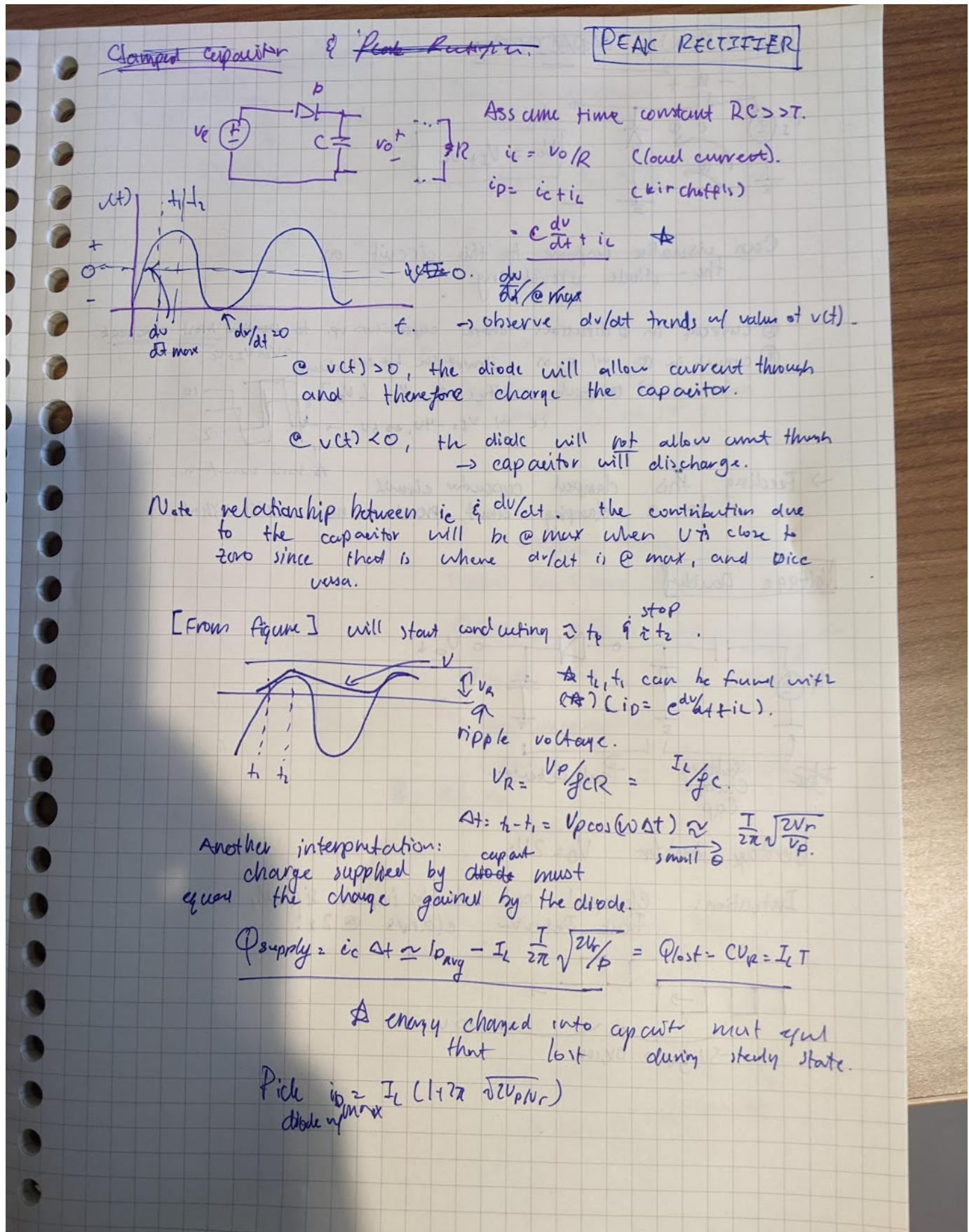
$$v_o = \frac{1}{2} v_{in} + \frac{1}{2} (v_d = 0.7V) \quad (21.2)$$

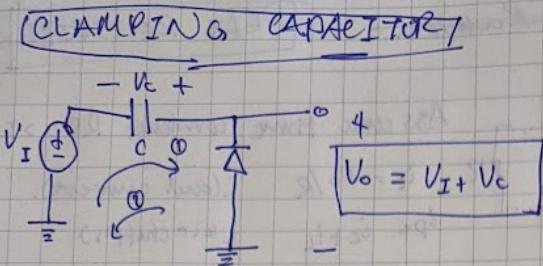
$$v_o = \frac{1}{2} v_{in} + \frac{1}{2} 0.7V \quad (21.3)$$

#### SUBSECTION 21.2

## Lecture 9

I missed this lecture, so here are my handwritten catchup notes





Can visualize response to this circuit as the diode preventing

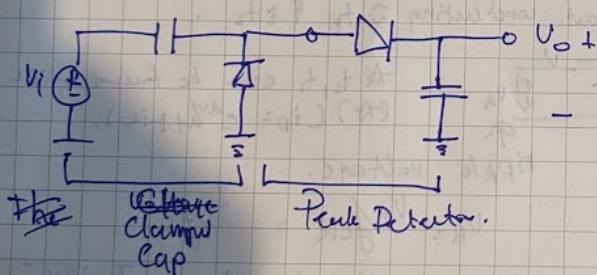
① current in  $\odot$  direction: charge capacitor up. to ~~most~~ most -ve peak.

② current in  $\odot$ : w/  $R_L \approx \infty$ , straight line but zero current.

$\rightarrow$  output voltage  $\propto V_C = L V_I$  [6-10] ie w/  $V_I = -4V \rightarrow 6V$   $\rightarrow$  same waveform.

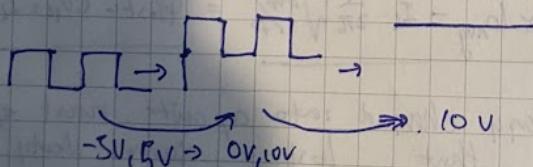
$\rightarrow$  Feeding this clamped capacitor circuit to a clamping circuit provides a nice DC signal.

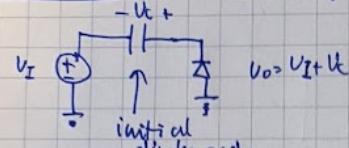
Voltage Doubler



Voltage doubler  $V_O = 2V_I$

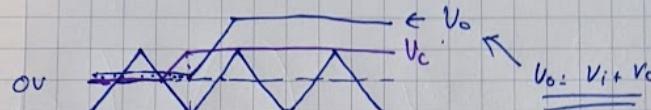
Intuition: Clamped cap doubles input signal, Peak Detector clamps @ 2x.



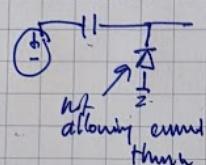
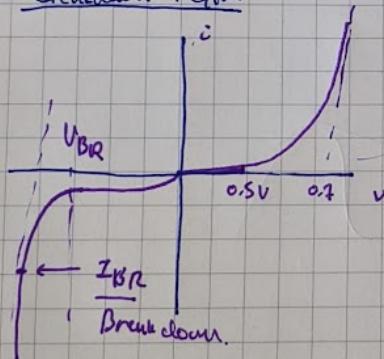
Example Clamped Analysis

Input: 5V triangular.

$$V_O = V_I + V_C = V_I + C \frac{dV}{dt}$$



diode off: ON: get value to -1ve peak.

Why does it only charge when circuit  $\rightarrow$ , not  $\leftarrow$ ?Breakdown Region"Zener Voltage"Typical  $V_{Zener} \approx 200V = V_Z$ Operation  $\approx V_Z$  is not destructive for small  $I$ 

- can be used to generate voltage.

 $V_{PC}$  large (uniquely) e.g. 1W.

$$R = \frac{V}{I} = \frac{7}{10 \text{ mA}} = 700 \Omega$$

$R \gtrsim V_R$

$V_O \approx V_Z$

Desired regulated voltage may be created by exploiting the breakdown current

$$(V_O = V_Z) + V_R = V_{DR}$$

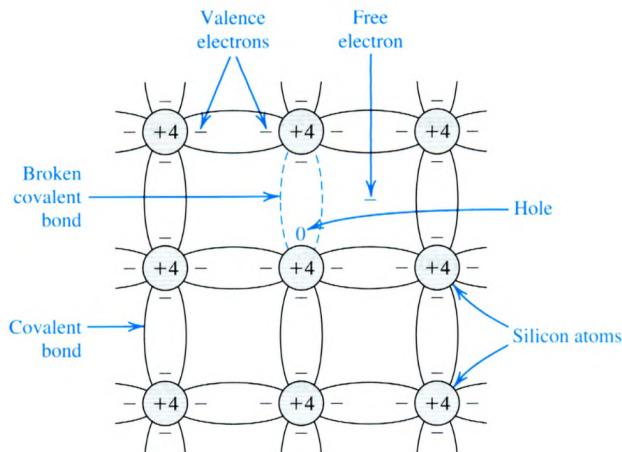
∴ pick  $V_R$  s.t.  $V_Z = V_{PC} - V_R$ 

$$\text{so pick } R = \frac{V_R}{I}$$

## SUBSECTION 21.3

**Lecture 10**

The most commonly used semiconductor in electronics is silicon. At a molecular level silicon crystals are linked together via covalent bonds which spontaneously break and reform. This process allows electrons to wander away from its parent and therefore become available to conduct electricity. The process of electrons filling some of these holes is called recombination and the process of electrons leaving the crystal to form holes is called generation. At thermal equilibrium the recombination rate is equal to the generation rate and therefore the concentration of free electrons  $n$  is equal to that of the holes  $p$ , i.e.



$$n = p = n_i \quad (21.4)$$

Where  $n_i$  denotes the number of free electrons in a unit volume of silicon at a given temperature.

