EngSci Year 3 Fall 2022 Notes

Brian Chen

Division of Engineering Science
University of Toronto
https://chenbrian.ca

brianchen.chen@mail.utoronto.ca

Contents

PART

I

ECE349: Introduction to Energy Systems

Taught by Prof. P. Lehn

Section 1

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

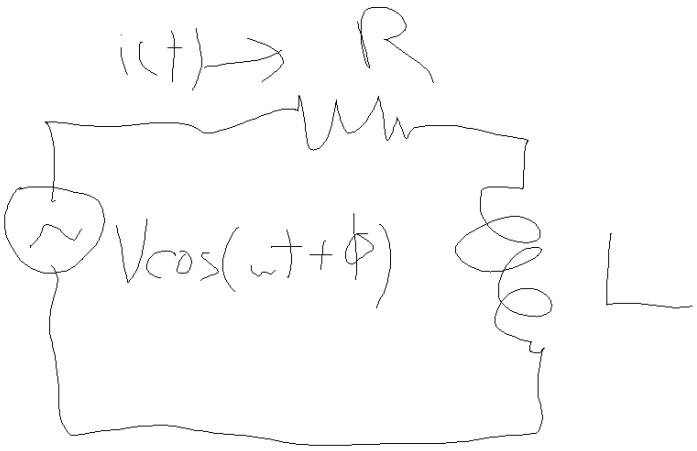
- Review Thomas 669-600

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, $\to 0$ as $t \to \infty$. 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 $\hat{P}$

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



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

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

$$V\cos(\omega t + \phi) = Re\{Ve^{j(\omega t+\phi)}\} \tag{2.2}$$

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

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

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,

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

And now reconstructing:

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

And therefore

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

The steps to solving a phasor problem are:

Notation: $Xe^{j\phi} \leftrightarrow X\lfloor\phi$

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

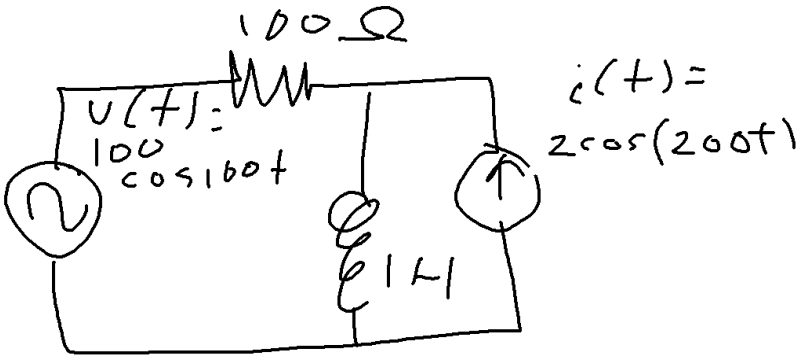
- $i = C\frac{dv}{dt} \leftrightarrow \hat{V} = \frac{1}{j\omega C}\hat{I}$

- 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.

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

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

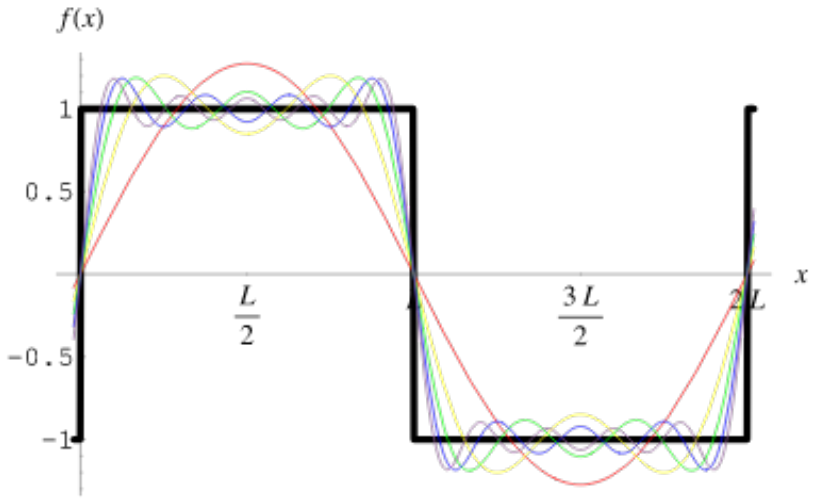


Figure 1. square waves with Fourier series superimposed

The general form of a Fourier transform is given as:

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

Where:

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

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^o, so

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

Section 3

AC Power

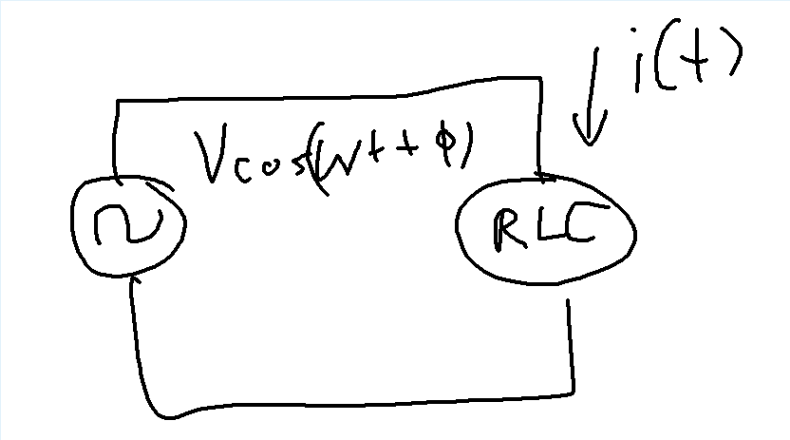
Definition 1 **Instantaneous Power**: $p(t) = v(t) \times i(t) [W, \frac{J}{s}]$

Example For a circuit with a voltage source, $v(t) = Vcos(\omega t)$ and a resistor Ω, $i(t) = Icos(\omega t)$, $p(t) = VIcos^2(\omega t) = \frac{VI}{2}(1 + cos(2\omega t))$

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

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 ϕ 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: ϕ is -'ve
- Leading DF: ϕ is +'ve'

We can also write P, Q from phasors.

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

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

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:

$$S = \underline{V} \cdot \underline{I^*} \tag{3.12}$$

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,

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

And if V_m is the rms value,

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

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

$$PF \equiv \frac{\text{average power}}{\text{rms voltage} \cdot \text{rms current}} \tag{3.15}$$

For a sinusoidal V, I:

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

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,

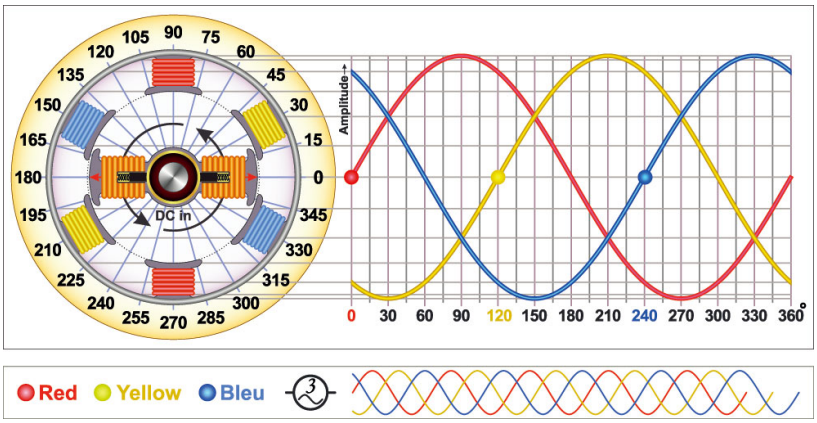
- In ideal systems, $PF = DF$ if all V and I are at one frequency.
- $PF = 1$ means no energy sloshing between load and source.

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 $0^o, 120^o, 240^o$, the voltages can be found with a little bit of trigonometry.

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

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 3ϕ power can carry 50% more power than 2ϕ power for less copper.

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

Four-wire three-phase power:

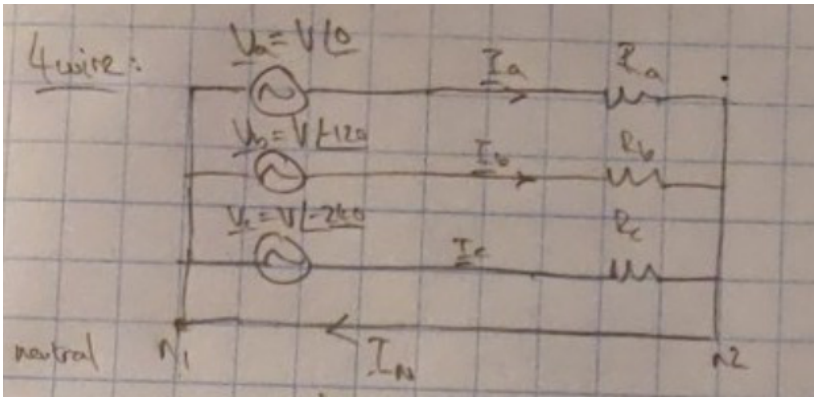


Figure 2. A four wire system for three phase power

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

$$Z_a = Z_b = Z_c = Z \tag{3.18}$$

$$\underline{V_b} V_a e^{\frac{-j_2 n}{3}} \tag{3.19}$$

$$\underline{V_c} V_a e^{\frac{-j_4 n}{3}} \tag{3.20}$$

Then,

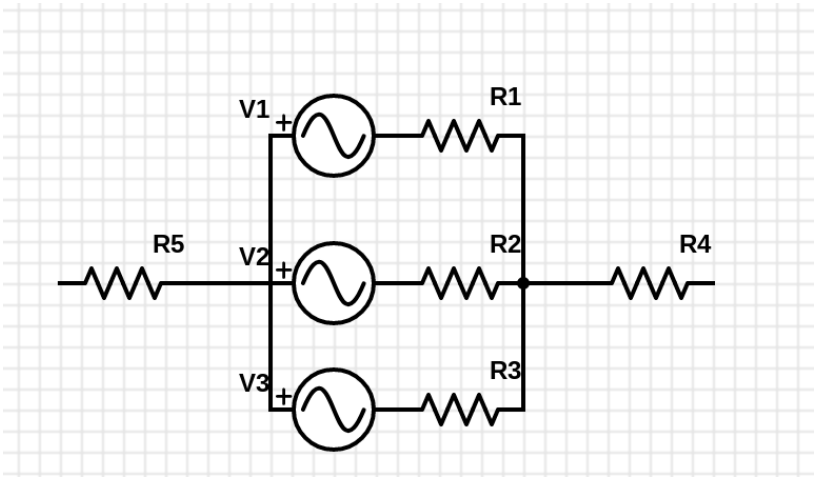
$$\underline{I_a} = \frac{\underline{V_a}}{Z} \tag{3.21}$$

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

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

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



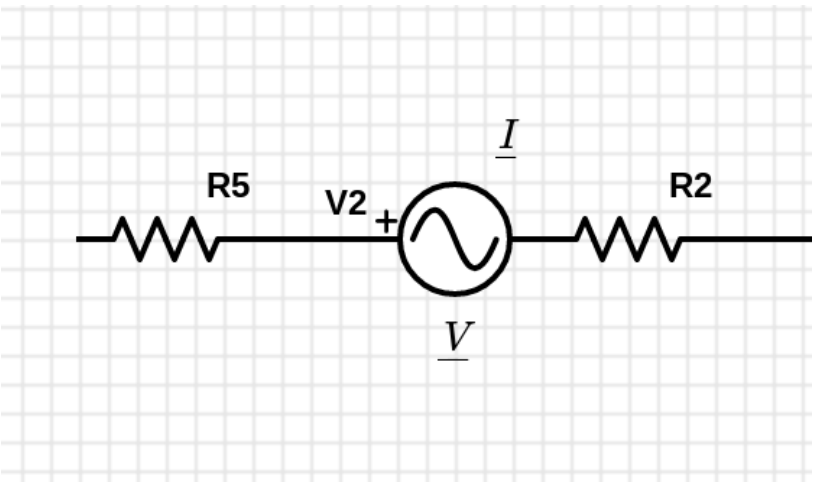


Figure 3. Condensed diagram for three phase power

Subsection 3.3

Lecture 6: Y and Delta connections

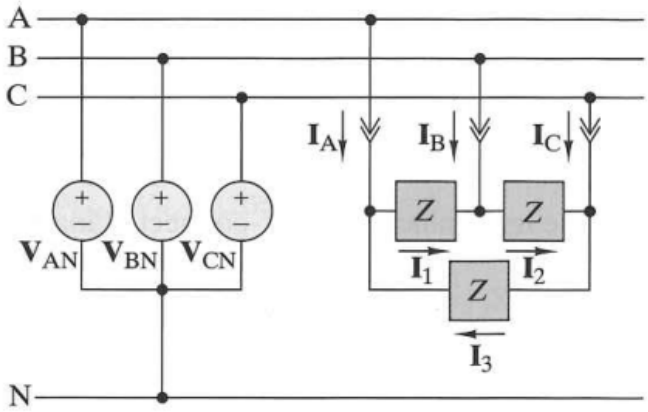
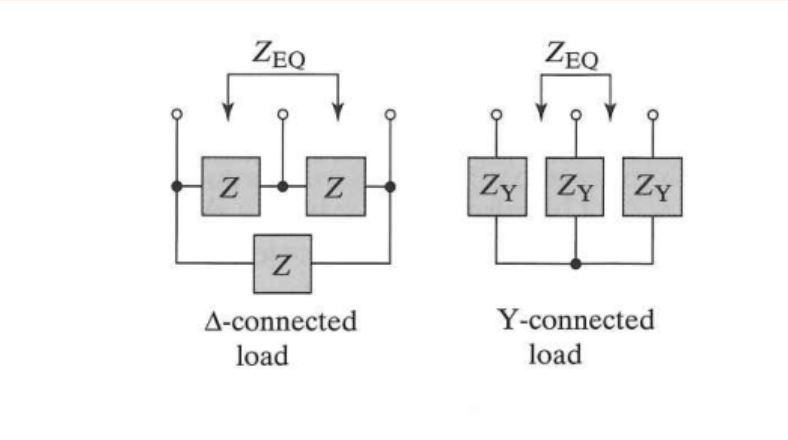


Figure 4. Three-phase system with a Y connected source and a Δ connected load

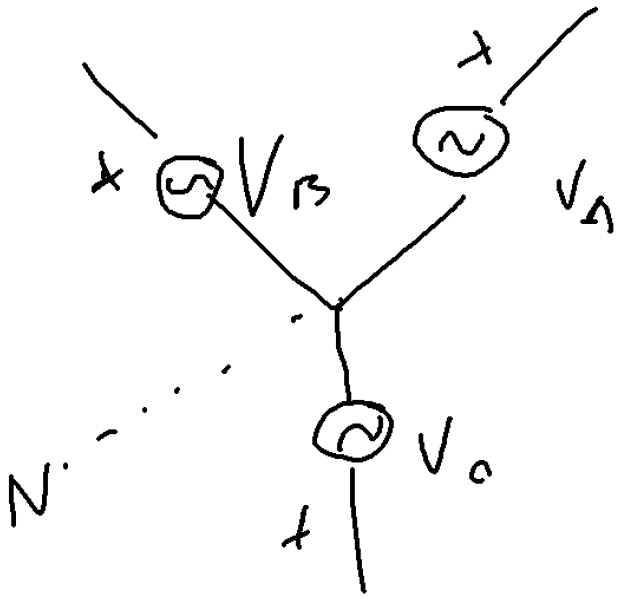
The current to ground I_N is always 0 for a Δ connected load. I_N is zero for a Y cnonnected load if Y has no neutral connection and the load/source are balanced.

Theorem 1 **$Y - \Delta$ conversion**
Any Δ load can be converted to a equivalent Y connected load



The derivation is in the textbook.
Sources can be connected in Y or Δ configurations.

Definition 9 Line currents are the currents on the lines a, b, c. Phase currents are the currents immediately beside the sources.

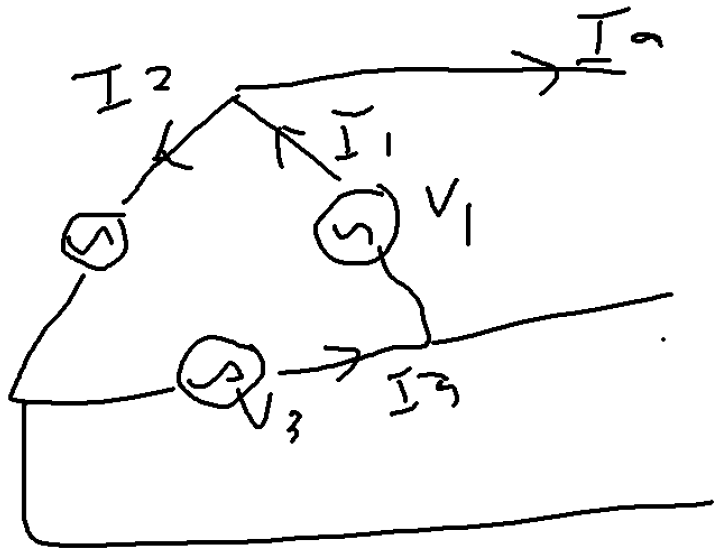


A Y connected source has the neutral line in the center.

$$\tilde{V}_{AB} = \tilde{V}_A - \tilde{V}_B = |\tilde{V}_A|(1 - 1\lfloor e^{-\frac{2\pi}{3}}) = \sqrt{3}|\tilde{V}_A|e^{\frac{j\pi}{6}} \tag{3.24}$$

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 Δ connected source has no neutral point.

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

The line and phase voltages are the exact same in the case of a Δ 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 3ϕ systems based on their line-to-line voltages. For exampl,e, a $208V$ system has $120V_{rms}$ on each phase.