**Theorem 11**

The number of free electrons and holes in a unit volume ( $cm^3$ ) of silicon at a given temperature  $T$  is given by

$$n_i = BT^{\frac{3}{2}} e^{\frac{-E_g}{2kT}} \quad (21.5)$$

Where  $k$  is the Boltzmann constant,  $B$  is a material-dependent parameter that is  $7.3 \cdot 10^{15} cm^{-3} K^{-\frac{3}{2}}$  for silicon,  $T$  is the temperature in kelvin, and  $E_g = 1.12 eV$  is the bandgap energy or the minimum energy required to break a covalent bond and create an electron-hole pair.

The intrinsic silicon crystal does not have sufficient amounts of free electrons and holes to be appreciably conductive at room temperature, plus their conductivity varies strongly with temperature. **Doping**, or introducing impurities into silicon, can be used to increase the number of free electrons and holes in the crystal. To increase the concentration of free electrons,  $n$  Silicon is commonly doped with an element with 5 valence electrons such as phosphorous. Likewise, to increase the concentration of free holes,  $p$  Silicon is commonly doped with an element with 3 valence electrons such as boron.

**Theorem 12**

For  $n$  type silicon, the concentration of free electrons is given by

$$p_n n_n = n_i^2 \Leftrightarrow p_n \approx \frac{n_i^2}{N_P} \quad (21.6)$$

For  $p$  type silicon, the concentration of free electrons is given by

$$p_p n_p = n_i^2 \Leftrightarrow p_p \approx \frac{n_i^2}{N_A} \quad (21.7)$$

Note that pieces of  $n$  or  $p$  doped silicon are electronically neutral; the charge of the majority free carriers (electrons in  $n$ , holes in  $p$ ) are neutralized by the bound charges of the impurity atoms.

There are two distinct ways that current can flow in semiconductors; drift and diffusion. **Drift current** is the movement of free electrons or holes in response to an electric field; holes are accelerated in the direction of  $E$  and free electrons opposite. Holes acquire a velocity of  $v_{p-drift} = \mu_p E$ , with  $\mu_p$  being a constant<sup>32</sup>. If we were to consider a cross section of a silicon bar with a field applied across it, the hole component of the direct current (and current density) flowing through the bar would be

$$I_{s,p} = A q p v_{p-drift} = A q p \mu_p E \Rightarrow J_p = q p \mu_p E \quad (21.8)$$

The current due to free electrons is found similarly and can be used to find the total drift current density

$$I_{s,n} = -A q n v_{n-drift} \Rightarrow J_{s,n} = q n \mu_n E \quad (21.9)$$

$$J_S = J_{s,p} + J_{s,n} = q(p\mu_o + n\mu_n)E = \frac{E}{\rho} \quad (21.10)$$

Where resistivity  $\rho$  is given by

$$\rho = \frac{1}{q(p\mu_p + n\mu_n)} \quad (21.11)$$

We now observe that (21.10) is a variant of Ohm's law and can be written as

$$\rho = \frac{E}{J} \quad (21.12)$$

**Diffusion current** is the movement of holes and electrons from areas of high concentration to low concentration in response to a gradient inside the semiconductor<sup>33</sup>.

The magnitude of the current at any point is proportional to the slope of the concentration gradient, i.e. for holes

$$J_{D,p} = -q D_p \frac{dp(s)}{dx} \quad (21.13)$$

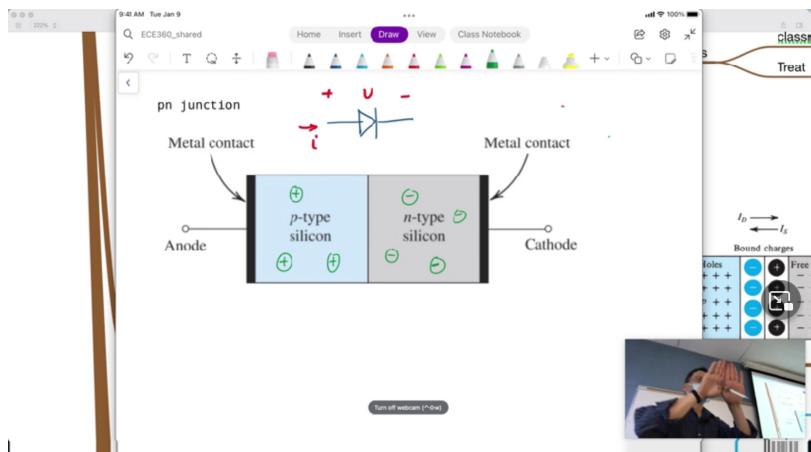
And for electrons

$$J_{D,n} = q D_n \frac{dp(s)}{dx} \quad (21.14)$$

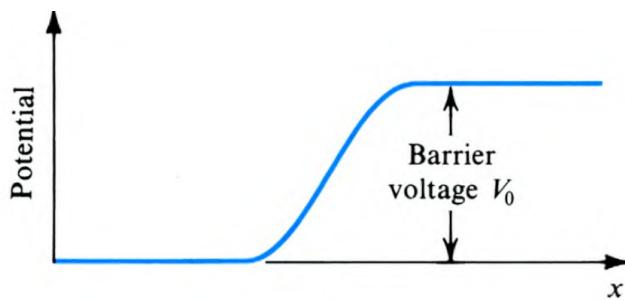
This brings us to our first example of a semiconductor device: the  $p - n$  junction diode.

<sup>32</sup>  $\mu_p = 480 \text{ cm}^2/\text{V} \cdot \text{s}$  for intrinsic silicon

<sup>33</sup> Think of dropping ink into water



Looking at a passive device without an applied voltage junction we note that free electrons from *n* side gets caught in the holes in the *p* type, leaving behind a positive charge and vice-versa causing a "depletion region" in the middle with an electric field without any holes or electrons. This causes a potential field and makes it such that work must be done to move charge from the anode to the cathode.



Reverse-biasing the diode causes an even greater potential inside the diode which causes the depletion region to grow and the electric field to increase, thereby prevent current from passing through. A forward bias would counteract the depletion region and therefore allow current to flow through the diode. Drift current<sup>34</sup> dominates in the reverse bias region, while diffusion current dominates in the forward bias region.

<sup>34</sup> recall: movement of carriers due to electric field, electrons drifting back to *n* region

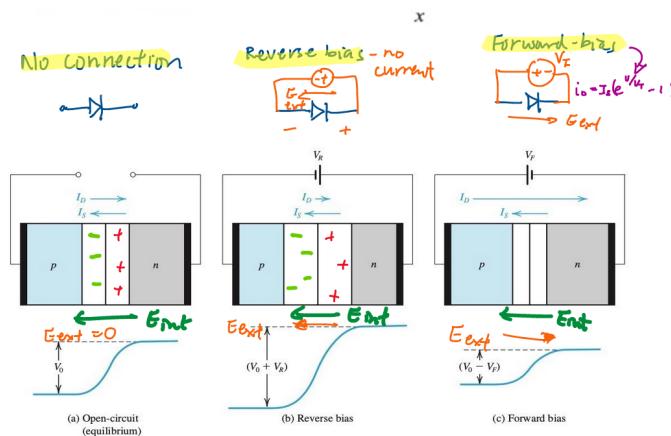


Figure 3.11 The *pn* junction in: (a) equilibrium; (b) reverse bias; (c) forward bias.

Skipping past the derivation, the expression for diode current is given by

$$i_D = I_s(e^{\frac{v_D}{v_T}} - 1) \quad (21.15)$$

This is a result of the Boltzmann distribution and the derivation is given in some Feynman lecture or in the textbook.

#### SUBSECTION 21.4

## Lecture 11: Introduction to transistors and MOSFETs

Transistors are essentially switches that can be turned on and off by applying a voltage, and are implemented in semiconductors by stacking another *p* or *n* layer onto a diode, i.e.



Figure 31. pnp vs npn transistor

There are two main types of transistors, MOSFET (metal-oxide semiconductor field-effect transistors) and BJT (bipolar junction transistor). MOSFETs have found more widespread use due to their simplicity of manufacture and their ability to be scaled down to smaller sizes.

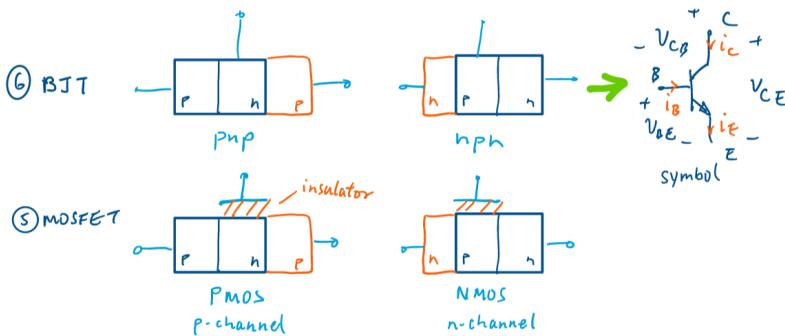


Figure 32. BJT transistors came first but are not commonly used now; they will be discussed later on in the course. Whereas BJTs directly make contact with the semiconductor, MOSFETs apply a field to induce a change in the device.

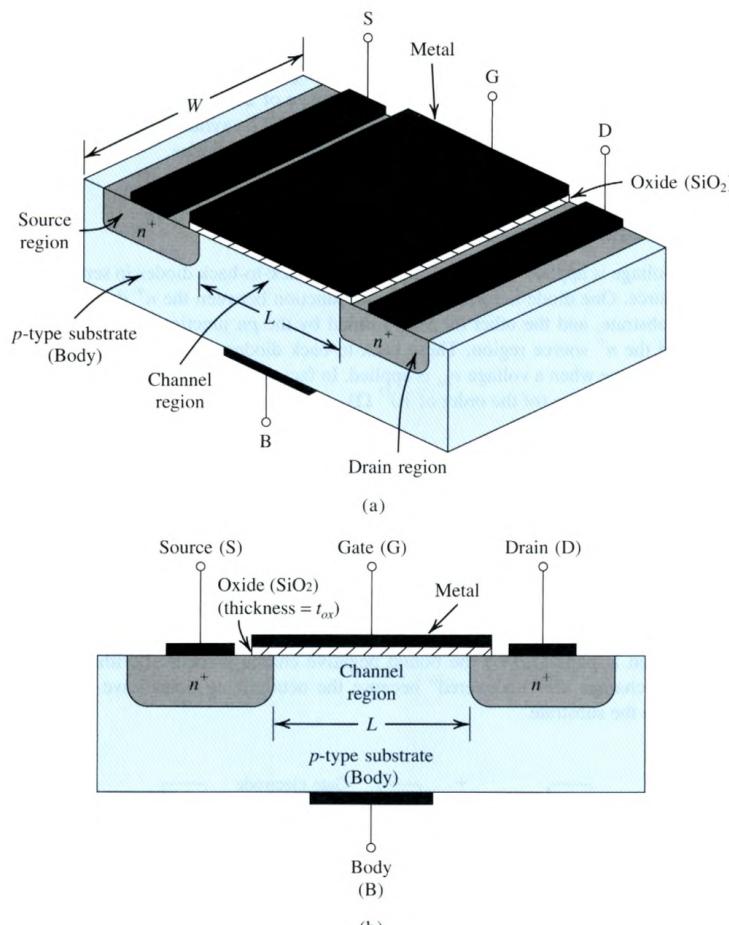


Figure 33. CMOS transistors

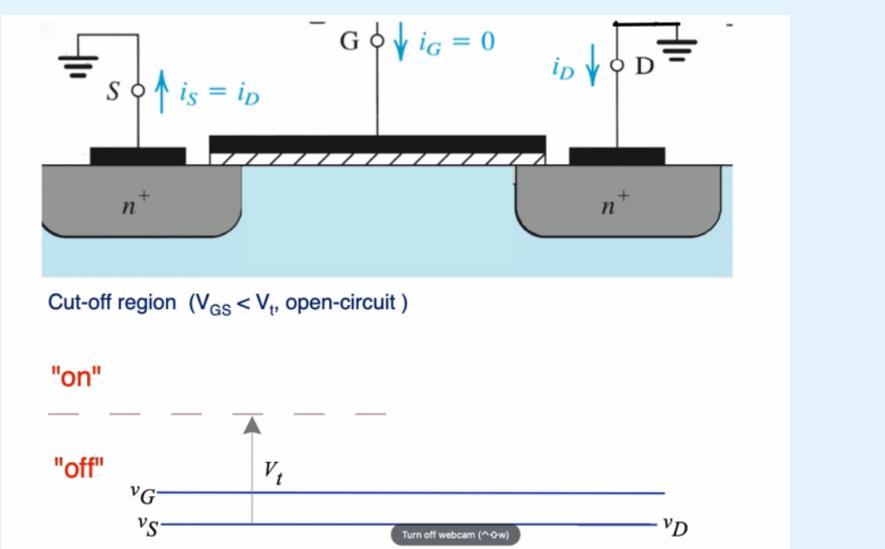
P transistors use holes as their primary charge carrier and allow current to pass if the base is negative with respect to the emitter<sup>35</sup>. Conversely, N transistors carry charge using electrons and allow current to pass if the base is positive with respect to the emitter<sup>36</sup>

Definition 48

**Cut-off region**

<sup>35</sup> For a PMOS this would be the application of a negative field

<sup>36</sup> For a NMOS this would be the application of a positive field



There is no charge if the gate voltage is the same as the source. At some point, the threshold voltage, there will be enough voltage to form a "n-channel"<sup>37</sup> through which electrons may pass.

<sup>37</sup> again, for a n-type

#### SUBSECTION 21.5

## Lecture 12: MOSFET Operating Regions

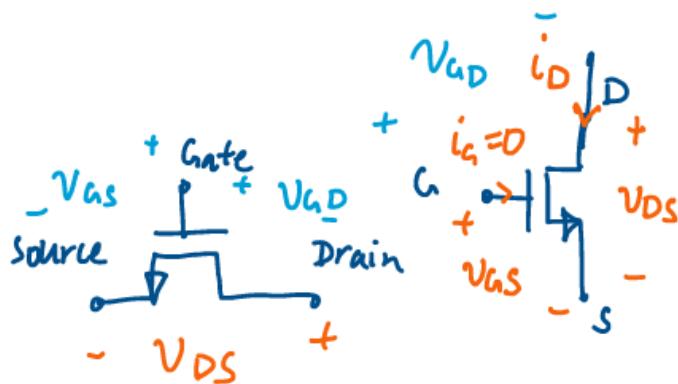


Figure 34. Labelled NMOS. The drain current can be thought of as the output.

There are three MOSFET operation regions; cut-off, triode, and saturation. Regions can be identified by inspecting the various voltages in the device.

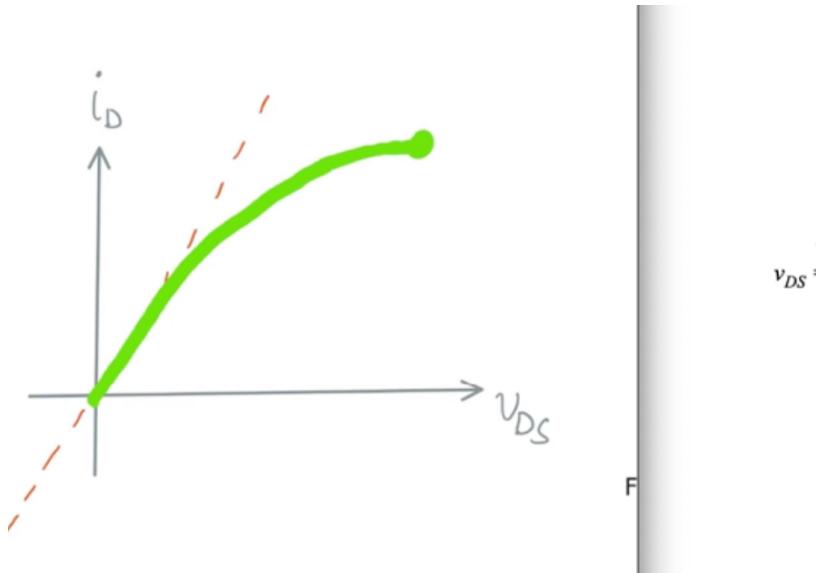
### Channel Inversion

•

**Table 6.** Channel inversion in operation regions

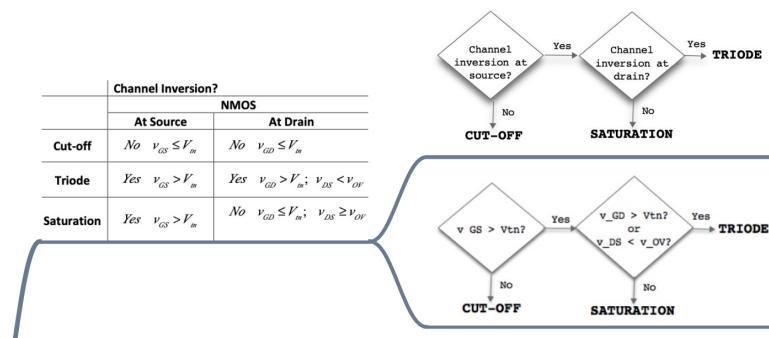
region	source	drain
cut-off	No, $V_{GS} < V_t$	No $V_{GD} < V_t$
triode	Yes, $V_{GS} > V_t$	Yes, $V_{GD} > V_t$
saturation	Yes, $V_{GS} > v_t$	No (pinched off) $V_{DS} > V_{OD}$ or $V_{GD} < V_t$