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

$$I = \frac{10}{\sqrt{3} = 5.8A} \tag{3.26}$$

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 every 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 regular the power instead.

semiconductors to process power $\Leftrightarrow$ power electronics

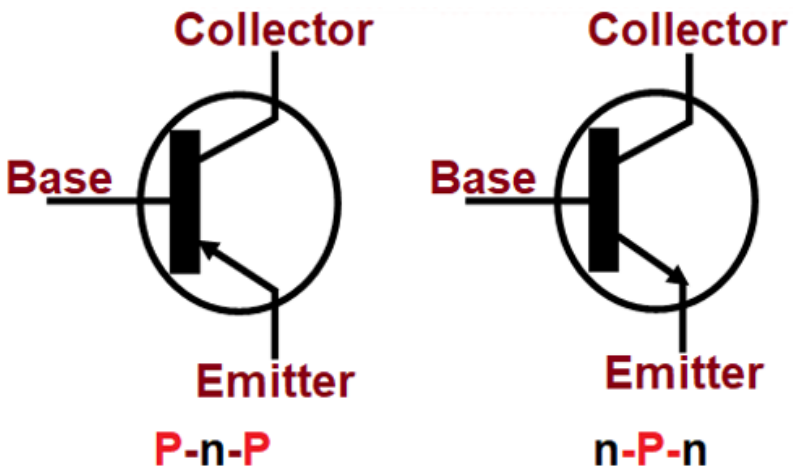


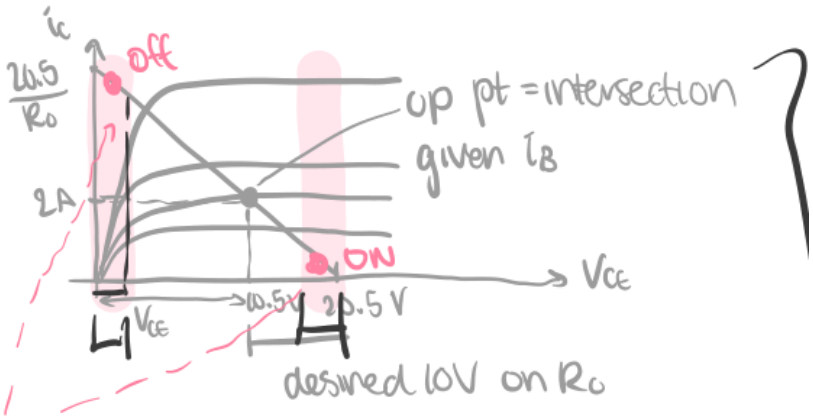
Figure 5. Transistors have a Base, Collector, and Emitter. At this point in the course, we just need to know that we can change the $V - I$ response by fiddling with the base current i_B

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 at 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 η 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 which 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.

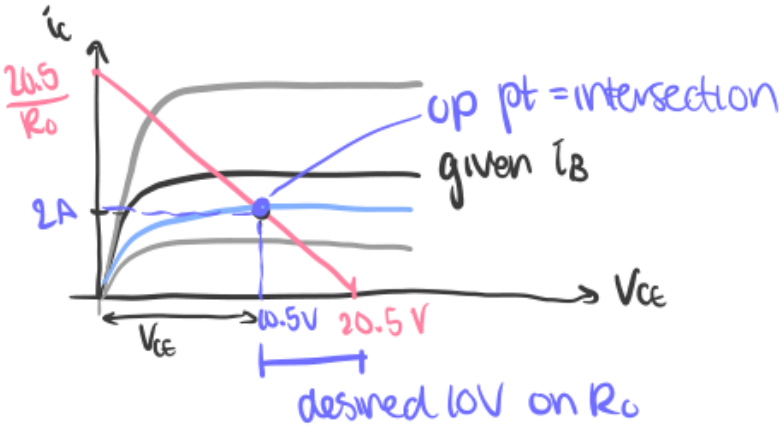
$$\begin{aligned} P &= \frac{1}{T}\int_0^T p(t)dt \\ &= \frac{1}{T}\int_0^{T_{on}} i_c^2 R_o dt \\ &= \frac{1}{T}\int_0^{T_{on}} 4^2 \cdot 5 dt \\ &= 80\frac{T_{on}}{T} \end{aligned} \tag{3.27}$$

So in order to transfer the same $20W$ to the 5Ω 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[1]

[1] *compared to the* 49% *before*

A few points of caution:

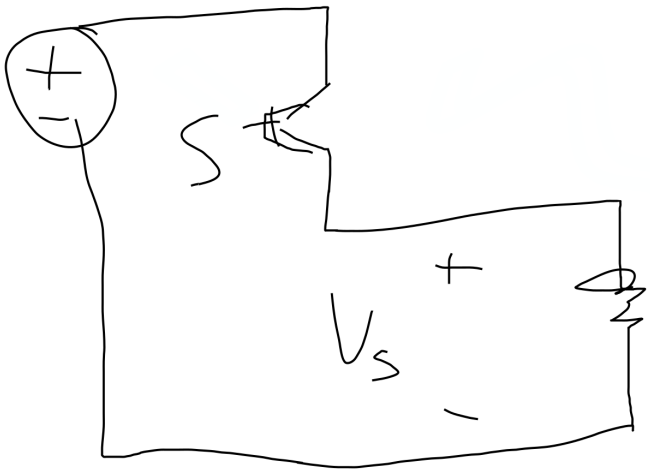
- A pulsing output voltage/power can be dangerous for some loads. For example motors aren't a fan of it. In these cases a filter can be applied to leave behind the average power.
- If we want to supply average voltage/power an on-time of 50% would be required.



Subsection 3.5

Lecture 8: Average output voltage

Consider a circuit with a transistor and a load.

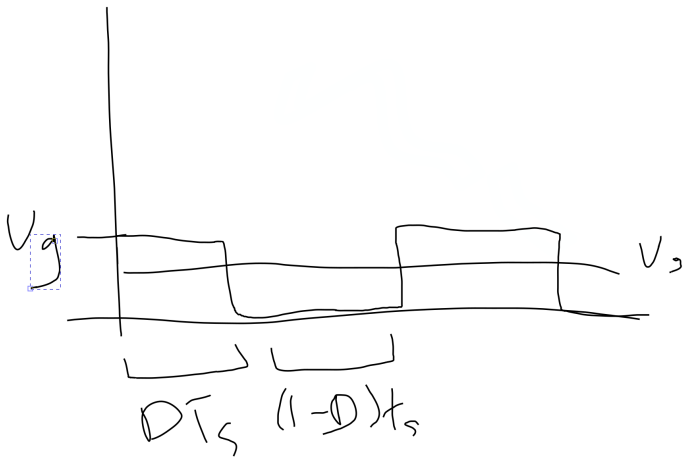


A switching signal S taking on a value from $0, 1$ periodically over time is applied to the transistor

Definition 10 **Duty cycle**: the fraction of time that S is 1

$$D = \frac{T_{on}}{T_s} \tag{3.28}$$

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 characteristic is the ripple voltage, i.e. unwanted harmonics

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

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

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

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:

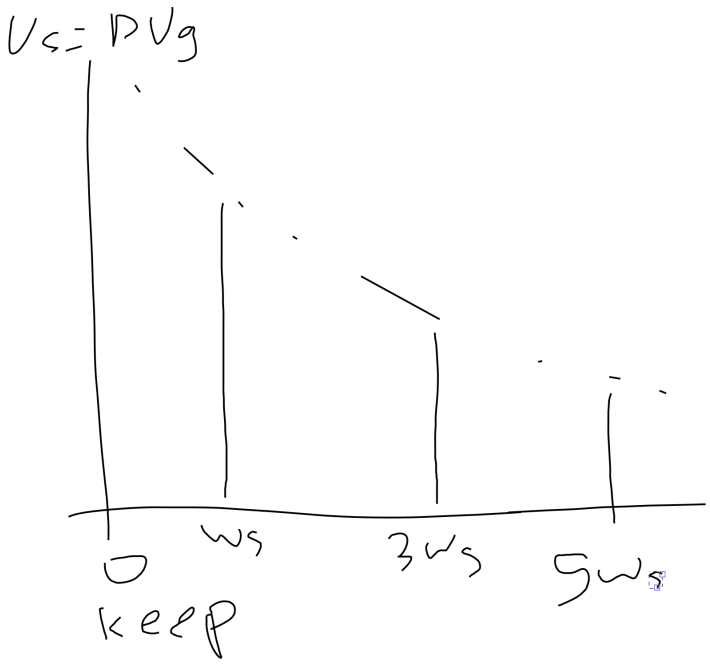
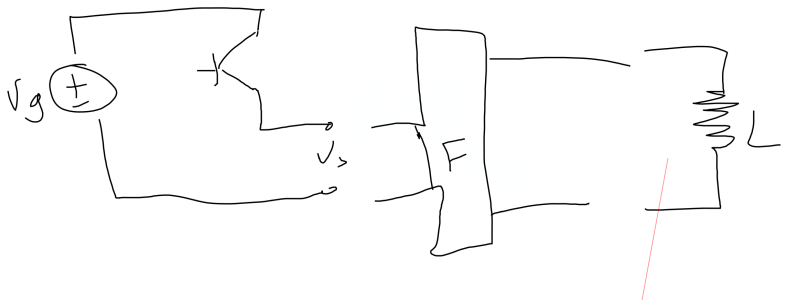


Figure 7. Voltage vs frequency. We want to keep the $\omega = 0$ term and then eliminate the rest with a filter. Note switching frequency $\omega_s = \frac{2\pi}{T_s}$

3.5.1 Filtering

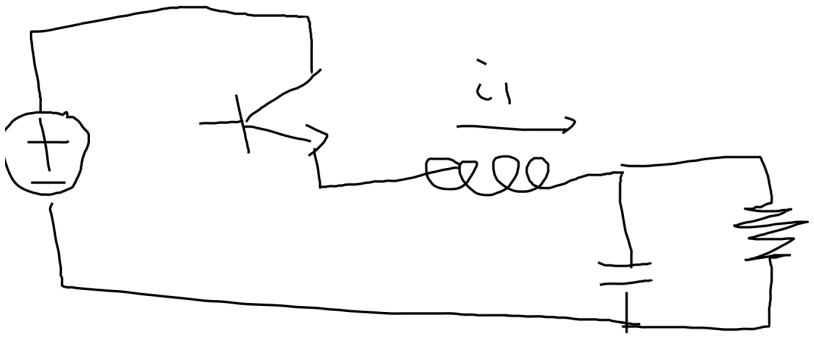
We'll need a "low pass" filter for dc/dc filtering.



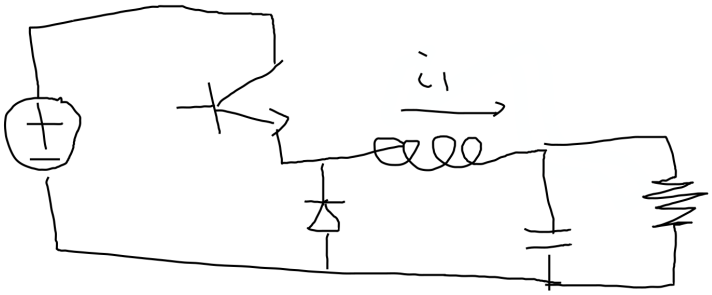
Choices for filters that may work for this system are the RC, LR , and LC filters. The LC is most promising because it is second order and it's theoretically lossless because there is no resistor.

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, $i \to 0$. This is a problem because this will cause us to lose all the energy stored in the inductor everything the switch is opened, which can be a lot given the frequency that these devices operate at.

$$E = \frac{1}{2}Li^2 \tag{3.31}$$



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[2] as it is an approximate solution to the differential equations based on periodic mapping

[2] *Well, we only use it for steady-state*

Subsection 3.6

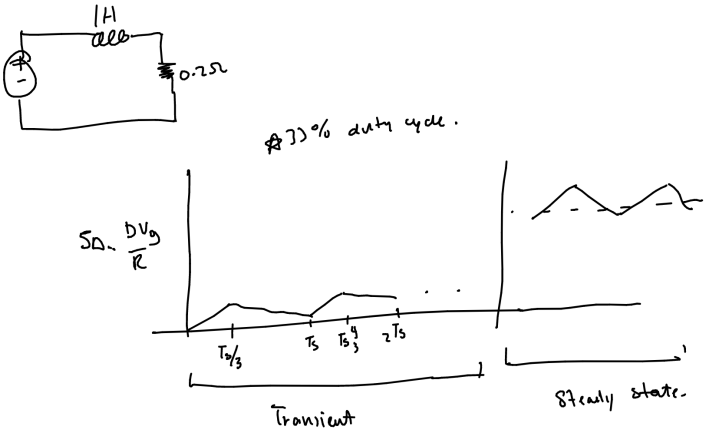
Lecture 9

A periodic system is in steady state if

1. all input sources are periodic
2. All states are periodic
3. switching events are periodic

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

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



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

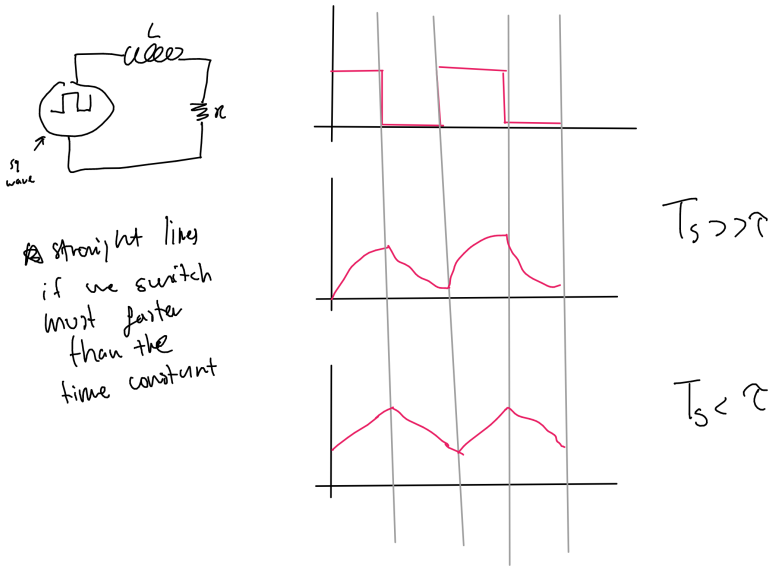


Figure 8. Note how the steady-state response becomes a bunch of straight lines if we switch faster than the time constant τ

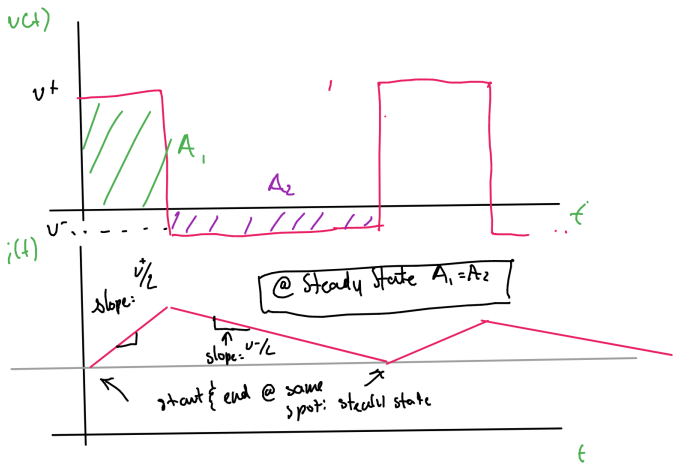
3.6.1 Inductor VoH-Seconds balance

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

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

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

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).

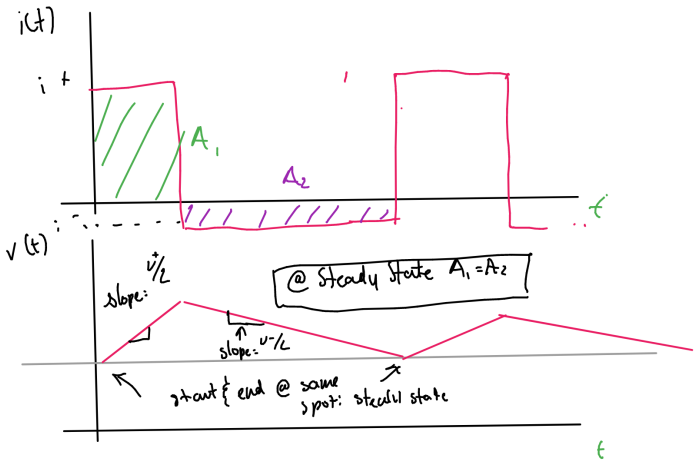


Figure 9. This $A_1 = A_2$ is the capacitor charge balance

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

$$\begin{aligned} i_l(t) &\approx I_l \\ v_l(t) &\approx V_c \end{aligned} \tag{3.36}$$

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

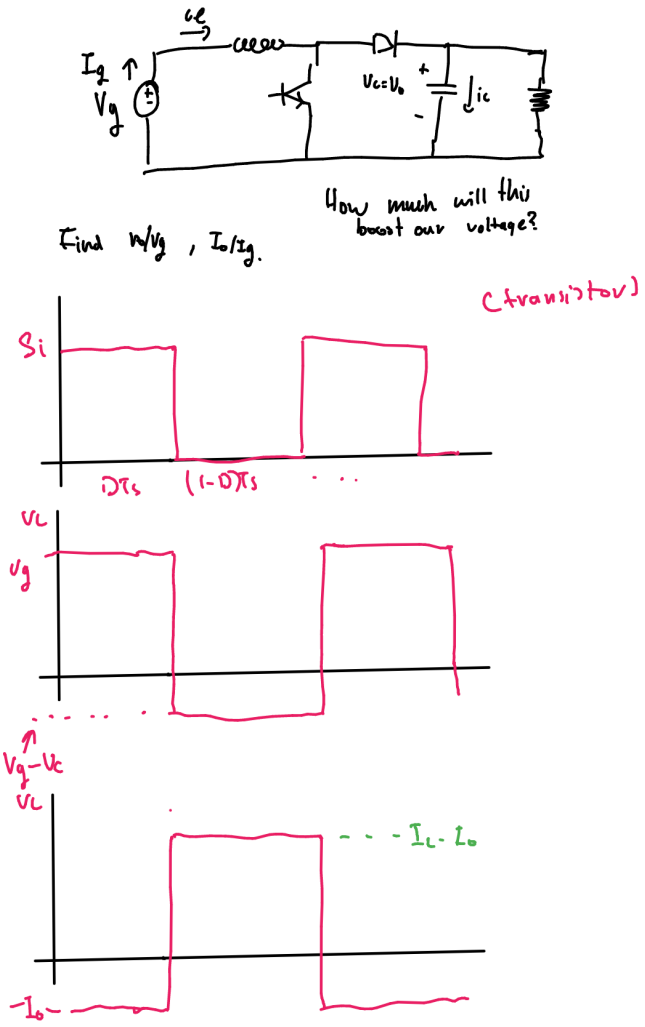


Figure 10. Example with a boost converter. Note: 3rd plot should be i_c vs t

Applying the $V - s$ balance:

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

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:

$$\begin{aligned} \int_0^{T_s} &= i_c(t)dt \\ -I_o(DT_s) + (I_L - I_o)(1-D)T_s &= 0 \\ \frac{I_o}{(I_l = I_g)} &= 1 - D \end{aligned} \tag{3.38}$$

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

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.