$V_t$  is the threshold voltage. Note that since this is a symmetrical device, we can compare  $V_{GS}$  and  $V_{GD}$  w.r.t the source and drain



**Figure 35.** The  $n$ -channel will thin out as the drain voltage goes up; at some point we will hit a drain voltage equal to the threshold voltage and cause the channel to disappear

When the drain voltage is equal to the threshold voltage, the overdrive voltage is equal to the  $V_{DS}$ . This is the boundary between the triode and saturation region.

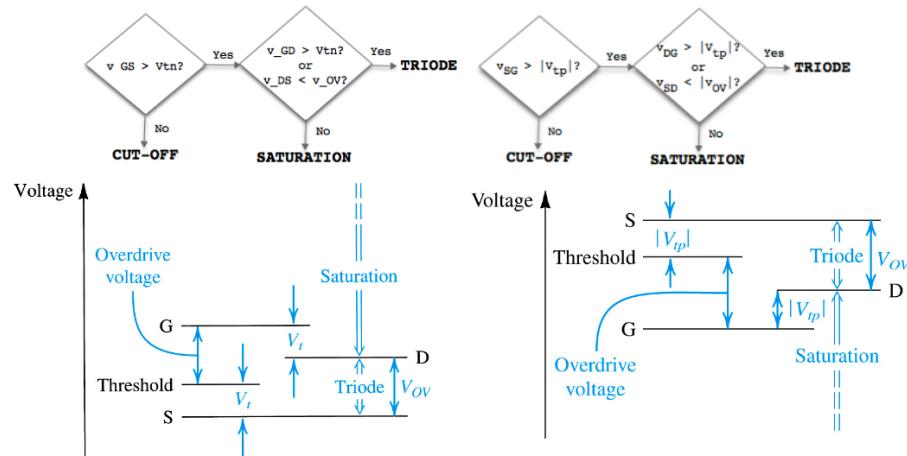


*Remark* For a NMOS the source is the terminal at the lower voltage and the drain is the terminal at the higher voltage. For a PMOS it is the opposite, i.e. the source is at higher voltage. It is conceivable that the source and drain could be swapped during operation.

## MOSFET Regions of Operation

 $v_{OV}$ : Overdrive voltage  $v_{OV} = v_{GS} - V_m$ 

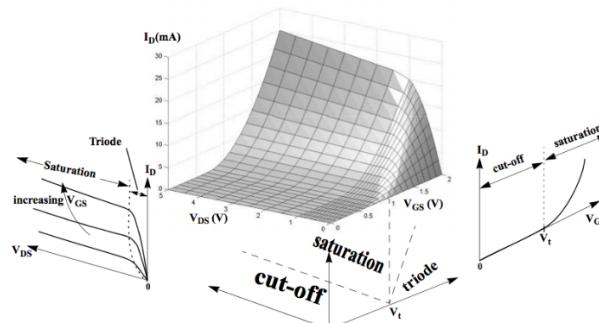
Channel Inversion?			
		NMOS	PMOS
		At Source	At Drain
<b>Cut-off</b>	No $v_{GS} \leq V_m$	No $v_{GD} \leq V_m$	No $v_{SG} \leq  V_{tp} $
<b>Triode</b>	Yes $v_{GS} > V_m$	Yes $v_{GD} > V_m$ ; $v_{DS} < v_{OV}$	Yes $v_{SG} >  V_{tp} $
<b>Saturation</b>	Yes $v_{GS} > V_m$	No $v_{GD} \leq V_m$ ; $v_{DS} \geq v_{OV}$	Yes $v_{SG} >  V_{tp} $



## MOSFET Device Equations

Operating Region	NMOS	PMOS
<b>Cut-off</b>	$i_D = 0$	$i_D = 0$
<b>Triode</b>	$i_D = \mu_n C_{ox} \left( \frac{W}{L} \right) (v_{GS} - V_m) v_{DS} - \frac{v_{DS}^2}{2}$	$i_D = \mu_p C_{ox} \left( \frac{W}{L} \right) (v_{SG} -  V_{tp} ) v_{SD} - \frac{v_{SD}^2}{2}$
<b>Saturation</b>	$i_D = \frac{1}{2} \mu_n C_{ox} \left( \frac{W}{L} \right) (v_{GS} - V_m)^2 [1 + \lambda v_{DS}]$	$i_D = \frac{1}{2} \mu_p C_{ox} \left( \frac{W}{L} \right) (v_{SG} -  V_{tp} )^2 [1 +  \lambda  v_{SD}]$

## n-Channel MOSFET Current-Voltage Characteristic



## Lecture 13: MOSFET current-voltage characteristics

The TLDR of this lecture is that the current through a MOSFET is given by those fancy big equations. A cool consequence of the MOSFET is that it acts like a variable resistor in the triode region.

PART

VI

# MAT389: Complex Analysis

## Complex Numbers

### Lecture 1

Consider a 2-vector  $\vec{x} = (x, y) \in \mathcal{R}$ . As complex numbers correspond to two-vectors

$$\vec{x} = (x, y) \leftrightarrow z = x + iy, i^2 = -1 \quad (22.1)$$

$z$  is, therefore, a complex variable. What are the benefits of a complex number like  $z$ ?

Definition 49

#### Imaginary and Complex Numbers

$i$  is an imaginary number such that

$$i^2 = -1 \quad (22.2)$$

A complex number has the form:

$$z = x + iy \quad (22.3)$$

This prof lectures at the speed of sound and talks *into* the board. Couldn't quite follow during this lecture, hopefully I get better about it in the following ones.

Definition 50

There are a number of operations we can perform on complex numbers.

#### Addition

$$z + z' = (x + x') + i(y + y') \quad (22.4)$$

#### Multiplication

$$zz' = (x + iy)(x' + iy') = (xx' - yy') + i(xy' + x'y) \quad (22.5)$$

PROOF

Proof of (22.5):

$$\begin{aligned} zz' &= (x + iy)(x' + iy') \\ &= x + ixy' + iyx' + i^2yy' \\ &= xx' - yy' + i(xy' + yx') \end{aligned} \quad (22.6)$$

□

#### Magnitude

$$|z| = \sqrt{x^2 + y^2} \quad (22.7)$$

**Conjugate**

The complex conjugate has the properties:

- $\bar{z}z = |z|^2$
- $\overline{(z + z')} = \bar{z} + \bar{z}'$
- $\overline{z \cdot z'} = \bar{z} \cdot \bar{z}'$

We can define a new operation

$$\forall \text{complex } z, \exists \text{ complementary number } w \text{ such that } zw = wz = 1 \quad (22.8)$$

Denote

$$w = \frac{1}{z} = z^{-1} \quad (22.9)$$

PROOF

Proof of (22.9): Find  $w$  s.t.  $zw = 1$

$$\begin{aligned} zw &= 1 \\ w\bar{z}z &= \bar{z}z = |z|^2 > 0 \\ |z|^2 w &= \bar{z} \\ w &= \frac{\bar{z}}{|z|^2} \rightarrow Z^{-1} = \frac{\bar{z}}{|z|^2} \end{aligned} \quad (22.10)$$

□

Furthermore, there are operators that we can define on complex numbers.

Definition 51

**Real and Imaginary Operators**

Given  $z = x + iy$ , we can define the real and imaginary operators

$$x = \operatorname{Re}\{z\} \quad (22.11)$$

$$y = \operatorname{Im}\{z\} \quad (22.12)$$

Example

$$\operatorname{Im}\left\{(1 + \sqrt{2}i)^{-1}\right\} \quad (22.13)$$

By (22.9), we have

$$\operatorname{Im}\{z^{-1}\} = \frac{-\operatorname{Im}\{z\}}{|z|^2} \quad (22.14)$$

And

$$\operatorname{Re}\{z^{-1}\} = \frac{-\operatorname{Re}\{z\}}{|z|^2} \quad (22.15)$$

Using these, for example, we find that the  $\operatorname{Im} = \frac{-\sqrt{2}}{3}$

We can get the real component in a similar way.