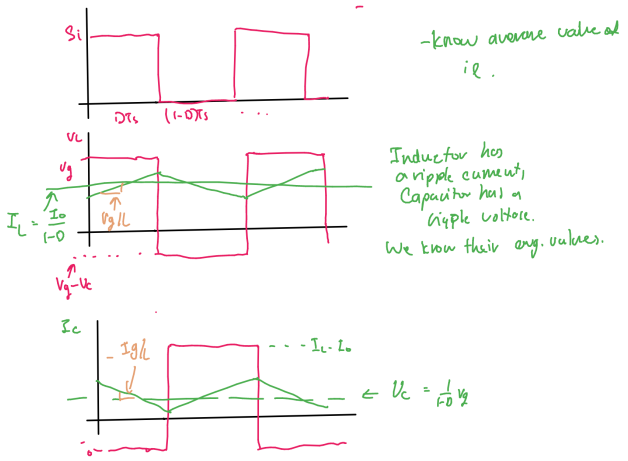


Figure 11. Inductor has a ripple current and the capacitor has a ripple voltage. The slope of the roughly saw-shaped waveform is inversely proportional to L, C. So we can pick L, C such that we get a small enough ripple.

So, what defines a 'small enough ripple?'
The amplitude of the ripple is defined from the peak to peak value of the waveform. [3]

[3] The textbook defines it as center to peak, but prof likes peak2peak

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

$$\Delta V_{op2p} = |\frac{1}{C}\int_0^{DT_s} -I_o dt| = \frac{I_o}{C} DT_s \tag{3.40}$$

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.

$V_o = V_c$. 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.

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[4]
4. Compute required duty cycle D for the desired output characteristics V_o/V_g

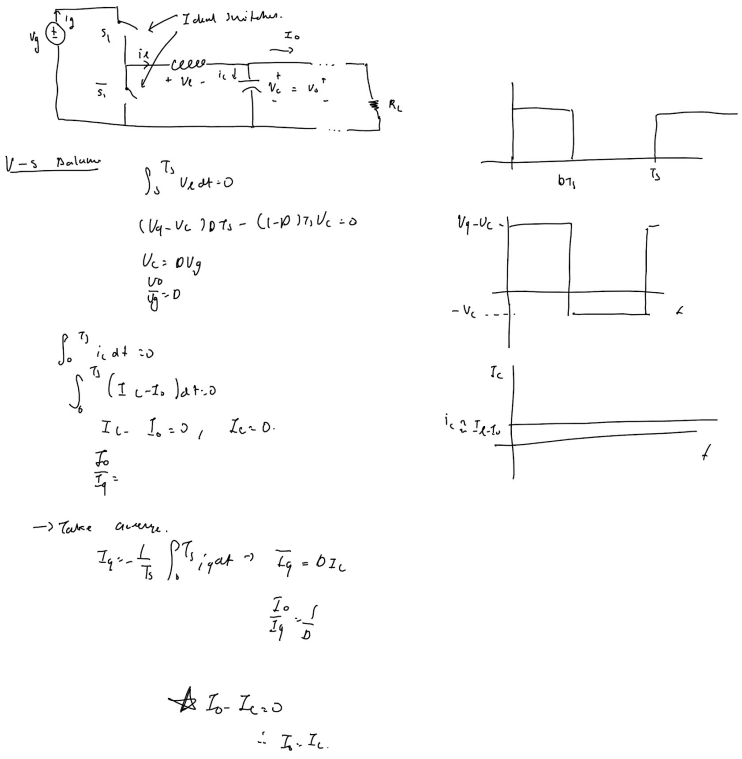
[4] *Usually smaller switches can run faster than larger ones*

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

Subsection 3.8

Lecture 11

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



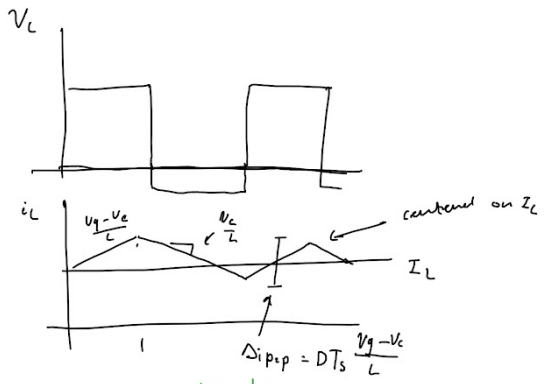
→ Take average.

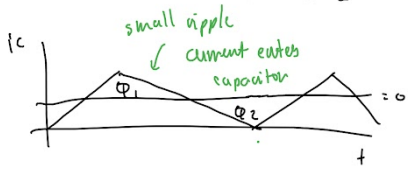
$I_g = \frac{1}{T_s} \int_0^{T_s} i_g dt \Rightarrow \overline{I_g} = D I_c$

$\frac{I_o}{I_g} = \frac{1}{D}$

$\therefore I_o = I_c$.

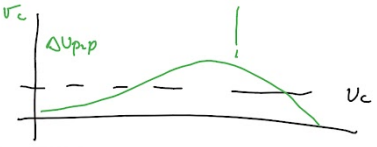
Steady ripple to size L, C





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

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



$$\Delta i_{L\,p\text{-}p} = D T_s \frac{v_g - v_o}{L}$$ Design eq #1

$$\Delta v_{p\text{-}p} = \frac{Q_1}{C} =$$

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

$$= \frac{T_s}{8} D T_s \frac{v_g - v_o}{L}$$

$$\Delta v_{p\text{-}p} = \frac{T_s^2}{8} D \frac{v_g - v_o}{LC}$$ Design eq #2

$v_o = D v_g \quad \therefore v_g = \frac{1}{D} v_o$

$(v_g - v_o) = \left(\frac{1}{D} - 1\right) v_o$

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

can rewrite in terms of others...

$$\frac{\Delta v_{p\text{-}p}}{v_o} = \frac{T_s^2}{8} \frac{(1-D)}{LC}$$ Alternative design eq

Subsection 3.9
Lecture 12

PART

II

ECE352: Computer Organization

Taught by Prof. Andreas Moshovos

Section 4
Admin stuff

Subsection 4.1
Lecture 1

- Lecture recordings on YouTube
- Online notes: https://www.eecg.utoronto.ca/ moshovos/ECE352-2022/. I "stole" a lot of figures from here but I think Prof. Moshovos is cool with it.

```
(*) Permission is given to reproduce these notes provided that a notice of their origin is clearly given. All
rights reserved just in case :)
```

Figure 12. Excerpt from online course notes

- 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.

Section 5
Preliminary

Subsection 5.1
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

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

- int: 32b (word)
- 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 [5]. 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 $-(2^{n-1}-1), +2^{n-1}-1$

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

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 [6] to the `char` so that math can be done on it.

[6] 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 MMM
```
(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:

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

Example

For example, given the following float:

$$1 \quad 10000001 \quad 10000000000000000000000$$

So S = 1, E = 10000001 = 129 and Mantissa = 10000000000000000000000. The number is therefore

$$(-1^1) \times 2^{(129-127)} \times 1.10000000000000000000000 = -6.0 \tag{5.3}$$

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

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

- 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

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

- Repeatedly divide the part of the number > 0 by 2 and get the remainders, i.e. 37/2 = 18, rem = 1 -> 18/2 = 0, rem = 0 -> 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 -> 1, 0.28 * 2 = 0.56 -> 0, 0.56 * 2 = 1.12 -> 1 ... and so forth. At some point we will hit a cycle but we'll just take the N_{mantissa} of digits.

So the full number is 01000010000101101010001111101011111

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.

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.

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 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.

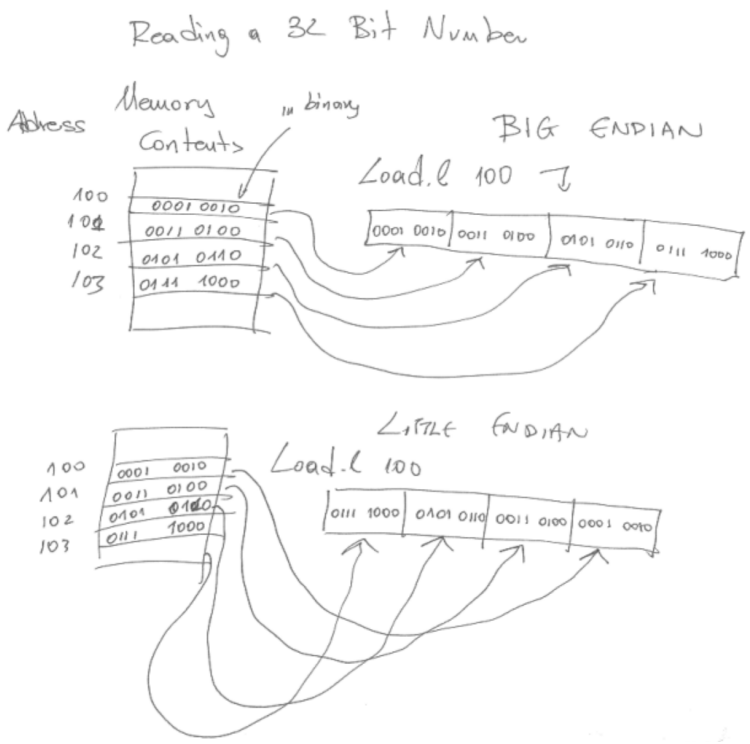
In this case it is convenient to draw memory as an array where each row comprises four consecutive bytes.

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[8]

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.

[8] 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?

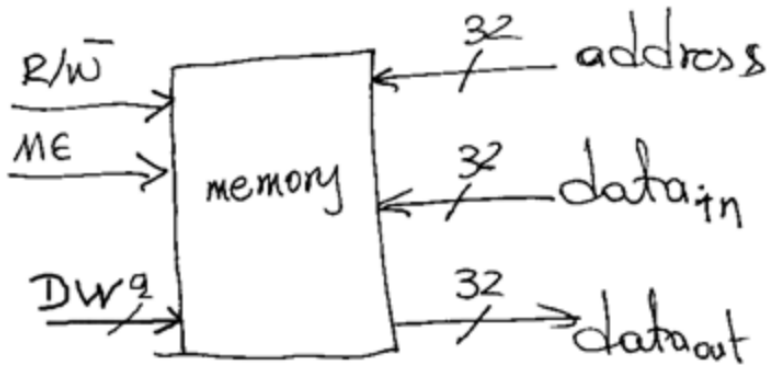
Given a summary of requirements as follows:

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

```
addi r9, r0, 0x1
        ; 1. read r0
        ; 2. Add value read in step 1 with 0x1
        ; 3. Write result of step 2 to r9
        ; 4. increment PC to next instruction
addi r10, r0, 0x2
        ; 1. read r0
        ; 2. Add value read in step 1 with 0x2
        ; 3. Write result of step 2 to r10
        ; 4. increment PC to next instruction
add r9, r10, r11
        ; 1. read r10
        ; 2. read r11
        ; 3. Add values read in steps 1 and 2
        ; 4. Write result of step 3 to r8
        ; 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.

```
movhi r9, 0x1122
        ; Sets the upper 16 bits of r9 to 0x1122
        ; and the lower 16 bits to zero
ori r9, r9, 0x3344
        ; bitwise OR the value in r9 with 0x3344
        ; 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

```
movi rX, Imm16
        ; sets rX to the sign-extended (signed) 16 bit immediate
movui rX, Imm16
        ; sets rX to a zero-extended unsigned 16 bit immediate
movia rX, Imm32
        ; 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 9, 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.

```
; read b from memory into r9
movhi r11, 0x0020
ori   r11, r11, 0x0004
ldw   r9, 0x0(r11)

; read c from memory into r10
movhi r11, 0x0020
ori   r11, 0x0008
ldw   r10, 0x0(r11)

; add, then store into r8
add   r8, r9, r10

; store r8 into memory
movhi r11, 0x0020
ori   r11, r11, 0x0000
stw   r8, 0x0(r11)
```

The new instructions introduced here are

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

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

This can be simplified using the `movia` macro

```
movia r11, 0x200004
ldw   r9, 0x0(r11)
movia r11, 0x200008
ldw   r10, 0x0(r11)
add   r8, r9, r10
movia r11, 0x200000
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.

```
movhi r11, 0x0020
ori   r11, r11, 0x0004
ldw   r9, 0x0(r11)
;; can be replaced with
movhi r11, 0x0020
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

```
movhi r11, (upper 16 bits of A)
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.

```
movhi r11, 0x0020
ldw r9, 0x8000(r11)
;; this is incorrect because
;; will extend to 0xFFFF8000, which would result
;; 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:

```
movhi r11, %hiadj(0x208000)
ldw r9, %lo(0x2080000)(r11)
;; will extend to 0xFFFF8000, which would result
;; 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.

```
unsigned int a = 0x00000000;
unsigned int b = 0x11223344;
unsigned int c = 0x22334455;

if (b == 0)
then a = b + c;
else a = b - c;

```

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

```
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:
```

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 $PC + 4 +$ displacement, meaning that we can jump at most $+32774, -32772$ bytes[9] from the current program counter. The compiler will complain if the branch cannot be implemented.
Encoding is as follows:
`AAAAA BBBBB IIIIIIIIIIIIIIII 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.

[9] *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

```
        .data
        .align 4 ;; Align to word size addresses which are faster to
↪  access
a: .word 0
b: .word 0x11223344
c: .word 0x55667788
        .text
        movia r11, a  ;; moves the address of a into r11
        ldw r9, 4(r11) ;; loads the value at address a+4 into r9
        ldw r10, 8(r11)
        add r8, r9, r10
        stw r8, 0(r11)
```

Subsection 7.2

Lecture 6, 7: For loops and arrays

How can we implement the following in assembly?

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

for (i = 0; i  < n; i++)
   sum = sum + arr[i];
```

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

Generally element `a[i]` is at address `&a[0] + sizeof(TYPE) * i`

```
                .data
                .align 1
arr .hword 1,2,3,4,5
n   .hword 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 $1x(m \cdot n)$ 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 at what `c` does first.

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

```
for (init; cond; post)
        body
```

Listing 2: General form of a C for loop

Breaking it down a little more into atomic steps

```
INIT
if (!COND), we are done
BODY
POST
GOTO line 2
```

Listing 3: General form of a C for loop

Let's now rewrite this in assembly

```
                .text
forloop:
        add r8, r0, r0 ;; INIT
        movia r9, n ;; set COND, i.e. the n that we compare i to. See:
↪  loop invarient :)
        ldh r9, 0(r9)
;; movhi r9, %hiadj(n)
;; ldh r9, %lo(n)(r9) ;; this code block is a shorter piece with the
↪  same effect
loop:
        bge r8, r9, endloop ;; test condition for when we are done
        ;; body goes here
        ;; let's use r10 to store the running sum at in the end we can
↪  load it to memory
        movia r11, arr ;; load the address of arr[0] into r11
        ;; add the index to the address of arr[0]; &arr + i
        add r11, r11, r8
        ;; add again; &arr + i + i = &arr + 2i
        add r11, r11, r8
        ldhio r12, 0(r11) ;; r12 = arr[i]
        add r10, r10, r12 ;; r[10] += arr[i]

        addi r8, r8, 1 ;; POST i.e. r8 += 1
        br loop
endloop:
        movia r11, sum
        sth r12, 0(r11) ;; write the sum to memory
```

A similar approach can be taken for while loops.