Here is an enumeration of absolute value properties for complex numbers:

$$|z \cdot w| = |z||w| \quad (22.16)$$

$$|z + w| \leq |z| + |w| \quad (22.17)$$

$$|\bar{z}| = |z| \quad (22.18)$$

$$|z + w|^2 = (\bar{x} + \bar{w})(z + w) = |z|^2 + |w|^2 + \bar{z}w + \bar{w}z \quad (22.19)$$

PROOF Note that  $\bar{z}w + \bar{w}z = 2\operatorname{Re}\{z\bar{w}\}$ , by (22.19)

And so

$$|z + w|^2 \leq |z|^2 + |w|^2 + 2|z||w| = (|z| + |w|)^2 \quad (22.20)$$

□

#### SUBSECTION 22.2

## Lecture 2

Whereas a two-vector  $\vec{x} \in \mathbb{Z}$ , complex numbers exist in the complex plane,  $z \in \mathbb{C}$

Theorem 13

### Polar Decomposition

Complex numbers can be expressed in polar form as well

$$z = r(\cos \theta + i \sin \theta) \quad (22.21)$$

Where

$$r = |z| \quad x = r \cos \theta \quad y = r \sin \theta \quad \theta = \tan^{-1} \left( \frac{y}{x} \right) \quad (22.22)$$

This has a number of useful properties

$$z \cdot z' = |z||z'|(\cos(\theta + \theta') + i \sin(\theta + \theta')) \quad (22.23)$$

$$\frac{z}{z'} = \frac{|z|}{|z'|}(\cos(\theta - \theta') + i \sin(\theta - \theta')) \quad (22.24)$$

PROOF Proof for (22.23):

$$\begin{aligned} z \cdot z' &= |z|(\cos(\theta + i \sin \theta)) \times |z'|(\cos \theta' + i \sin \theta') \\ &= |z||z'|(\cos \theta \cos \theta' + i \cos \theta \sin \theta' + i \sin \theta \cos \theta' - \sin \theta \sin \theta') \\ &= |z||z'|[\cos \theta \cos \theta' - \sin \theta \sin \theta' + i(\cos \theta \sin \theta' + \sin \theta \cos \theta')] \end{aligned} \quad (22.25)$$

And the proof follows

□

Lemma 3

A corollary exists

$$z^2 = |z|^2(\cos 2\theta + i \sin 2\theta) \quad (22.26)$$

Theorem 14

### Moivre's Theorem

$$z^n = |z|^n(\cos(n\theta) + i \sin(n\theta)) \quad (22.27)$$

More generally,

$$(\cos \theta + i \sin \theta)^n = \cos(n\theta) + i \sin(n\theta) \quad (22.28)$$

So we may define  $z$  to be the  $n^{\text{th}}$  root of  $w$  which implies that

**Lemma 4** Every complex number has a  $n^{\text{th}}$  root  $\forall n$

PROOF

$$\text{Let } z = |w|^{\frac{1}{n}} \left( \cos \frac{\theta}{n} + i \sin \frac{\theta}{n} \right) \quad (22.29)$$

Then

$$w = |w|(\cos \theta + i \sin \theta), \text{ then } z^n = w \quad (22.30)$$

Intuition: define  $z$  to be  $\frac{1}{n}$  and then take the  $n^{\text{th}}$  power of both sides to show that  $z^n = w$

□

This leads us to the conclusion that representations of complex numbers are not unique<sup>38</sup>.

PROOF If every  $z$  can be written as  $z = r(\cos \theta + i \sin \theta)$ , then it holds for  $\theta + 2\pi n \forall n \in \mathbb{Z}$  since  $\sin \theta = \sin(\theta + 2\pi n)$  and  $\cos \theta = \cos(\theta + 2\pi n)$ . □

### 22.2.1 Functions on complex planes

**Definition 52**

Given a domain  $\mathbb{D} \in \mathbb{C}$ , a function  $f$  is a rule such that

$$z \in \mathbb{D} \xrightarrow{f} w \in \mathbb{D} \leftrightarrow w = f(z) \quad (22.31)$$

**Definition 53**

We may define  $\mathbb{D}$  to be the domain of  $f$

Likewise, range is defined as

$$\text{Ran}\{f\} = \{w \in \mathbb{C} : \exists z \in \mathbb{D} : f(z) = w\} \quad (22.32)$$

*Example*

$$f(z) = \frac{1}{z+i} \quad (22.33)$$

What is the maximum domain of  $f$ ?

$$\text{Dom}\{f\} = \{z \in \mathbb{C} : |z| < -i\} \quad (22.34)$$

What is the range of  $f$ ?

$$\frac{1}{z+i} = w \quad (22.35)$$

For which values of  $w$  can we solve this equation?

$$z = -i + \frac{1}{w} \quad (22.36)$$

So the range of the function is

$$\text{Ran}\{f\} = \{w \in \mathbb{C} : |w| \neq 0\} \quad (22.37)$$

*Example*

$$f(z) = z^2 + 1 \quad (22.38)$$

It is fairly clear that  $\text{Dom}\{f\} = \mathbb{C}$

The range can be found by solving for  $z$  in

$$z^2 + 1 = w \quad (22.39)$$

And so

$$Ran\{f\} = \{w \in \mathbb{C}\} \quad (22.40)$$

### 22.2.2 Exponential Functions

**Definition 54**

Given  $z = x + iy$ ,

$$e^z = e^x(\cos y + i \sin y) = e^{Re\{z\}}(\cos(Im\{z\}) + i \sin(Im\{z\})) \quad (22.41)$$

$$1. e^{z+w} = e^z e^w$$

$$2. |e^z| = e^{Re\{z\}} \neq 0$$

$$3. e^{z+i2\pi n} = e^z$$

**PROOF** (1) follows from the product rule for complex numbers

(2) follows by definition

(3) follows by definition (recall: writing  $z$  w.r.t. sin, cos)  $\square$

More properties:

- $Dom\{e^z\} = \mathbb{C}$
- $Ran\{e^z\} = \{\mathbb{C} \setminus \{0\}\}$
- $e^z = w \quad \text{if } w \neq 0$

39

arg, or argument is the angle from the real axis to that on the complex plane. Usually denoted by  $\theta$

<sup>39</sup> Note: ‘\’ denotes set exclusion

$$\begin{aligned} z &= \ln|w| + i \arg w \\ e^z &= e^{\ln|w| + i \arg w} \\ &= e^{\ln|w|} e^{i \arg w} \\ &= |w| \cos(\arg w) + i \sin(\arg w) \\ &= w \end{aligned} \quad (22.42)$$

**Remark** **Polar representation**

$$w = |w| e^{i \arg w} \quad (22.43)$$

**Example** **Find polar coordinates of**  $z = i + 1$

$$r = |w| \quad \theta = \arg w \quad (22.44)$$

$$\begin{aligned} |z| &= \sqrt{1+i} = \sqrt{2} \\ \cos \theta &= \frac{1}{\sqrt{2}} \rightarrow \theta = \frac{\pi}{4} \\ z &= \sqrt{2} e^{i\pi/4} \end{aligned} \quad (22.45)$$

**Example** **Find**

$$(1+i)^{\frac{1}{3}} \quad (22.46)$$

| Solution:  $z = \sqrt{2}e^{\frac{i\pi}{4}} \rightarrow z^{1/3} = 2^{\frac{1}{6}}e^{i\pi/12}$

**Definition 55** **Trig functions for complex numbers**

$$\cos z = \frac{1}{2}(e^{iz} + e^{-iz}) \quad (22.47)$$

PROOF

$$\cos x = \frac{1}{2}(e^{ix} + e^{-ix}) = \frac{1}{2} \left( \cos x + i \sin x + \underbrace{\cos(-x)}_{\text{odd; } = \cos(x)} + \underbrace{i \sin(-x)}_{\text{even; } = -\sin(x)} \right) = \cos x \quad (22.48)$$

□

$$\sin z = \frac{1}{2}(e^{iz} - e^{-iz}) \quad (22.49)$$

And a similar proof follows for  $\sin z$ .

These have the following properties

$$\sin z|_{Im Z=0} = \sin x \quad (22.50)$$

$$\cos(z + 2\pi n) = \cos z \forall n \in \mathbb{Z} \quad (22.51)$$

$$\sin(z + 2\pi n) = \sin z \forall n \in \mathbb{Z} \quad (22.52)$$

PROOF

$$\begin{aligned} \cos z + 2\pi n &= \frac{1}{2}(e^{i(z+2\pi n)} + e^{-i(z+2\pi n)}) \\ &= \frac{1}{2}(e^{iz}e^{i2\pi n} + e^{-iz}e^{-i2\pi n}) \\ &= \frac{1}{2}(e^{iz} + e^{-iz}) \\ &= \cos z \end{aligned} \quad (22.53)$$

□

The domain of  $\{\cos z, \sin z\} = \mathbb{C}$

Range?

Solve  $\cos z = w$  for  $z$

$$\begin{aligned} \frac{1}{2}(e^{iz} + e^{-iz}) &= w \\ \dots \times 2e^{iz} \text{ on both sides} \\ e^{2iz} - 2we^{iz} + 1 &= 0 \\ \dots \text{Let } S = e^{iz} \\ S^2 - 2ws + 1 &= 0 \\ S = w \pm \sqrt{w^2 - 1} &\equiv u \end{aligned} \quad (22.54)$$

Now we note that  $e^{iz} = u$  can be solved for  $z$  for any  $u \neq 0$

$$u = 0 \leftrightarrow w = \pm\sqrt{w^2 - 1} \quad (22.55)$$

$$w^2 = w^2 - 1 \text{ impossible for } u \neq 0 \quad (22.56)$$

Therefore:

$$\text{Ran}\{\cos z\} = \text{Ran}\{\sin z\} = \mathbb{C} \quad (22.57)$$

*Remark* An intuitive way of interpreting this result is thinking of  $\{\sin, \cos\}$  being a function that projects values from the complex domain to the real plane; though  $\{\sin, \cos\}$  takes on a limited range of values in the real domain, in the complex domain it spans the entire plane. Think: mental model of a complex number spinning around and having that project onto a real line. More formally, see: the [Little Picard Theorem](#)

#### SUBSECTION 22.3

## Lecture 3: Exponent and Logarithm

### 22.3.1 Exponential

Recall: the complex exponential function  $\exp$  is defined as

$$\exp : e^z = e^x(\cos y + i \sin y) \quad (22.58)$$

Where  $z = x + iy$ .

Properties:

1.  $e^{w+z} = e^z e^w$
2.  $e^z \neq 0$
3.  $e^{2\pi mi} = 1$

The first and third properties imply that the exponential function is a periodic function.

$$e^{z+2\pi mi} = e^z \quad (22.59)$$

Consider the equation

$$e^z = w \quad (22.60)$$

If this has the solution  $z_*$ , then  $z_* + 2\pi mi, m = 0, \pm 1, \pm 2, \dots$  is also a solution.

### 22.3.2 Logarithm

**Definition 56**

$$\log \equiv \log w = \ln |w| + i \arg w \quad (22.61)$$

$$w \neq 0$$

**PROOF** Proof that  $\log w = \ln |w| + i \arg w$

$$\begin{aligned} w &= |w|e^{i \arg w} \\ &= e^{\ln |w|} e^{i \arg w} \\ &= e^{\ln |w| + i \arg w} \\ \rightarrow e^z &= e^{\ln |w| + i \arg w} \\ \Rightarrow z &= \ln |w| + i \arg w \end{aligned} \quad (22.62)$$

□

Note that  $\arg$  is a multivalued function, i.e.

$$\arg w = \arg(w + 2\pi m_i), i \in \mathbb{Z}, \arg w \in [-\pi, \pi) \quad (22.63)$$

Example

$$e^z = e^5 \quad (22.64)$$

Solve for  $z$

$$z = 5 + 2\pi m i, m \in \mathbb{Z} \quad (22.65)$$

Example

Solve  $e^z = i$

$$i = e^{\frac{i\pi}{2}} \quad (22.66)$$

The solution is therefore

$$z = i(\pi/2 + 2\pi m), m \in \mathbb{Z} \quad (22.68)$$

Note that providing only a single solution is wrong; must provide all

$$|i| = 1; \arg i = \frac{\pi}{2} \quad (22.67)$$

The complex logarithm is a multivalued function (like  $\arg$ ).

$$\log z = \ln |z| + i \arg z, \arg z \in [-\pi, \pi) \quad (22.69)$$

Note that  $i \arg z$  denotes the principal branch of  $\log$ .

Though it is multivalued, in general,  $\log zw \neq \log z + \log w$

Example

Assume  $\arg z = \frac{2\pi}{3}$  and  $\arg w = \frac{3\pi}{4}$ .

$$\arg(zw) = ? \quad (22.70)$$

Typically we would just add them together, i.e.  $\frac{2\pi}{3} + \frac{3\pi}{4}$ . But this is  $> \pi$  which is not allowed as per the definition of  $\arg$ , so we must add or subtract something.

Let's try subtracting  $2\pi$ <sup>40</sup>

<sup>40</sup>since adding  $\pm 2\pi$  doesn't change the angle, just rotates it around once

$$\arg(zw) = \frac{17\pi}{12} - 2\pi = -\frac{7\pi}{12} \in [-\pi, \pi) \quad (22.71)$$

So we proved that in general the arguments don't sum up. But we want to go from here to proving that the logs don't sum up.

$$\begin{aligned} \log zw &= \ln |zw| + i \arg zw \\ &\neq \ln |z| + \ln |w| + i \arg z + i \arg w \end{aligned} \quad (22.72)$$

In general this is not correct because after breaking apart  $\arg zw$   $\arg z$  and  $\arg w$  when summed can exceed the range allowable for  $\arg$

$$\therefore \log(zw) = \log z + \log w \quad (22.73)$$

Example

Compute  $\log(\sqrt{3} + i)$ .  
Just apply the formula.

$$\log w = \ln |w| + i \arg w \quad (22.74)$$

$$\log(\sqrt{3} + i) = \ln \sqrt{4} + i \arg(\sqrt{3} + i) = \ln 2 + i\left(\frac{\pi}{6} + 2\pi m\right), m \in \mathbb{Z} \quad (22.75)$$

## 22.3.3 Powers

Definition 57

$$\forall a \neq 0, a^z \equiv e^{z \log a} \quad (22.76)$$

Example Complete  $(1+i)^i$ 

$$\begin{aligned} (1+i)^i &= e^{i \log(1+i)} \\ \log(1+i) &= \ln \sqrt{2} + i\left(\frac{\pi}{4} + 2\pi m\right), m \in \mathbb{Z} \\ \dots &= e^{-\frac{\pi}{4} - 2\pi m} e^{i \ln \sqrt{2}} \end{aligned} \quad (22.77)$$

SUBSECTION 22.4

## Lecture 4

Definition 58

## Analytical Functions

A complex function is a function that maps a complex variable to a complex result. A complex *analytic* function does the same thing, *and* is continuously differentiable over  $\mathbb{C}$

Define  $\mathbb{D}$ , an open and connected subset of the complex domain  $\mathbb{C}$

We define

$$f : \mathbb{D} \rightarrow \mathbb{C} \quad \text{to be the analytic of } z_0 \in \mathbb{D} \quad (22.78)$$

Now, given  $f$ , we can define the complex derivative  $f'$

$$f'(z_0) = \lim_{z \rightarrow z_0} \frac{f(z) - f(z_0)}{z - z_0} \quad \text{exists} \quad (22.79)$$

Noting that

$$z \rightarrow z_0 \leftrightarrow |z - z_0| \rightarrow 0 \quad (22.80)$$

(22.79) can be rewritten as

$$f'(z_0) = \lim_{h \rightarrow 0} \frac{1}{h} (f(z_0 + h) - f(z_0)) \quad (22.81)$$

$$h = z - z_0; z = z_0 + h$$

Example

Find the complex derivative

$$f(z) = z^n \quad (22.82)$$

PROOF

$$\begin{aligned} f'(z) &= \lim_{h \rightarrow 0} \frac{1}{h} ((z+h)^n - z^n) \\ &= \lim_{h \rightarrow 0} \frac{1}{h} \left( z^n + nz^{n-1}h + \binom{n}{2} z^{n-2}h^2 + \dots + h^n - z^n \right) \\ &= \lim_{h \rightarrow 0} \frac{1}{h} \left( nz^{n-1} + \binom{n}{2} z^{n-2}h + \dots + h^{n-1} \right) \end{aligned} \quad (22.83)$$

... Cancel out terms that go to 0

$$\begin{aligned} &= (z^n)' \\ &= nz^{n-1} \end{aligned}$$

□

$$z = x + iy, \bar{z} = x - iy$$

Example | Find the complex derivative

$$f(z) = \bar{z} \quad (22.84)$$

PROOF

$$\begin{aligned} f'(z) &= \lim_{h \rightarrow 0} (\bar{z} + \bar{h} - \bar{z}) \\ &= \lim_{h \rightarrow 0} \frac{\bar{h}}{h} \end{aligned} \quad (22.85)$$

We then take the limit along the real and imaginary axis separately  
Real:

$$\lim_{h \rightarrow 0} \frac{h_1}{h_1} = 1 \quad (22.86)$$

Imaginary:

$$\lim_{h \rightarrow 0} \frac{-h_2}{h_2} = -1 \quad (22.87)$$

□

So the limit (22.85) does not exist

In the previous two examples we found that  $z^n$  is analytic and  $\bar{z}$  is not.