```
.text

add       r8, r0, r0 ;; zero out r8
movia     r9, n ;; assumes n >= 1
ldh       r9, 0(r9) ;; loads condition

doloop:
                                        movia r11, arr ;; grab address
    ↪ of arr[0]
                                        add  r11, r11, r8
                                        add  r11, r11, r8 ;; grab addr
    ↪ of arr[i]
                                        ldh  r12, 0(r11) ;; load
    ↪ arr[i] into r12
                                        add   r10, r10, r12 ;; add
    ↪ arr[i] to sum

          addi r8, r8, 1 ;; increment i
          blt  r8, r9, doloop ;; evaluate loop condition
endloop:
          movia r11, sum
          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, $f.c \to f.i$; dealing with `;; defines`, etc. Inspect via `gcc -E`
2. `gcc -s f.i`: the `cc`, the c compiler $f.i \to f.s$: parse pure c from preprocessor and spew out assembly code
3. `as`: the assembler, $f.s \to f.o$: to turn assembly to machine code, creating objects ready for linkage
4. `ld`: the linker, $f.o \to f$: link together all the objects into a single executable. Looks through `LDPATH` to look for symbols to link. [10]

[10] *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*

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

```
int add3(int a, int b, int c){
        return a+b+c;
}
```

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

1. Be callable from anywherei n the program

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 it's 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.[11] There is no way to represent an empty stack in NIOS II; `r27` will always contain a valid pointer address. [12]

[11] *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*

[12] An element is in the stack if it's address is greater than `sp`

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

```
        // access ith elem, assuming index in r9
        add r9, r9, r9 ;; assume r9 contains nedex
        add r9, r9, r9 // r9 = 4 * i
        add r9, sp, r9 ;; sp + 4*i
        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.

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

Consider the following pseudo `c` code

```
boo_calls(){
        coo();
        doo();
        return;
}

coo(){
        doo();
        return;
}
doo(){
        return;
}
```

The NIOS II code would be as follows

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

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

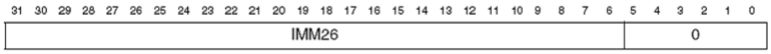


Figure 13. The call instruction accepts a label as the 2nd argument which is encoded as a 26 bit immediate. This limits the called function to be within 256 MB of the caller; the actual target is the immediate multiplied by 4 and then concatenating with the program counter, i.e. $PC_{31\ldots28} = IMM26 * 4$

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    0x2fc1(%rip),%rax
   ↪   # 3fd0 <__gmon_start__@Base>
11     100f:        48 85 c0                    test   %rax,%rax
12     1012:        74 02                       je     1016 <_init+0x16>
13     1014:        ff d0                       call   *%rax
14     1016:        48 83 c4 08                 add    $0x8,%rsp
15     101a:        c3                          ret
16
17 Disassembly of section .plt:
18
19 0000000000001020 <printf@plt-0x10>:
20     1020:        ff 35 ca 2f 00 00           push   0x2fca(%rip)
   ↪   # 3ff0 <_GLOBAL_OFFSET_TABLE_+0x8>
21     1026:        ff 25 cc 2f 00 00           jmp    *0x2fcc(%rip)
   ↪   # 3ff8 <_GLOBAL_OFFSET_TABLE_+0x10>
22     102c:        0f 1f 40 00                 nopl   0x0(%rax)
23
24 0000000000001030 <printf@plt>:
25     1030:        ff 25 ca 2f 00 00           jmp    *0x2fca(%rip)
   ↪   # 4000 <printf@GLIBC_2.2.5>
26     1036:        68 00 00 00 00              push   $0x0
27     103b:        e9 e0 ff ff ff              jmp    1020 <_init+0x20>
28
29 Disassembly of section .text:
30
31 0000000000001040 <_start>:
32     1040:        f3 0f 1e fa                 endbr64
33     1044:        31 ed                       xor    %ebp,%ebp
34     1046:        49 89 d1                    mov    %rdx,%r9
35     1049:        5e                          pop    %rsi
36     104a:        48 89 e2                    mov    %rsp,%rdx
37     104d:        48 83 e4 f0                 and
   ↪   $0xfffffffffffffff0,%rsp
38     1051:        50                          push   %rax
39     1052:        54                          push   %rsp
40     1053:        45 31 c0                    xor    %r8d,%r8d
41     1056:        31 c9                       xor    %ecx,%ecx
42     1058:        48 8d 3d da 00 00 00        lea    0xda(%rip),%rdi
   ↪   # 1139 <main>
43     105f:        ff 15 5b 2f 00 00           call   *0x2f5b(%rip)
   ↪   # 3fc0 <__libc_start_main@GLIBC_2.34>
44     1065:        f4                          hlt
45     1066:        66 2e 0f 1f 84 00 00        cs nopw 0x0(%rax,%rax,1)
46     106d:        00 00 00
47
48                ;; and this goes on for a while loading in the .so &
   ↪   printf
```

And more assembly later ...

```
0000000000001139 <main>:
    1139:       55                          push   %rbp
    113a:       48 89 e5                    mov    %rsp,%rbp
    113d:       48 8d 05 c0 0e 00 00        lea    0xec0(%rip),%rax
    ↪  # 2004 <_IO_stdin_used+0x4>
    1144:       48 89 c7                    mov    %rax,%rdi
    1147:       b8 00 00 00 00              mov    $0x0,%eax
    114c:       e8 df fe ff ff              call   1030 <printf@plt>
    1151:       b8 00 00 00 00              mov    $0x0,%eax
    1156:       5d                          pop    %rbp
    1157:       c3                          ret

Disassembly of section .fini:

0000000000001158 <_fini>:
    1158:       f3 0f 1e fa                 endbr64
    115c:       48 83 ec 08                 sub    $0x8,%rsp
    1160:       48 83 c4 08                 add    $0x8,%rsp
    1164:       c3                          ret
```

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

[13] *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 place the
   ↪  sum into r2
35         add r2, r2, r6    # add the third argument to r2
36         add r2, r2, r7    # add the fourth argument to r2
37         ldw r7, 0(sp)     # read the fifth argument from the stack
38         add r2, r2, r7    # add to r2
39         ldw r7, 4(sp)     # read the sixth argument from the stack
40         add r2, r2, r7    # add to r2
41         ldw r7, 8(sp)     # read the seventh argument from the stack
42         add r2, r2, r7    # add to r2 (return value in r2 by
   ↪  convention)
43          ret
44 ;; 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,

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

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

```
;; prologue
addi sp, sp, -20
stw, ra, 16(s0)  ;; note little-endian and remaining 16 bytes for 4
↪  argument words

;; epilogue
ldw ra, 16(sp) // pop return addr from stack
addi sp, sp, 20;
```

Subsection 7.5

Lecture 10: Recursive Subroutines

Consider this code block that computes the Ackerman function

```
int Ackerman(unsigned int x, unsigned int y)
{
        if (x==0) return y+1;
        if (y==0) return Ackerman(x-1, 1);
        return Ackerman(x-1, Ackerman(x, y-1));
}
```

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

```
int Ackerman(unsigned int x, unsigned int y)
{
        if (x==0) return y+1;
        if (y==0) return Ackerman(x-1, 1);
        int tmp = Ackerman(x, y-1)
        return Ackerman(x-1, tmp);
}
```

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

```
        .text
Ackerman:
        addi sp, sp, -8
        stw ra, 4(sp)

        bne r4, r0, Xnot0;
Xis0:
        addi r2, r5, 1
        br epilogue
Xnot0:
        bne r5, r0, Ynot0
Yis0:
        ;; pass arguments
        addi r4, r4, -1 ;;First one is x-1, i.e. r4-1
        addi r5, r0, 1 ;; second is 1
        call Ackerman
        br epilogue
Ynot0:
        stw r4, 0(sp) ;; perserve value of x on the stack
        ;; x is already at the right place for fn call
        addi r5, r5, -1 ;; decrement y
        add r5, r5, 0
        call Ackerman
epilogue:
        ldw ra, 4(s0)
        addi sp, sp, 8
        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.

```c
struct node{
        int value;
        struct node *left;
        struct node *right;
};
```

The memory layout of the struct is word-aligned and *exactly* like that of the struct definition. In this case value lives as 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

```c
int findv(int d, struct node) {
        struct node * m;
        m = root;
        if (root == NULL) {
                return 0;
        }
        if (root->val == d) {
                return 1;
        }
        if (d <= root->v) {
                return findv(d, root->left);
        }
        else {
                return findv(d, root->right);
        }
        // Note that this is a tail-recursive function, i.e.
        // nothing is done after the recursive call
        // some compilers may unfurl this into a loop
}
```

How can we represent this in assembly?

```
// int d is in r4, root is in r5 (by convention)
        .text

findv:
        addi sp, sp, -4
        stw R4, 0(sp);
isrootnull:
        bne r5, r0, isdv
rootisnull:
        movi r2, 0
        br epilogue
isdv:
        ldw r2, 0(r5) ;; r2 <- root->val
        bne r2, r4, tryagain
found:
        mov r2, 1
        br epilogue
tryagain:
        ble r4, r2, goleft
goleft:
        ldw r5, 4(r5) ;; offset struct addr pointer to right member
        call findv
        br end
goright:
        ldw r5, 8(r5) ;; offset struct addr pointer to right member
        call findv
        br end
notfound:
        add r2, r0, r0
epilogue:
        ldw ra, 0(sp)
        addi sp, sp, +4
        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 an null-terminated array of characters

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

Which in assembly looks like:

$$.datas : .byte'a','b','c','a',0;;Or, canwritewithdoublequotesi.e.stringliterals : .string"abba" \tag{7.1}$$

The C implementation of this function is as follows:

```c
int palindrome(char *a)
{
        char * e;
        if (a == NULL) return 1; // empty string is a palindrome
        e = a;
        while (*e != 0) e++; // find end of string
        if (e == a) return 1; // string of length 1 is a palindrome
        e--;
        while ((*a != 0) && (*a == *e)) {
                a++;
                e--;
        }
        if (*a == 0) return 1; // is a palindrome
        return 0;

}
```

In assembly this would be as follows:

```
        .text
palindrome:
         beq r4, r0, ret1  ;; null string is a palindrome
         add r8, r4, r0 ;; a in r4 (function call convention), e in r8

findzero:
        ldb r9, 0(r8) ;; r9 <- *e
        beq r9, r0, foundzero
        addi r8, r8, 1
        br findzero

foundzero:
        beq r8, r4, ret1 ;; string of length 1 is a palindrome
        subi r8, r8, 1 ;; move back to last nonzero char

cmploop:
        ldb r10, 0(r4) ;; r9 <- *a
        beq r10, r0, after ;; end loop when a gets to the end of the
↪  string
        ldb r9, 0(r8) ;; r9 <- *e
        bne r9, r10, after ;; if *a != *e, then not a palindrome
        addi r4, r4, 1 ;; increment a
        subi r8, r8, 1 ;; decrement e
        br cmploop

after:
        ldb r9, 0(r4) ;; r9 <- *a
        beq r9, r0, ret1 ;; if *a == 0, then is a palindrome

ret0: ;; not palindrome

ret1: ;; is palindrome
        addi r2, r0, 1

```

As an example this palindrome function can be called as follows

```
        .data
s: .string "abba"

        .text
        .global main

main:
        movia r4, s
        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[14]

[14] *Of which the NIOS II board has two*

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

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

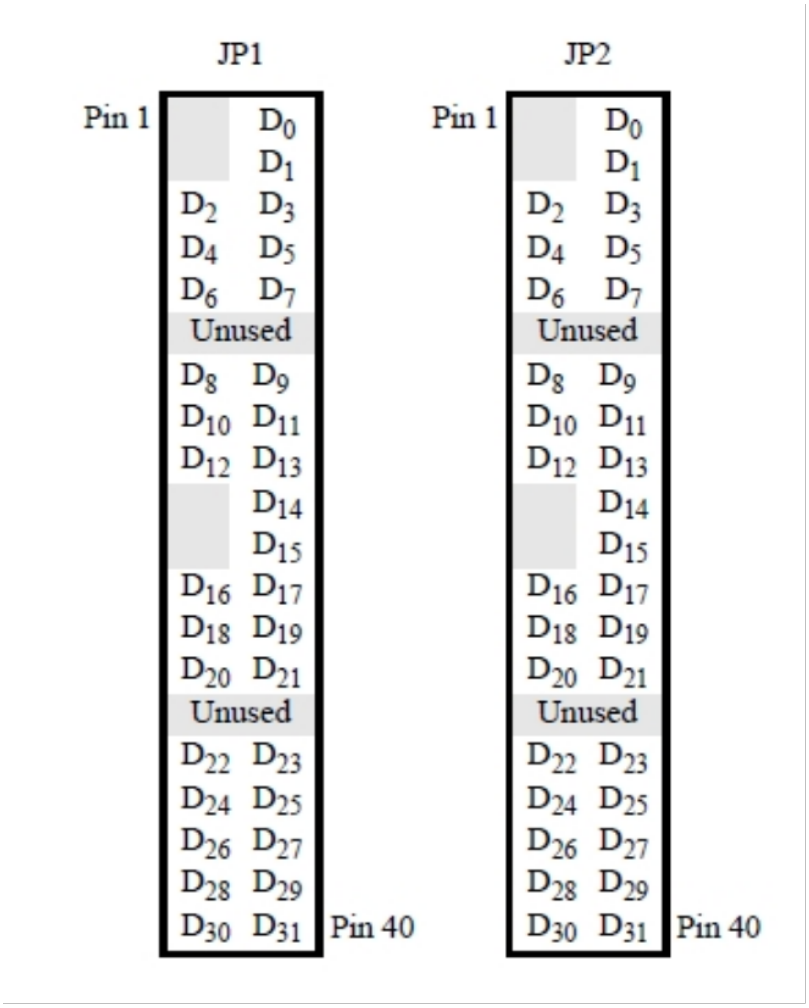


Figure 14. GPIO pinout

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

```
;; configure all pins as outputs
addi r8, r0, 0xFFFFFFFF
movia r9, 0xFF200064
stwio r8, 0(r9)

;; configure pins D0-D3 as outputs, D4-D31 as inputs
addi r8, r0, 0x0000000F
movia r9, 0xFF200064
stwio r8, 0(r9)
```

A write to an input pin will be silently ignored.

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

```
PIT1_BASE equ, 0xFF200060
PIT1_DR_OFFSET equ, 0x0
PIT1_DIR_OFFSET equ, 0x4

        .text
heat:
        movia r9, PIT1_BASE
        addi r8, r0, 8
        stwio PIT1_DIR_OFFSET(r9) ;; configure all pins but d3 as
↪  inputs (0x00000008)
        stwio r0, PIT1_DR_OFFSET(r9) ;; turn off the motor
fever:
        ldwio r8, PIT1_DR_OFFSET(r9) ;; read all pins
        andi r8, r8, 0x1 ;; mask off all but d0
        beq r8, r0, fanon ;; turn fan on because it's cold (d0 == 9)
fanoff:
        stwio r0, PIT1_DR_OFFSET(r9) ;; turn off the motor
        br fever
fanon:
        addi r8, 8
        stwio r8, PIT1_DR_OFFSET(r9) ;; turn on the motor
        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 $\frac{N}{2}$

```
PIT1_BASE equ, 0xFF200060
PIT1_DR_OFFSET equ, 0x0
PIT1_DIR_OFFSET equ, 0x4

        .text
heat:
        movia r9, PIT1_BASE
        addi r8, r0, 8
        stwio PIT1_DIR_OFFSET(r9) ;; configure all pins but d3 as inputs (0x00000008)
        stwio r0, PIT1_DR_OFFSET(r9) ;; turn off the motor

fever:
        addi r10, r0, 1000 ;; take 1000 samples
        add r11, r0, r0 ;; count of samples that are 1

sampling:
        ldwio r8, PIT1_DR_OFFSET(r9) ;; read all pins
        andi r8, 0x1 ;; mask out r0 (thermostat)
        add r11, r11, r8 ;; increment count if r8 is 1
        subi r10, r10, 1 ;; decrement sample count
        bne r10, r0, sampling ;; loop if we still have samples to take

        addi r10, r10, 500 ;; check if we have more than 500 samples that are 1
        bgt r11, r10, fanon

fanoff:
        stwio r0, PIT1_DR_OFFSET(r9) ;; turn off the motor
        br fever

fanon:
        addi r8, 8
        stwio r8, PIT1_DR_OFFSET(r9) ;; turn on the motor
        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

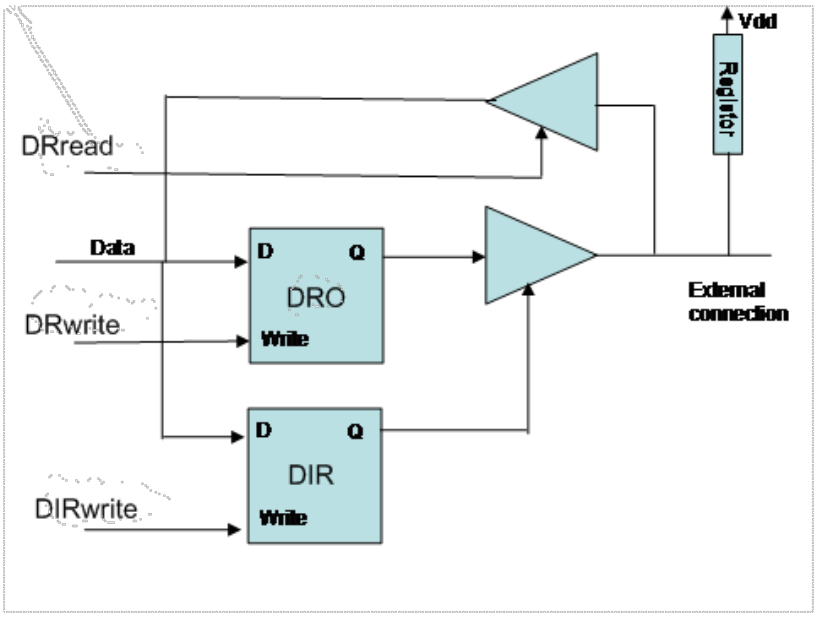


Figure 15. One PIT bit

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 signle 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

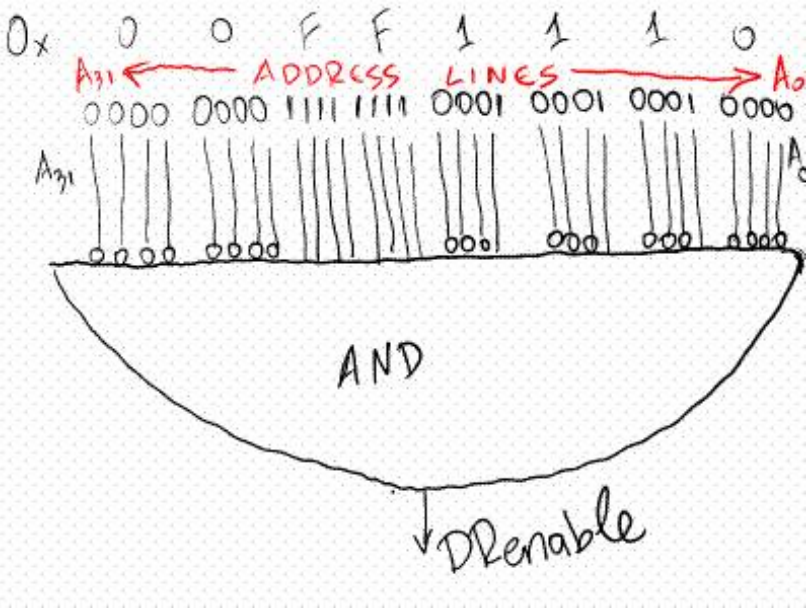
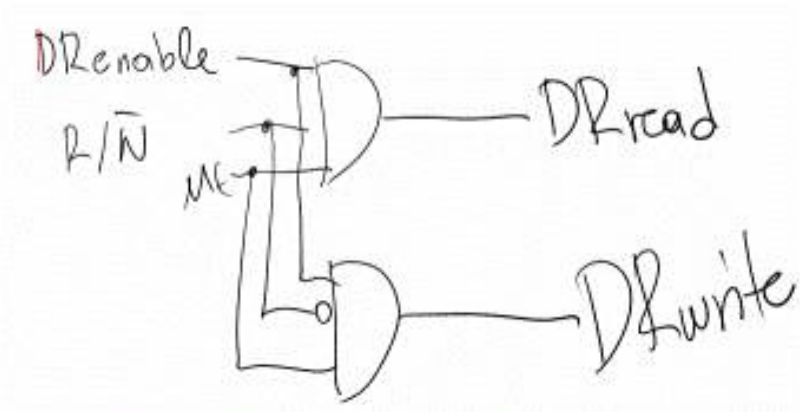


Figure 16. We'll also need to check if the address being accessed is the address of the PIT. The easiest way to do this is to just use an AND across the address lines and pass that signal to DREnable

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.

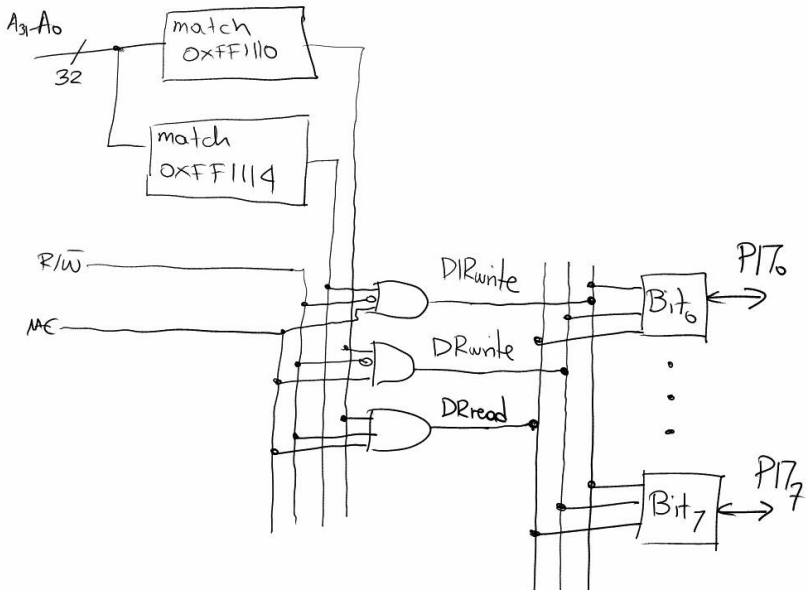


Figure 17. The complete design

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

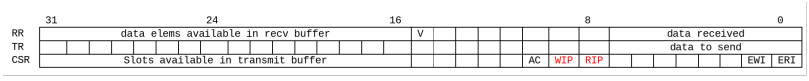


Figure 18. Register format

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`.

```
.equ JTAG_UART_BASE, 0xFF201000
.equ JTAG_UART_RR, 0
.equ JTAG_UART_TR, 0
.equ JTAG_UART_CSR, 4

        .text
putchar:
        movia r8, JTAG_UART_BASE
wait:
        ldwio r2, JTAG_UART_CSR(r8) # read csr
        srli r2, r2, 16 # keep only upper 16 (shift right logical
↪  immediate)
        ;; The csr's bits 16-31 contains a number which
        ;; is the number of slots in the queue avaliable
        ;; to accept requests. If the number is 0, then
        ;; the queue is full and we must wait.
        beq r2, r0, wait # wait until upper 16 bits are nonzero

        stwio r4, JTAG_UART_TR(r8) # place it into the FIFIO
        ret
wait:


```

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.

```
.equ JTAG_UART_BASE, 0xFF201000
.equ JTAG_UART_RR, 0
.equ JTAG_UART_TR, 0
.equ JTAG_UART_CSR, 4

        .text
getchar:
        movia r8, JTAG_UART_BASE

wait:
        ldwio r2, JTAG_UART_RR(r8) ;; read RR in r2
        andi r10, r2, 0x8000 ;; extract bit 15
        beq r10, r0, wait ;; wait until bit 15 is 1
        andi r2, r2, 0xff ;; keep only character (bit 0-7) in r2
        ret


```

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

```
;; An echo routine
.equ JTAG_UART_BASE, 0xFF201000
.equ JTAG_UART_RR, 0
.equ JTAG_UART_TR, 0
.equ JTAG_UART_CSR, 4

  .text
echo:
  movia r8, JTAG_UART_BASE
waitr:
  ldwio r2, JTAG_UART_RR(r8)    # read RR in r2
        # extract bit 15 in register r10 / keep a copy of r9 since it
↪  contains the character if any
  andi  r9, r2, 0x8000
        # if bit 15 was zero, there was no character, keep
↪  waiting/trying
  beq   r9, r0, waitr
        # a character was received, copy the lower 8 bits to r2 and
↪  return
  andi  r2, r2, 0xff

waitt:
        # read CSR in r9
  ldwio r9, JTAG_UART_CSR(r8)
        # keep only the upper 16 bits
  srli  r9, r9, 16
        # as long as the upper 16 bits were zero keep trying
  beq   r9, r0, waitt
        # place it in the FIFO
  stwio r2, JTAG_UART_TR(r8)
        # life is interesting, keep doing what you do
  br    waitr
        # never reaches here, this is for show
  ret


```

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 on 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.

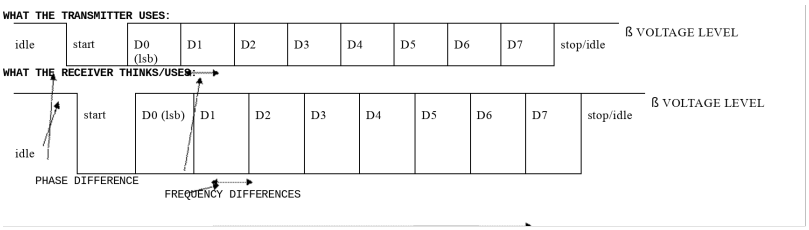
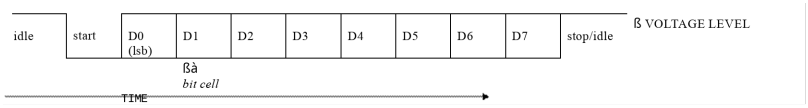


Figure 19. Phase differences

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 ith 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[16] can still occur. Further reduction of errors can be achieved by using using an additional parity bit which can be used to detect single errors.



[15] In practice there shoulnd't be a difference of more than 20% between the stop bit and where the receiver things 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.

[16] *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 it's 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 interrograte 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 IRQi 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

Subsection 8.4

Lecture 14: Timer

In lieu of lecture notes here is a big chunk of code which uses the NIOSII timer device to wait a second.

```
.equ  TIMER0_BASE,      0xFF202000
.equ  TIMER0_STATUS,    0
.equ  TIMER0_CONTROL,   4
.equ  TIMER0_PERIODL,   8
.equ  TIMER0_PERIODH,   12
.equ  TIMER0_SNAPL,     16
.equ  TIMER0_SNAPH,     20
.equ  TICKSPERSEC,      500000000
      .text
waitasec:
      movia r8, TIMER0_BASE
      addi  r9, r0, 0x8                   ;; stop the counter
      stwio r9, TIMER0_CONTROL(r8)

      ;; Set the period registers to 50M
      addi  r9, r0, %lo (TICKSPERSEC)
      stwio r9, TIMER0_PERIODL(r8)
      addi  r9, r0, %hi(TICKSPERSEC)
      stwio r9, TIMER0_PERIODH(r8)

;; tell the counter to start over automatically and start counting
      addi  r9, r0, 0x6                   ;; 0x6 = 0110 so we write 1
↪  to START and to CONT
      stwio r9, TIMER0_CONTROL(r8)

onesec:
      ldwio       r9, TIMER0_STATUS(r8)   ;; check if the TO bit of
↪  the status register is 1
      andi        r9, r9, 0x1
      beq         r9, r0, onesec

      addi        r9, r9, 0x0             ;; clear the TO bit

      stwio       r9, TIMER0_STATUS(r8)

      ;; decrement the number of remaining seconds and repeat until
↪  this becomes zero
      subi        r4, r4, 1

      bne         r4, r0, onesec

      ;; stop the counter before exiting

      addi        r9, r9, 8
      stwio       r9, TIMER0_CONTROL(r8)
      ret

;; And here's a function that calls waitasec asking it to wait for 20
↪  seconds:
      .text
      .global main

main:
      addi  r4, r0, 20
      call  waitasec
      ret

```

Checking the current value of the period registers cannot happen directly and instead must be done through snap registers.

```
   movia r8, TIMER0_BASE
   stwio r0, TIMER0_SNAPL(r7)  # Take a snapshot of the period registers
   ldwio r9, TIMER0_SNAPL(r7)  # Read snapshot bits 0..15
   ldwio r10, TIMER0_SNAPH(r7) # Read snapshot bits 16...31
   slli  r10, r10, 16          # Shift the upper bits to positions 16 through 31 in register r10
   or    r9, r9, r10           # Combine bits 0..15 and 16...31 into one register
```

Subsection 8.5

Lecture 15: Code Races

Consider the following c code:

```
      int   cnt = 9;
interrupt:
            cnt++;
program:
            while (1) cnt--;
```

At the machine code level this can look like

```
      .data
cnt:  .word 9
      .section .exceptions, "ax"
iHdlr:
H1:         ldw         r9, 0(r8)
H2:         addi        r9, r9, 1
H3:         stw         r9, 0(r8)
            eret
      .text
main:
            movia       r8, cnt
            movi        r9, 1
            wrctl       ctl0, r1
wait:
I1:         ldw         r11, 0(r8)
I2:         addi        r11, r11, -1
I3:         stw         r11, 0(r8)
            br          wait
```

We can possibly observe the sequence $10, 11, 10, 9$, and so forth because of a *race* between the interrupt and the while loop. This is because NIOS II does not implement atomic operations, therefore the same value can be loaded a register for two operations that change it

without coordinating between the two programs, and therefore cause undesirable behaviour. Other architectures may implement atomic operations, which are operations that cannot be by another instruction (usually at the expense of performance).

8.5.1 Buffer Overflow Stack Attacks

Consider the following `c` method which takes a `char*` input over the network

```c
void packet_parse_string(char* packet){
        char buffer[100];
        // some code
        // copy packet (null-terminated string) into buffer
        while (packet[i]){
                buff[j++] = packet[i++];
        }
        // some other code
}
```

One may immediately notice that there is no check to see if the packet is larger than 100 bytes, and therefore the buffer may overflow. Though this may seem like an innocuous invalid write, this function can be exploited to run arbitrary code. From this course we know that when returning it will try to restore `RA` from the stack, perform `RET`, and start executing code from `PC=RA`. This means that if the malicious caller can identify where the top of the stack was when calling, they may change the value of `RA` via the buffer overflow to the address of `buffer` and therefore execute arbitrary code that they sent inside of `packet`.

Ways to defend against such attacks include writing non-sloppy code or forbidding execution of code on the stack[17]

[17] *though there are valid reasons for doing this at times, i.e. JIT compilation.*

Subsection 8.6

Lecture 16: Emulating instructions in software

Interrupts can be used for more than just communicating with I/O devices, i.e. for detecting erroneous conditions during execution, misaligned memory access, or OS calls. They can also be used to emulate an instruction in software.

For example, on the NIOSII there is a `mulxuu rC, rA, rB` instruction to multiply two 32 bit registers and then store the upper 32 bits of the result into a rC. Some NIOSII implementations may include a hardware unit which implements this instruction, but others may rely on emulation to execute this instruction instead. Likewise, the 80386 processor does not have a hardware unit to implement floating point instructions (but one could do them in hardware with the 80387 co-processor). If one did not have the 80387 they can use interrupts to emulate the instruction in software instead (though slower)

Instruction Type: R

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

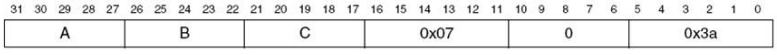


Figure 20. Encoding of the `mulxuu` instruction

Bits 31-27, 26-22, 21-17 encodes the source and destination registers. 16-11 contains 0x7, 10-6 the value 0, and 5-0 the value 0x31. So TLDR bit 16 is 0 and the lower 16 bits 0x383a.

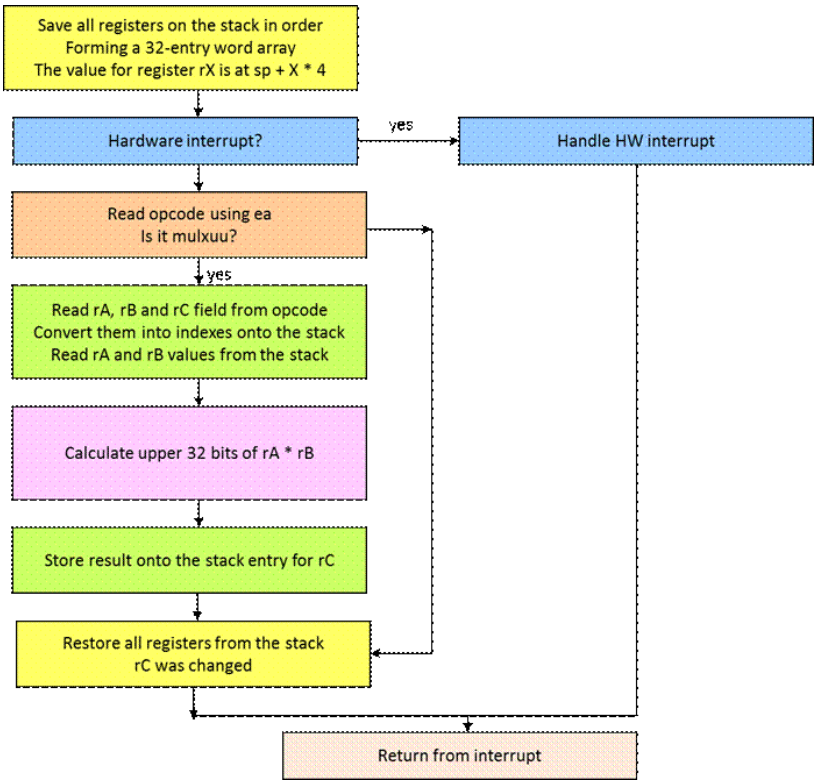


Figure 21. Interrupt handler flowchart

This chunk of code implements the first item in the flow chart, saving all the registers onto the stack. Note the commented `rdctl` checking for software interrupts

```
1       .section exceptions
2
3       ; tell the assembler to not introduce any additional
    ↪  instructions overwriting registers
4       .set nobreak
5       .set noat
6
7   handler:
8         ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
9         ; store all registers on the stack
10        ; forming an array of words
11        ; the value for register X is at sp+X*4 where X a number
    ↪  0...32
12        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
13        ; save all registers on the stack
14        subi  sp, sp, 32 * 4
15        stw r0,0(sp)
16        stw r1,4(sp)
17        stw r2,8(sp)
18        stw r3,12(sp)
19        stw r4,16(sp)
20        stw r5,20(sp)
21        stw r6,24(sp)
22        stw r7,28(sp)
23        stw r8,32(sp)
24        stw r9,36(sp)
25        stw r10,40(sp)
26        stw r11,44(sp)
27        stw r12,48(sp)
28        stw r13,52(sp)
29        stw r14,56(sp)
30        stw r15,60(sp)
31        stw r16,64(sp)
32        stw r17,68(sp)
33        stw r18,72(sp)
34        stw r19,76(sp)
35        stw r20,80(sp)
36        stw r21,84(sp)
37        stw r22,88(sp)
38        stw r23,92(sp)
39        stw r24,96(sp)
40        stw r25,100(sp)
41        stw r26,104(sp)
42        stw r27,108(sp)
43        stw r28,112(sp)
44        stw r29,116(sp)
45        stw r30,120(sp)
46        stw r31,124(sp)
47      rdctl et, ctl4               ; Check that interrupt was caused
    ↪  by      software
48      beq et, r0, software         ; if not, it's a hardware
    ↪  interrupt ignore
49
50      HANDLE HARDWARE INTERRUPTS HERE
51
52      br  iEpilogue
53
```

The software interrupt handler checks the opcode in order pass execution to the appropriate subroutine. For `mulxuu` this is just comparing the lower 16 bits of `ea` with `0x383a`

```
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ; read the instruction opcode to test whether it is a mulxuu
      ; ea points to the instruction
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
software:
      ldw   r9, -4(ea) ;; lecture notes uses stw but this should be ldw
      add   r10, r9, r0 ; keep a copy of the opcode
      andi  r9, r9, 0xffff    ; keep just the lower 16 bits
      cmpeqi      r11, r9, 0x383a
      beq   r11, r0, notmulxuu
      srli  r10, r10, 16      ; shift the upper 16 bits into the lower 16
      andi  r11, r10, 0x1     ; test bit 0 which used to be bit 17
      bne   r11, r0, notmulxuu ; if not zero this is not mulxuu

```

If we get to the `mulxuu` handler we need to load the registers from the stack

```
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ; Operand index calculations
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
ismulxuu:
      ; now calculate indexes into the stack for accessing
      ; the input and output operands
      ; treat the stack as a 32-entry array of words
      ; we extract the 5 bit field for each operand
      ; multiply by four because each entry is four bytes
      ; and add the stack point which is the base of the array
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      srli  r10,r10,1   ; keep just the upper 15 bits of the opcode
      ; rC
      andi  r11, r10, 0x1f    ; these are the 5 bits indicating rC the destination register
      slli  r11, r11, 2 ; multiply by 4
      add   r11, r11, sp      ; add the base of the array
      ; rB
      srli  r10, r10, 5
      andi  r12, r10, 0x1f ; keep the bits for rB
      slli  r12, r12, 2 ; multiply by 4
      add   r12, r12, sp      ; add the base of the array
      ; rA
      srli  r10, r10, 5
      andi  r13, r10, 0x1f ; keep the bits for rA
      slli  r13, r13, 2 ; multiply by 4
      add   r13, r13, sp      ; add the base of the array

```

...and then multiply the two registers and store the upper 32 bits into the destination reg-

ister.