Example |

$$f(z) = e^z \quad (22.88)$$

PROOF

$$\begin{aligned}
 f'(z) &= f(z+h) - f(z) \\
 &= e^{z+h} - e^z \\
 &= e^{x+h_1}(\cos(y+h_2) + i\sin(y+h_2)) - e^x(\cos y + i\sin y)
 \end{aligned} \tag{22.89}$$

A Taylor series can be used to expand  $e^{x+h_1}$ ,  $\cos(y+h_2)$ ,  $\sin(y+h_2)$

$$\begin{aligned}
 e^{x+h_1} \cos(y+h_2) \times & \\
 \left( e^x + e^x \frac{h_1}{1!} + \text{higher order terms} \right) \times & \\
 \left( \cos y - \frac{\sin y}{1!} h_2 + \text{higher order terms} \right)
 \end{aligned} \tag{22.90}$$

(22.91)

And then a bunch of terms can be cancelled out to leave us with a couple of terms and a bunch of higher order terms in  $h_1, h_2$

$$= e^x \cos y - e^x \sin y + e^x h_1 \cos y + \dots \text{higher order terms} \tag{22.92}$$

And as a result

$$(e^z)' = \lim_{h \rightarrow 0} \frac{1}{h} e^z h = e^z \tag{22.93}$$

□

So  $e^z$  is analytic.

#### 22.4.1 Properties of complex derivative

1.  $(f+g)' = f' + g'$
2.  $(fg)' = f'g + fg'$
3.  $\frac{f'}{g} = \frac{f'g - fg'}{g^2} \cdot f(g(z))' = f'(g(z))g'(z)$

For the 3rd case here, range of  $g \in \text{domain of } f$

Example

$$(e^{z^3})' = e^{z^3} (z^3)' = 3z^2 e^{z^3} \tag{22.94}$$

Definition 59

A function  $f$  is **entire** if  $f$  is analytic in  $\mathbb{C}$

Examples of entire functions include  $e^z$ ,  $z^n$ . Non-analytic functions include  $\frac{1}{z}$  since it is not defined at 0 i.e. it is not entire over  $\mathbb{C}$

However, is  $\frac{1}{z}$  analytic over the rest of  $\mathbb{C}$ ?

PROOF

$$\begin{aligned}
 \frac{1'}{z} &= \lim \frac{1}{h} \left( \frac{1}{z+h} - \frac{1}{z} \right) \\
 &= \lim \frac{1}{h} - \frac{h}{(z+h)z} \\
 &= -\frac{1}{z^2} - \lim -\frac{h}{(z+h)z^2} = -\frac{1}{z}
 \end{aligned} \tag{22.95}$$

Therefore  $\frac{1}{z}$  is analytic in  $\mathbb{C} - \{0\}$

□

**Theorem 15****Cauchy-Riemann equations**

The Cauchy-Riemann equations give us a direct way of checking if a function is differentiable and if it is, it gives us the derivative. It is a consequence of the fact that the limit defining  $f(z)$  must be the same no matter what direction  $z$  is approached. Namely, if  $f$  as defined below is analytic<sup>41</sup>

$$f(z) = u + iv \quad (22.96)$$

Then,

$$f'(z) = \frac{\partial u}{\partial x} + i \frac{\partial v}{\partial x} = \frac{\partial v}{\partial y} - i \frac{\partial u}{\partial y} \quad (22.97)$$

In particular we're interested in the this set of PDEs (which is called the Cauchy-Riemann equations)

$$\begin{aligned} \frac{\partial u}{\partial x} &= \frac{\partial v}{\partial y} \\ \frac{\partial u}{\partial y} &= -\frac{\partial v}{\partial x} \end{aligned} \quad (22.98)$$

The short form is as follows

$$u_x = v_y \quad u_y = -v_x \quad (22.99)$$

If  $f = u + iv$  and in  $\mathbb{D}$ , then  $u, v$  satisfy

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y} \quad (22.100)$$

And

$$\frac{\partial u}{\partial y} = -\frac{\partial v}{\partial x} \quad (22.101)$$

<sup>41</sup>Complex differentiable

PROOF

Since  $f$  is analytic, then

$$f'(z) = \lim_{h \rightarrow 0} \frac{1}{h} (f(z+h) - f(z)) \quad (22.102)$$

Which exists and is independent of the way  $h \rightarrow 0$ .

Take the limits of the real and imaginary parts of (22.102):

Real:

$$\lim_{h_1 \rightarrow 0, h_1 \in \mathbb{R}} \frac{1}{h_1} (f(z+h_1) - f(z)) = \lim \frac{1}{h_1} (f(x+h_1+iy) - f(x+iy)) = \partial_x f(z) \quad (22.103)$$

$$\lim_{ih_2 \rightarrow 0, h_2 \in \mathbb{R}} \frac{1}{h_2} (f(z+ih_2) - f(z)) = \lim \frac{1}{ih_2} (f(x+ih_2+iy) - f(x+iy)) = -\partial_y f(z) \quad (22.104)$$

Since this limit is independent of how  $h \rightarrow 0$ ,

$$\partial_x f(z) = -i \partial_y f(z) \quad (22.105)$$

Recall that  $f = u + iv$

$$\partial_x(u+iv) = -i \partial_y(u+iv) = -i \partial_y u + \partial_y v \quad (22.106)$$

Therefore

$$\partial_x u = \partial_y v, \partial_x v = -\partial_y u \quad (22.107)$$

□

Definition 60

Complex derivative:

Using the Cauchy-Riemann equations, we can define the complex derivative of  $f$  as

$$\frac{\partial f}{\partial z} \frac{1}{2} \left( \frac{\partial f}{\partial x} - i \frac{\partial f}{\partial y} \right) \quad (22.108)$$

$$\frac{\partial f}{\partial \bar{z}} \frac{1}{2} \left( \frac{\partial f}{\partial x} - i \frac{\partial f}{\partial y} \right) \quad (22.109)$$

Then, via the Cauchy-Riemann equations, we have

$$\frac{\partial f}{\partial \bar{z}} = 0 \quad (22.110)$$

PROOF

Proof of (22.110):

Recall  $f = u + iv$

Plug this into the LHS of the expression:

$$\frac{\partial u}{\partial x} + i \left( \frac{\partial v}{\partial y} \right) + i \left( \frac{\partial u}{\partial x} + i \frac{\partial v}{\partial y} \right) \Rightarrow \partial_x u - \partial_y v = 0, \partial_y u + \partial_x v = 0 \quad (22.111)$$

Which is the Cauchy-Riemann equations (22.107)

□

Example

Is  $f(z) = e^{z^5} \cdot \sin z \cdot \bar{z}^3$  analytic?

$$\frac{\partial f}{\partial z} e^{z^5} \cdot \sin z e^{\bar{z}^2} \neq 0 \quad (22.112)$$

| So  $f$  is not analytic

Example | Is  $f(z) = |z|^6$  analytic?

PROOF

$$\frac{\partial y}{\partial \bar{z}} = \frac{\partial}{\partial z} (z\bar{z})^3 = z^3 \cdot 3\bar{z}^2 \neq 0 \quad (22.113)$$

□

|  $f$  is not analytic.

Now that we know that analytic functions satisfy the Cauchy-Riemann equations, we can use this to prove that the converse holds, i.e. that non-analytic functions do not satisfy [the Cauchy-Riemann equations]

Theorem 16

Given  $f(x, y)$  continuously differentiable and satisfies the Cauchy-Riemann equations, then  $f$  is analytical

PROOF

$$\begin{aligned} f(x + h_1, y + h_2) - f(x, y) &\xrightarrow{\text{Taylor series}} \\ &= f(x, y) + \partial_x f(x, y)h_1 + \partial_y f(x, y)h_2 + \text{H.O.T} - f(x, y) \\ &\dots \text{cancel terms} \dots \\ &= \partial_x = \frac{\partial f}{\partial z} + \frac{\partial f}{\partial \bar{z}} \\ &= \partial_y = \frac{\partial f}{\partial \bar{z}} - \frac{\partial f}{\partial z} \frac{1}{i} \end{aligned} \quad (22.114)$$

Note abbreviation higher order terms  $\Leftrightarrow$  H.O.T

And then plugging this back into the initial expression we get

$$\begin{aligned} f(x + h_1, y + h_2) - f(x, y) &= \left( \frac{\partial f}{\partial z} \frac{\partial f}{\partial \bar{z}} \right) h_1 + \left( \frac{\partial f}{\partial \bar{z}} \frac{\partial f}{\partial z} \right) \frac{1}{i} h_2 + \text{H.O.T} \\ &= \frac{\partial f}{\partial z} (h_1 + ih_2) + \text{H.O.T} \\ &= \frac{\partial f}{\partial z} h + \text{H.O.T} \end{aligned} \quad (22.115)$$

$$\begin{aligned} f'(z) &= \lim_{h \rightarrow 0} \frac{1}{h} \left( \frac{\partial f}{\partial z} h + \text{H.O.T} \right) \\ &= \frac{\partial f}{\partial z} z \text{ exists} \end{aligned}$$