```
1
2       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
3       ; Access input registers
4       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
5       ; at this point:
6       ; r11 points to the entry for rC
7       ; r12 points to the entry for rB
8       ; r13 points to the entry for rA
9       ; read rA and rB into r9 and r10 respectively
10      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
11      stw r9, 0(r13)
12      stw r10, 0(r12)
13
14      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
15      ; Multiplication : No need to understand how this works
16      ; end result is in r10
17      ; I haven't tested it much :(
18      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
19      srli  r4, r9, 16  ; a = (v1 >> 16) & 0xffff;
20      andi  r5, r9, 0xffff    ; b = v1 & 0xffff;
21      srli  r6, r10, 16 ; c = (v2 >> 16) & 0xffff
22      andi  r7, r10, 0xffff   ; d = v2 & 0xffff;
23
24      mul   r9, r5, r7  ; LO = b * d;
25      srli  r9, r9, 16  ; y = ((LO >> 16) & 0xffff)
26      mul   r10, r4, r7 ; x= a * d
27      mul   r12, r5, r6 ; x1 = c * b
28      add   r10, r10, r12     ; x = x + x1
29      add   r9, r9, r10 ; y = y + x
30      srli  r9, r9, 16  ; y = (y >> 16) & 0xffff
31      mul   r10, r4, r6 ; HI = a * c
32      add   r10, r10, r9      ; HI = HI + y
33
34      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
35      ; write result onto the corresponding stack entry
36      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
37      ; store the result to the stack
38      stw r10, 0(r11)
39
40      ; declare this instruction as executed
41      addi  ea, ea, 4
42
43
```

And clean up!

```
1
2
3
4
5  iEpilogue:
6  notmulxuu:
7          ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
8          ; restore all registers from the stack
9          ; one value has been changed
10         ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
11         stw r0,0(sp)
12         stw r1,4(sp)
13         stw r2,8(sp)
14         stw r3,12(sp)
15         stw r4,16(sp)
16         stw r5,20(sp)
17         stw r6,24(sp)
18         stw r7,28(sp)
19         stw r8,32(sp)
20         stw r9,36(sp)
21         stw r10,40(sp)
22         stw r11,44(sp)
23         stw r12,48(sp)
24         stw r13,52(sp)
25         stw r14,56(sp)
26         stw r15,60(sp)
27         stw r16,64(sp)
28         stw r17,68(sp)
29         stw r18,72(sp)
30         stw r19,76(sp)
31         stw r20,80(sp)
32         stw r21,84(sp)
33         stw r22,88(sp)
34         stw r23,92(sp)
35         stw r24,96(sp)
36         stw r25,100(sp)
37         stw r26,104(sp)
38         stw r27,108(sp)
39         stw r28,112(sp)
40         stw r29,116(sp)
41         stw r30,120(sp)
42         stw r31,124(sp)
43
44       ; restore the stack
45       addi  sp, sp, 32 * 4
46       br    idone
47
48       ; for hardware interrupts re-execute instruction that was
   ↪  interrupted
49 eadec:
50       subi  ea, ea, 4
51 idone:
52       eret
```

A piece of code that uses the new instruction is therefore

```
main:
            movhi r9, 0xffff
      ori   r9, r9, 0xffff
      add   r10, r9, r0
      mulxuu r11, r9, r10
```

Subsection 8.7

Lecture 17: A single cycle processor

For the purposes of this class we will consider a simple CPU capable of executing these 10 instructions (which are encoded as follows)

1. **load r1, r2**

```
                TMP = MEM [R2]
                R1 = TMP
                PC = PC + 1
```

7	6	5	4	3	2	1	0
R1	R1	R2	R2	0	0	0	0

2. **store r1, r2**

```
    MEM [R2] = R1
                PC = PC + 1
```

7	6	5	4	3	2	1	0
R1	R1	R2	R2	0	0	1	0

3. **add r1, r2**

```
    TMP = R1 + R2
    R1 = TMP
                        IF (TMP == 0) ZERO = 1; ELSE ZERO = 0;
                        IF (TMP < 0) NEGATIVE = 1; ELSE NEGATIVE = 0;
    PC = PC + 1
```

7	6	5	4	3	2	1	0
R1	R1	R2	R2	0	1	0	0

4. **sub r1, r2**

```
1       TMP = R1 - R2
2       R1 = TMP
3                   IF (TMP == 0) ZERO = 1; ELSE ZERO = 0;
4                   IF (TMP < 0) NEGATIVE = 1; ELSE NEGATIVE = 0;
5                   PC = PC + 1
```

7	6	5	4	3	2	1	0
R1	R1	R2	R2	0	1	1	0

5. **nand r1, r2**

```
1       TMP = R1 bitwise NAND R2
2       R1 = TMP
3                   IF (TMP == 0) ZERO = 1; ELSE ZERO = 0;
4                   IF (TMP < 0) NEGATIVE = 1; ELSE NEGATIVE = 0;
5                    PC = PC + 1
```

7	6	5	4	3	2	1	0
R1	R1	R2	R2	1	0	0	0

6. **ori imm5**

```
1                   TMP = K1 bitwise OR IMM5, where IMM5 is a 5 bit
                    ↪  constant
2                   K1 = TMP
3                   IF (TMP == 0) ZERO = 1; ELSE ZERO = 0;
4                   IF (TMP < 0) NEGATIVE = 1; ELSE NEGATIVE = 0;
5                   PC = PC + 1
```

7	6	5	4	3	2	1	0
IMM5_4	IMM5_3	IMM5_2	IMM5_1	IMM5_0	1	1	1

7. **shift{left, right} r1, r2**

```
     IF (L) THEN TMP = R1 << IMM2
            ELSE TMP = R1 >> IMM2
     R1 = TMP
IF (TMP == 0) ZERO = 1; ELSE ZERO = 0;
IF (TMP < 0) NEGATIVE = 1; ELSE NEGATIVE = 0;
PC = PC + 1
Alternative definition (equivalent to the previous one):
          IMM3 is a 3 bit immediate in the sign - magnitude
          ↪  representation, i.e., bit 2 = sign, value = bits 1 and 0
          IF (IMM3 > 0) THEN TMP = R1 << IMM3
                        ELSE TMP = R1 >> (-IMM3)
          R1 = TMP
IF (TMP == 0) ZERO = 1; ELSE ZERO = 0;
IF (TMP < 0) NEGATIVE = 1; ELSE NEGATIVE = 0;
PC = PC + 1
```

7	6	5	4	3	2	1	0
R1	R1	L/!R	IMM2_1	IMM2_0	0	1	1

8. bz imm4

```
     where IMM4 is a 2's complement 4-bit immediate
     IF (ZERO == 1) PC = PC + 1 + (SIGN-EXTEND8(IMM4))
     ELSE PC = PC + 1
```

7	6	5	4	3	2	1	0
IMM4_3	IMM4_2	IMM4_1	IMM4_0	0	1	0	1

9. bnz imm4

```
     IMM4 as in BZ
     IF (ZERO == 0) PC = PC + 1 + (SIGN-EXTEND8(IMM4))
     ELSE PC = PC + 1
```

7	6	5	4	3	2	1	0
IMM4_3	IMM4_2	IMM4_1	IMM4_0	1	0	0	1

10. **bpz imm4**

```
1       IMM4 as in BZ
2       IF (NEGATIVE == 0) PC = PC + 1 + (SIGN-EXTEND8(IMM4))
3       ELSE PC = PC + 1
```

7	6	5	4	3	2	1	0
IMM4_3	IMM4_2	IMM4_1	IMM4_0	1	1	0	1

A CPU can be broken down into two units: the datapath and the control path. The data path is where all the data manipulation occurs; it should be capable of computing all results of the instruction execution. The control path orchestrates signals sent to the data path (by means of a giant FSM).

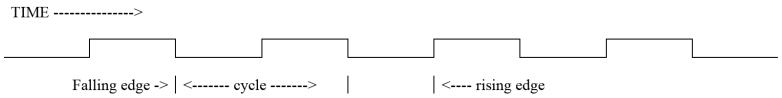


Figure 22. In the single-cycle processor each instruction will be executed entirely inside a single cycle. Code execution starts immediately after a falling edge and then all state changes will be complete right before the next falling edge.

CPU components include:

Definition 18

Register

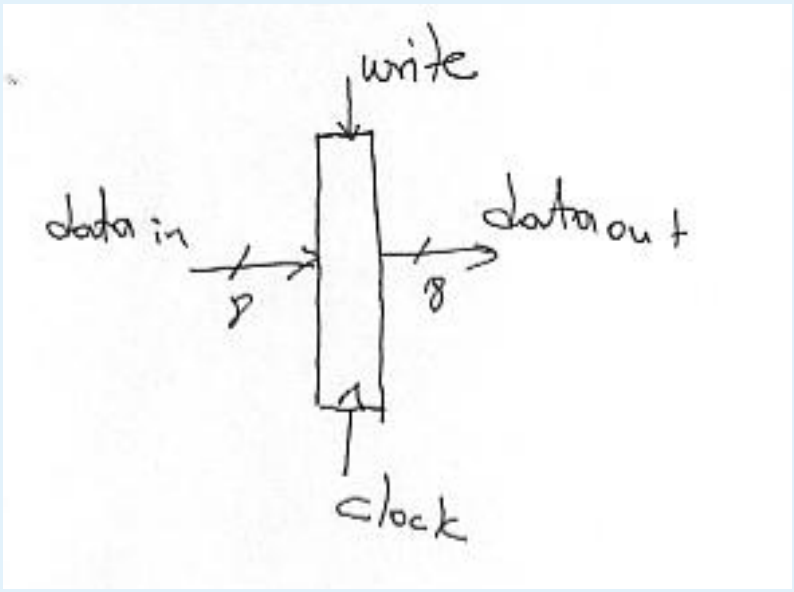


Figure 23. A collection of D flip-flops that can be read or written. Dataout gives values in the register, adding new values via datain. Write wire tells the register to latch onto the values in datain

Definition 19 Mux

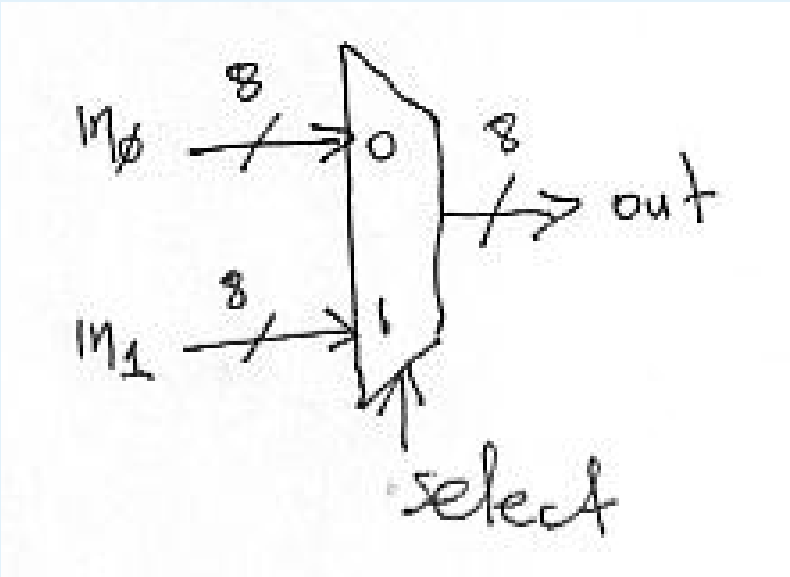


Figure 24. Multiplexes (Muxs) select a signal

Definition 20 Register File

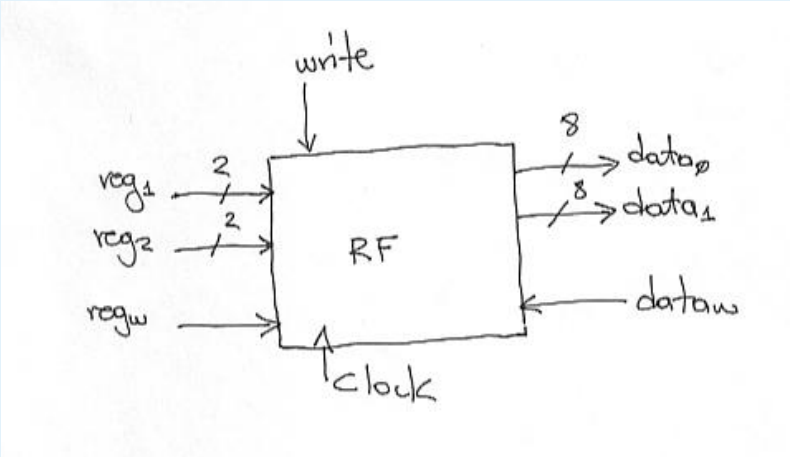


Figure 25. A register file contains a number of registers. Input signals to the register file cause it to output the data from the corresponding register to the data out lines. It also supports selecting and writing to a register by means of `regw` and `dataw`
. Note `write` signal as well.

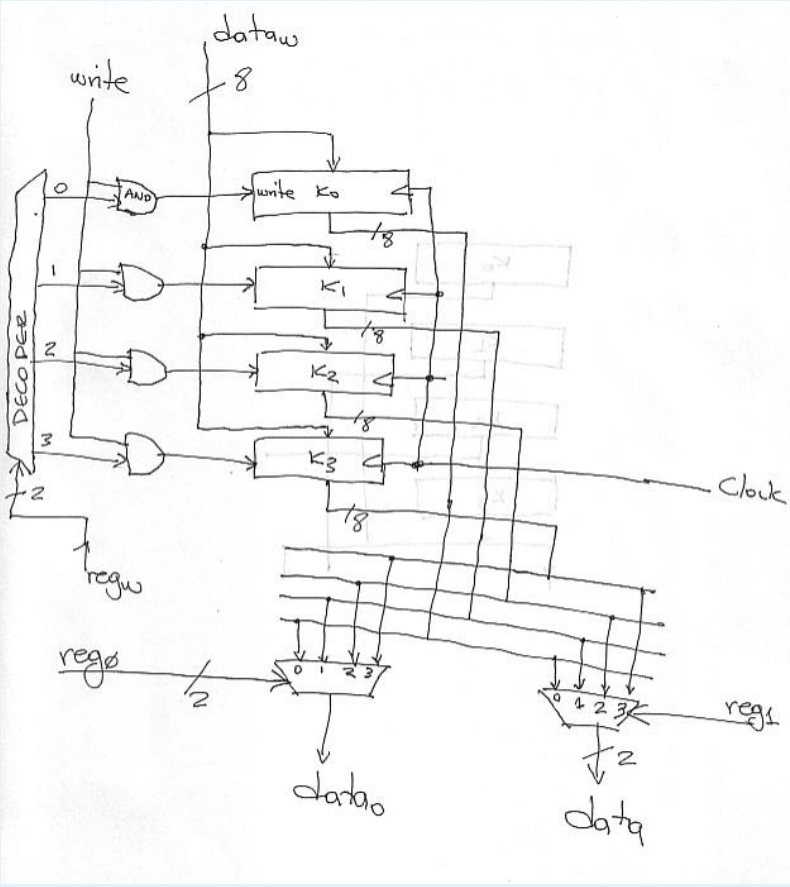


Figure 26. A register file implementation

Definition 21 ALU

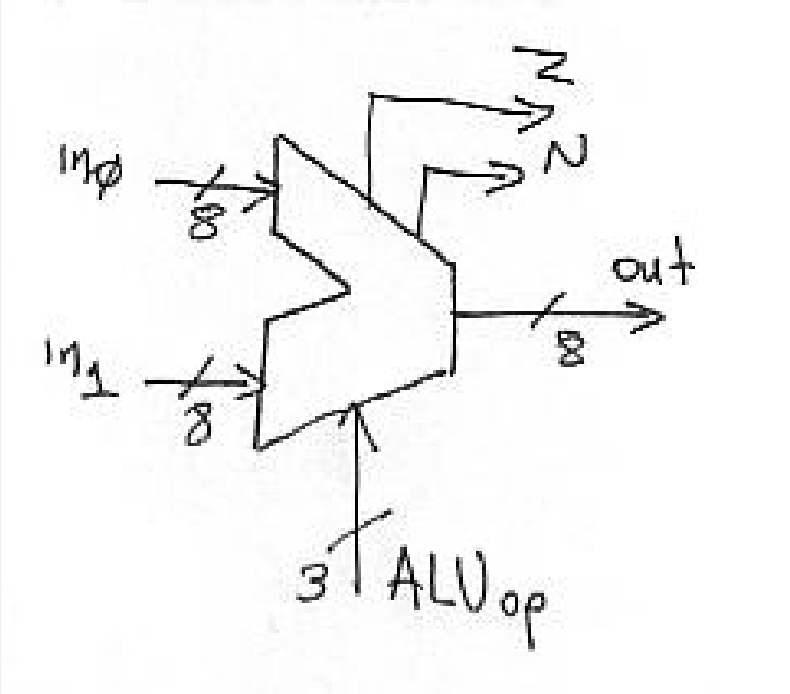


Figure 27. An ALU takes two inputs and then performs an operation as determined by the ALUOp signal. In this generalized ALU block we have Z, N signals which indicate if output values are zero or negative. Simpler ALUs can be used at times if things such as only addition is necessary

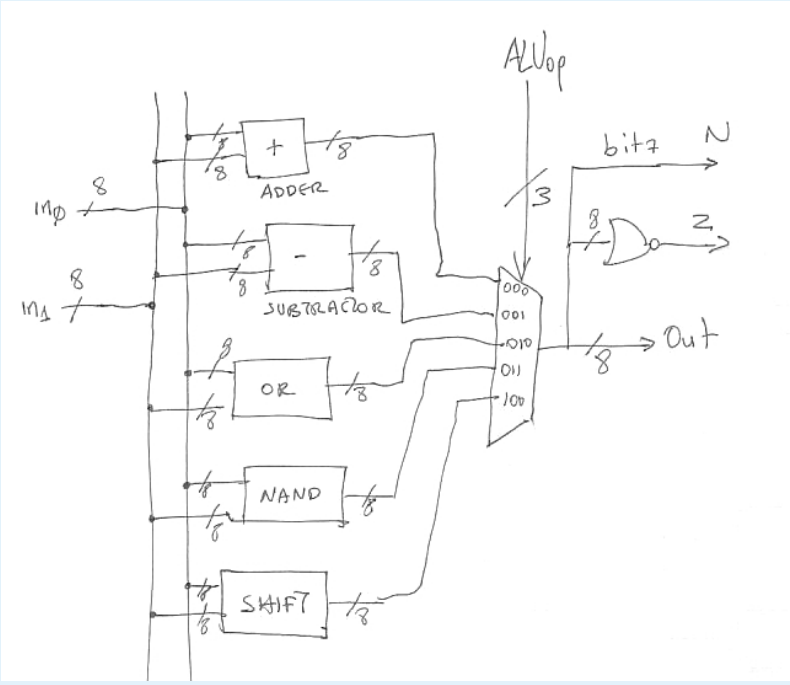


Figure 28. Possible impl

The and/sub boxes are fairly trivial, and so or OR/ANDs. Shift is a little more interesting

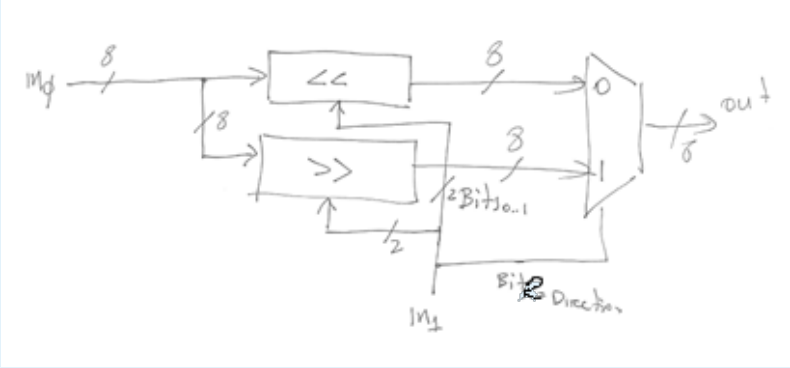


Figure 29. Shift is muxed on a shift left and a shift right

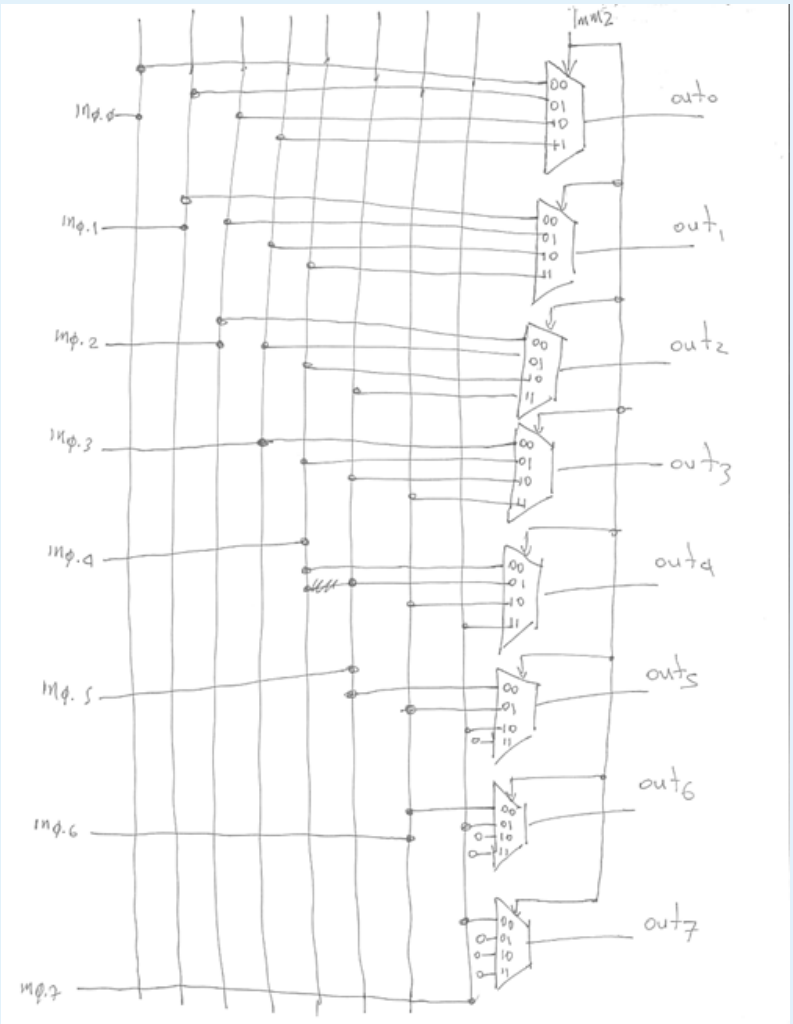
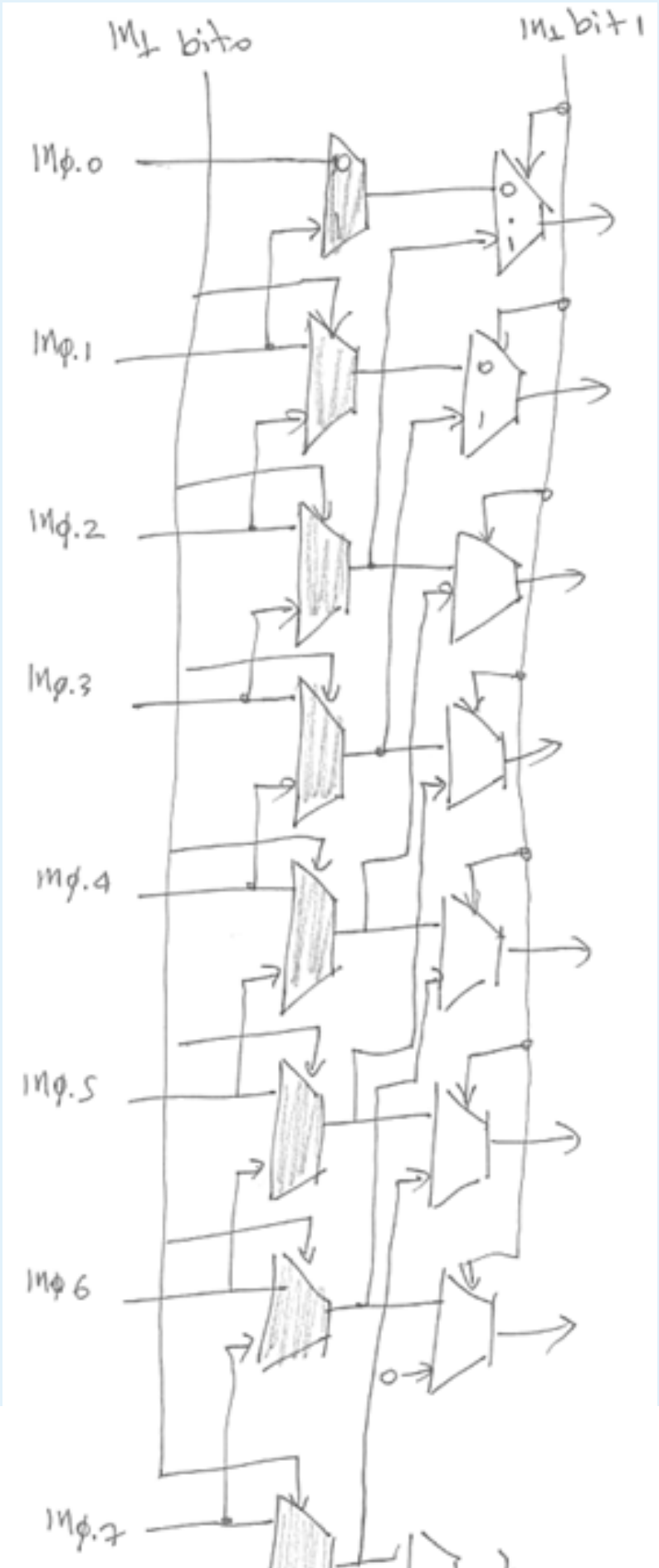


Figure 30. Shift left is a ton of muxes


IN bit 0
IN bit 1
IN.0
IN.1
IN.2
IN.3
IN.4
IN.5
IN.6
IN.7

Most modern processors use unified instruction and data memories. For the single-cycle processor we will use the *Harvard Architecture* which uses separate instruction and data memories.

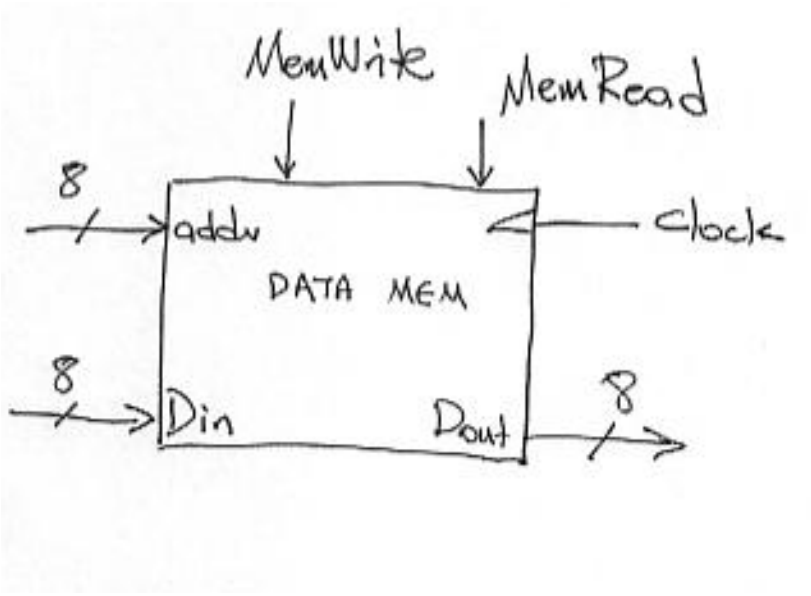


Figure 32. Memory

Data memory is capable of reading and writing (whereas instruction memory is read-only). Generally speaking it takes an address and depending on the value of MemWrite or MemRead it will either update the value at the address with `Din` or present the value at `Dout`

ADD

7	6	5	4	3	2	1	0
R1	R1	R2	R2	0	1	0	0

Figure 33. Recall: format of ADD instruction

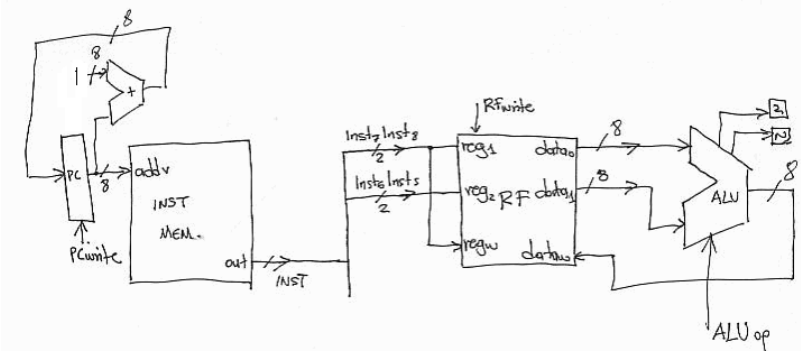


Figure 34. Datapath for ADD

Notice that bits 7 and 8 of the instruction are being piped directly into `reg1` and `regw` since our instruction definition gives `reg1 = reg1 + reg2`. The adder attached to the PC register is used to increment PC after each cycle.

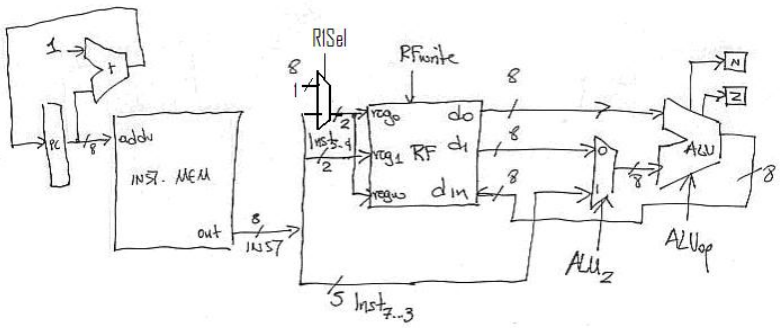


Figure 35. Augmented datapath to support ORI

ORI takes an immediate value from the instruction.

Subsection 8.8

Lecture 18: Modifying the single cycle processor

Subsection 8.9

Lecture 19: Multi-cycle processor