So therefore  $f$  is analytic

□

Example | Is  $\sin z$  analytic?

$$\frac{\partial}{\partial \bar{z}} \sin z = 0 \rightarrow \sin z \text{ is analytic} \quad (22.116)$$

| In 'slow motion',

$$\begin{aligned}
 \frac{\partial \sin z}{\partial \bar{z}} &= \frac{1}{2} \left( \frac{\partial \sin z}{\partial x} + i \frac{\partial \sin z}{\partial y} \right) \\
 \sin z &= \frac{1}{2i} (e^{iz} - e^{-iz}) \\
 e^{iz} &= e^{ix-y} = e^{-y} (\cos x + i \sin y) \\
 \frac{\partial e^{iz}}{\partial \bar{z}} &= \frac{1}{2} (\partial_x + i \partial_y) \\
 &= \dots = 0
 \end{aligned} \tag{22.117}$$

Therefore  $\sin z$  is analytic

TLDR of the lecture; one can find if the function is analytic by checking if the Cauchy-Riemann equations hold. This can be done by taking the complex derivative of the function w.r.t  $z$  and  $\bar{z}$ . Then, by the theorem we proved, if  $\frac{\partial f}{\partial \bar{z}} = 0$ , then  $f$  is analytic and  $f'(z) = \frac{\partial f}{\partial z}$

#### SUBSECTION 22.5

## Power series

**Definition 61**

A **Power series** is an expression of the form

$$\sum_{n=0}^{\infty} a_n (z - z_0)^n \tag{22.118}$$

Where  $a_n$  is a coefficient and  $z_0$  the centre of the series. The power series diverges if it is not converging absolutely, which it does at  $z_*$  if

$$\sum |a_n| (z_* - z_0)^n \tag{22.119}$$

converges

**Theorem 17**

There exists radius of convergence  $R \geq 0$  such that

- The series converges absolutely in the disk  $|z - z_0| < R$
- The series diverges for  $|z - z_0| \geq R$

If

$$\lim_{n \rightarrow \infty} |a_n|^{1/n} \tag{22.120}$$

exists, then

$$\frac{1}{R} = \lim_{n \rightarrow \infty} |a_n|^{1/n} \tag{22.121}$$

**Lemma 5**

If  $\lim |a_{n+1}/a_n|$  exists, then

$$\frac{1}{R} = \lim_{n \rightarrow \infty} |a_{n+1}/a_n| \tag{22.122}$$

*Example*

Find the radius of convergence for

$$\sum_{n=0}^{\infty} \sqrt{n} z^n \tag{22.123}$$

PROOF | Let  $z_0 = 0, a_n = \sqrt{n}$

$$\begin{aligned} \lim_{n \rightarrow \infty} \sqrt{n}^{\frac{1}{n}} &= \lim_{n \rightarrow \infty} n^{1/2n} \\ \ln n^{1/2n} &= \frac{1}{2n} \ln(n) \text{ which } \rightarrow 0 \text{ as } n \rightarrow \infty \\ n^{\frac{1}{2n}} &\rightarrow e^0 = 1 \Rightarrow R = \frac{1}{1} = 1 \end{aligned} \quad (22.124)$$

□

**Theorem 18** If  $\sum a_n z^n$  and  $\sum b_n z^n$  have radius of convergence of at least  $R$

1.

$$\sum a_n z^n + \sum b_n z^n = \sum (a_n + b_n) z^n \quad (22.125)$$

and has radius of convergence of at least  $R$

2.

$$\sum a_n z^n \cdot \sum b_n z^n \quad (22.126)$$

has radius of convergence equal to  $R$

3.

$$\sum a_n z^n \quad (22.127)$$

is complex differentiable (and therefore analytic) in  $|z| < R$  and it's derivative is

$$\sum a_n n z^{n-1} \quad (22.128)$$

with radius of convergence  $R$

4.

$$a_n = \frac{1}{n! f^n|_0} \quad (22.129)$$

where

$$f(z) = \sum_{n=0}^{\infty} a_n z^n \quad (22.130)$$

SUBSECTION 22.6

## Lecture 5

PART

VII

# ECE444: Software Engineering

SECTION 23

## Preliminary

Taught by Prof. Shurui Zhou

SUBSECTION 23.1

### Lecture 1, 2

- Software engineering is different from what coding is; design, architecture, documentation, testing, etc v.s. just script kiddie-ing
- *Vasa syndrome*
- Rockstar engineers are a myth

## SECTION 24

## Project Management

## SUBSECTION 24.1

### Lecture 3

**Definition 62**

Conway's law states that 'Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure'.

The waterfall method is slow and costly and defects can be extremely costly, especially early on in the development lifecycle.

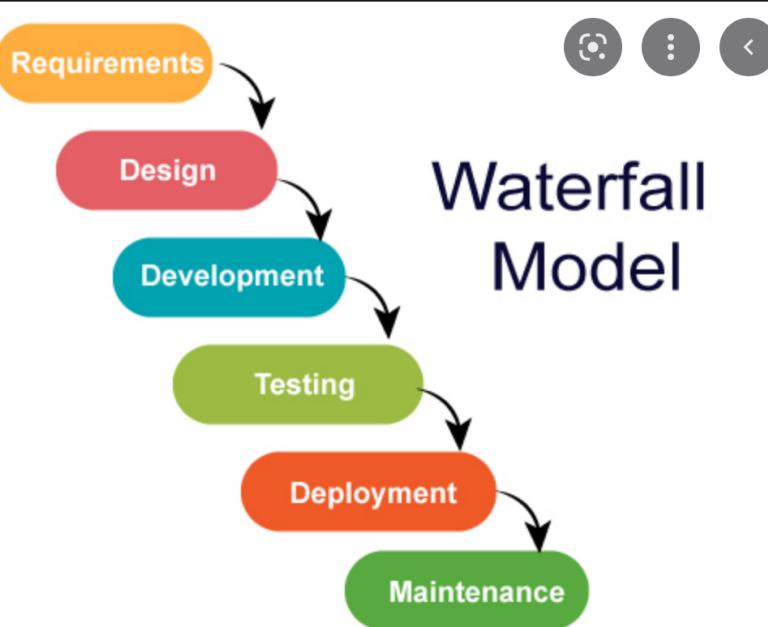


Figure 36. Waterfall method

In order to address this the V model was introduced which increases the amount of testing to reduce the possibility of having to rework everything

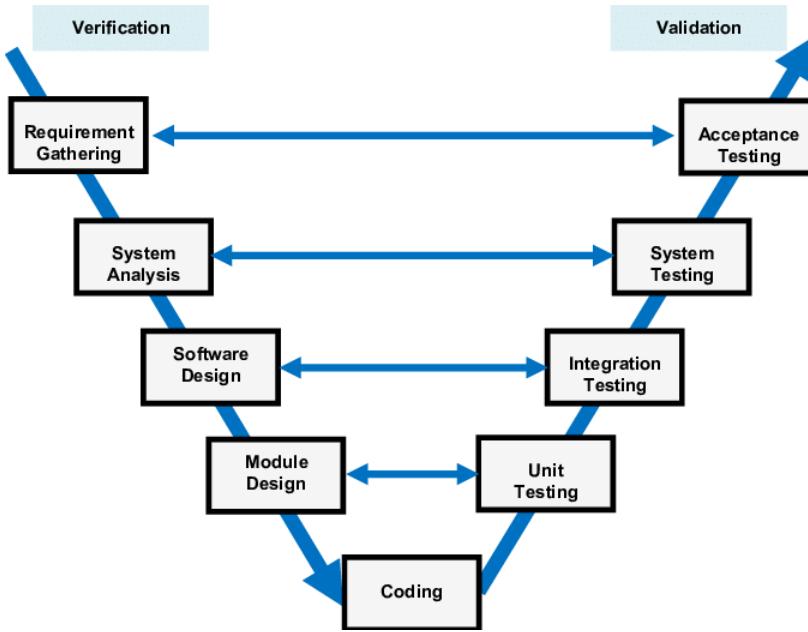


Figure 37. V model

Generally speaking the waterfall model isn't used much anymore due to the reality that software specifications change on a near daily basis.

Recall: aUToronto Spring 2022 integration hell

#### 24.1.1 Agile

Agile is a project management approach which, in most general terms, seeks to respond to change and unpredictability using incremental, iterative work (sprints). This allows for a balance between the need for predictability and the need for flexibility. Some agile methods include:

- Extreme programming: really really fast iteration (think days)
- Scrum: 2-4 week sprints with standups and backlogs; sticky notes for tasks, etc. Think kanban boards. Daily scrum meetings to unblock ASAP. Development lifecycle is therefore a series of sprints.
- On-site customer; frequent interaction with end users to figure out what exactly they need.

SUBSECTION 24.2

#### I dropped this course

I decided to drop this course because the courseload was a little too much to handle between EngSci ECE, clubs, design teams, work, and trying to have a life.

PART

VIII

What are notes?