```
ADD R1, R2
```

1. Read mem[PC] into IR.

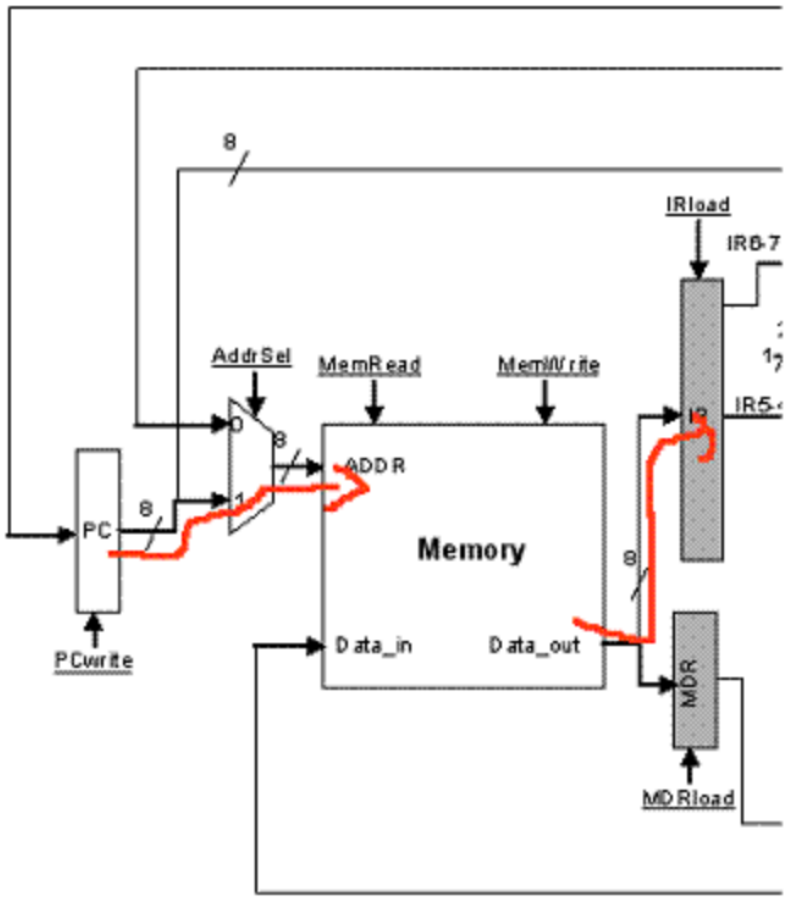


Figure 36. Cycle one for the ADD. This fetches the instruction by sending the address from PC register to MemRead and then storing the stored instruction into the Instruction Register IR

2. Decoding the instruction and reading from the register file

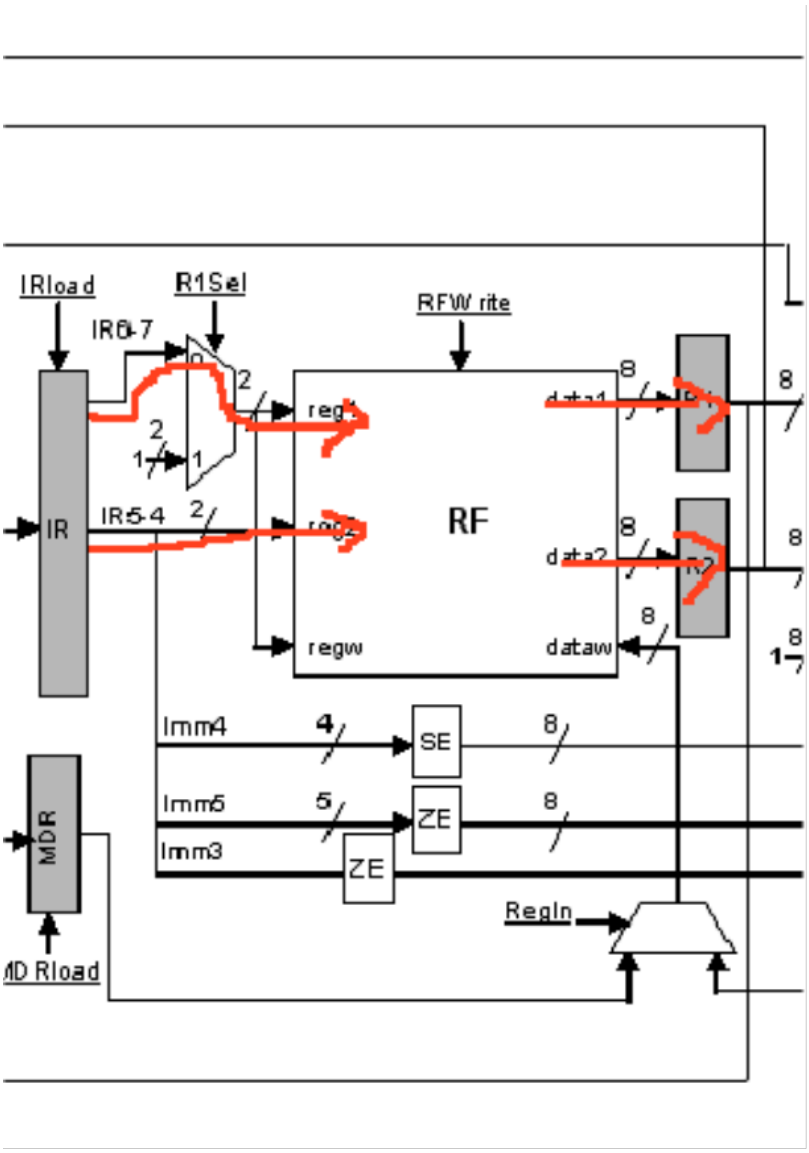


Figure 37. Control uses the instruction opcode to decide what happens during the next cycle. In this case, we read the two registers from the register file RF and then latch them into R1, R2

3. R1 = RF[127, 6], R2 = RF[IR5, 4]
4. AluOut = R1 + R2, update Z & N (Zero, Negative flags) This is done by muxing the latched values into the ALU (as well as the appropriate instruction as applicable) and then latching the ALU output and zero/negative flags. This step can be combined with the above step (discussed later).
5. RF[127, 6] = AluOut
6. Increment PC by 1. At this point the ALU is idle, so we can take a small detour and increment PC.

By the end of this, this just needs to ensure that the programmer sees the desired behaviour; that the values have been added and the PC has been incremented.

What are some optimizations?

- PC: reading at cycle 1, read at cycle 6. So we only need the current value at the beginning of cycles 1 and 6, so if we think about this we can probably speed things up by hoisting cycle 6 to cycle 1. So Cycle 1 can become PC = PC + 1 . This way we can save a clock cycle. While this may sound good on paper, can it happen during actual execution? In this case nobody else is using the ALU during cycle 1, so this would only necessitate a change in control. This optimization is always ideal, all the time.

LHS happens at end of clock cycle, RHS happens during

- Another optimization can happen during Cycle 2 and 3. It isn't certain that we can move Cycle 3 into Cycle 2; sometimes this will be helpful and sometimes it won't be helpful. But, it'll be helpful *always* and it won't destroy any state. So we can *speculate* that storing these results into the registers will *probably* be useful and combine steps and 2. This gives us

  1. PC = PC + 1
  2. Decode & Read from RF, R1 = RF[IR7, 6], R2 = RF[IR5, 4]
  3. AluOut = R1 + R2, update Z & N (Zero, Negative flags)
  4. RF[IR7, 6] = AluOut

There are situations where we speculate in a way that will destroy the state of the machine and then it turns that the speculation is not that helpful.

Comment Note that cycle 1 and cycle 2 must be the exact same for all instructions – this is because at that point we don't know what the instruction actually is. Only after instruction 2 do they change.

`ORI imm5`

1. Read mem[PC]
2. Decode the instruction and read from the register file (R1, R2). We've read both but we only need one of them. But it doesn't hurt that we already have R2.
3. R1 = RF[K1]
4. ALUout = R1 OR imm5 (zero-extended)

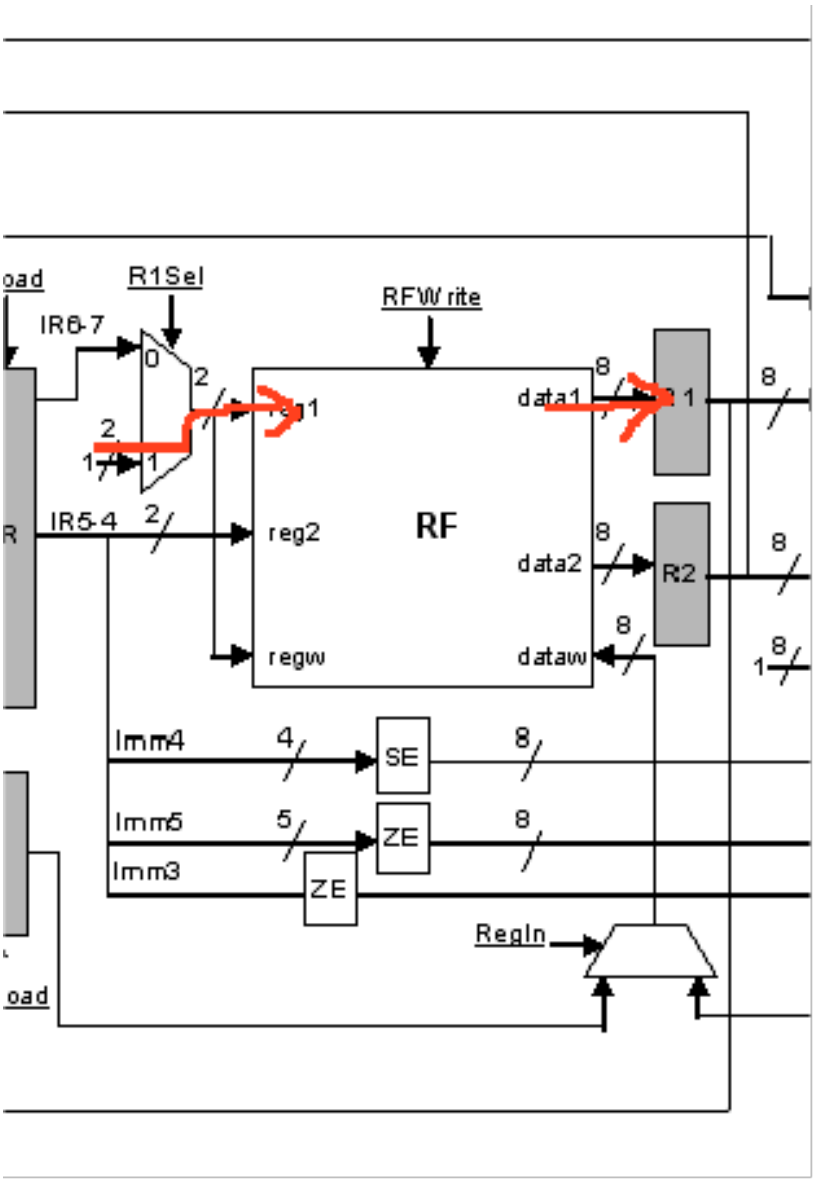


Figure 38. Note that imm5 is actually encoded into the instruction (see the control path)

5. RF[K1] = ALUout

LOAD

We could've actually speculated another way to favor the ORI instruction to make it take 4 cycles instead. But statistically speaking the NAND, ADD, and SUB instructions end up being used a lot more often and therefore more useful to optimize for

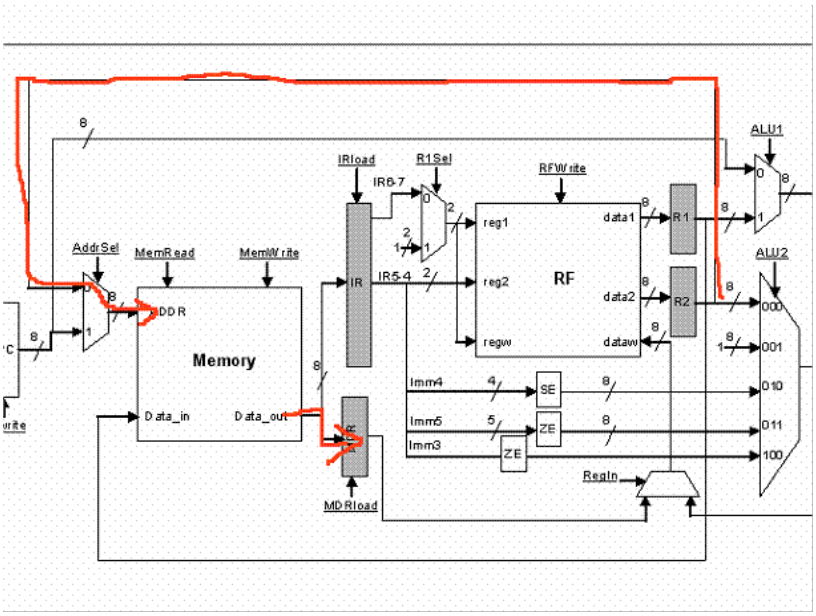


Figure 39. The LOAD instruction accesses memory during cycle 3 and then stores the returned value into MDR.

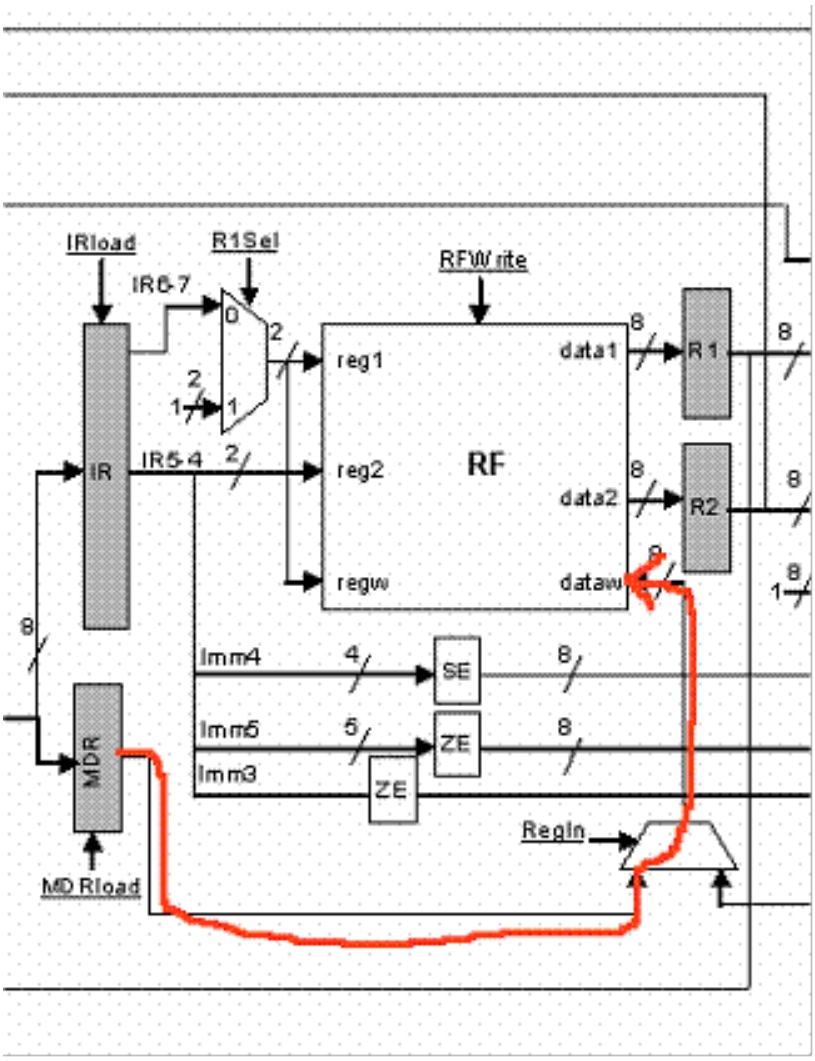


Figure 40. In cycle 4 the MDR value can be written to the register file.

`STORE`

1. PC = PC+1; grab value & +1 using ALU
2. Grab the two register values from the register file and latch them
3.

`BZ Imm4`

- Read mem[PC]
- Decode
- If (Z is true), PC = PC + 1 + SE(Imm4) [18]
-
-
- Decode
- Read

[18] *SE denotes sign extend*

Subsection 8.10

Lecture 20: Modifying the multi-cycle processor

Subsection 8.11

Caches

Caches are small and fast memories which contain a subset of the current contents of main memory. Modern processors usually include three levels of caches; L1, L2, and L3, with L1 being the fastest. Caches are also invisible to the ISA and the programmer with the exception of methods such as `ldwio`, `stwio`[19].

[19] *and* `volatile` *in* `c`*, etc, I guess*

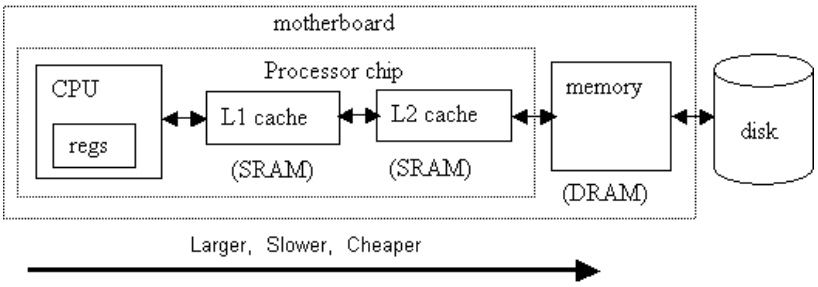


Figure 41. Cache and memory hierarchy

Caches exploit a common tendency of programs: spatial and temporal *locality*; recently referenced items are likely to be referenced again soon and nearby elements tend to be referenced close together in time.

For example, consider

```
1 for (int i = 0; i < n; i++){
2         sum += A[i];
3 }
```

This snippet displays spatial locality in the data access of the array and temporal locality in that the body of the loop composes a sequence of instructions that will be fetched and executed repeatedly. By storing a subset of memory into spatially local *blocks*, we can often reduce the amount of disk accesses by having a, well, cached copy available nearby.

Typical processor implementations have separate L1 caches for instructions and data, and a single (larger) L2 cache that is shared between the two.

Some keywords:

- cache capacity: the number of bytes the cache can hold
- placement: where the cache block should be placed
- identification: how to find a block in the cache
- cache hit: when the requested block is in the cache
- hit rate: total number of hits / total number of accesses
- cache miss: when the requested block is not in the cache

- miss rate: total number of misses / total number of accesses
- replacement: remove block to make room for new block
- replacement strategy: how to choose which block to remove
- write strategy: how to handle writes to the cache

Cache operation usually involves the following steps:

1. CPU loads from addr A
2. If cache hit -> return value at A stored in cache [DONE]
3. If cache miss -> load block containing A into cache
4. Place block in cache, replacing another block if necessary
5. Return value at A stored in cache [DONE]

Since cache is a subset of memory, we need to be able to identify which block of memory we're looking for. Cache lookups by convention break up the bits of a memory address into the *tag, set index, and offset.*

- *tag* (t) : since multiple locations can map to each cache block, the tag is used to distinguish between them i.e. which memory location currently is cached
- *set index* (s): how to locate a block in the cache: index into the array of sets containing blocks in the cache
- *offset* b: which byte within the block we're looking for

A *valid bit* is also associated for every block in the cache since a $0x0$ tag is a valid address.
Tag: $2^{n_{rows}} \rightsquigarrow$ tag $= 32 - N$. Note that the number of ways does not have to be a power of 2. We can have a 3-way cache, etc.

8.11.1 Implementations

Definition 22 **Direct mapped cache**: each memory location maps to a single specific location in the cache. For a byte-addressable address space of 2^n, then the size of a block is $B = 2^b$ bytes, the number of sets $S = 2^S$, and S is also equal to the total number of cache blocks in the cache; i.e. there is one block per set in cache. Capacity of a direct mapped cache is therefore $B \cdot S = 2^{b+s}$ bytes

Example

Consider a 16 bit address space with a direct-mapped cache with $t = 8, s = 4, b = 4$. The capacity of this cache is $2^{s+b} = 256$ bytes. The number of sets is $2^4 = 16$ and the block size $b = 2^4 = 16$.

Set Index	Valid?	Tag	Hex data values (for bytes 15..0) 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
0x0	0	0x00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x1	0	0x00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2	0	0x00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x3	0	0x00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x4	0	0x00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x5	0	0x00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x6	0	0x00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x7	0	0x00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x8	0	0x00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x9	0	0x00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xa	0	0x00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xb	0	0x00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xc	0	0x00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xd	0	0x00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xe	0	0x00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xf	0	0x00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Figure 42. Initial cache state

```
movia r8,0xface; ldb r8,0(r8)
```

1. Break address `0xface` to `tag = 0xfa`, `set = 0xc`, `offset = 0xe`.
2. The set-index is used to index the cache (`0xc`)
3. Check the valid bit if the block is valid. Initially this is invalid, so we will have a cache miss.
4. A request will be made to main memory to load the block containing the address `0xface`, i.e. the cache block that starts at $0xfac0$ and contains 16 bytes of data between $0xfac0 \rightsquigarrow 0xfacf$
5. Memory returns the block (after a delay) and the cache set at $0xc$ is updated with the new block; the contents are copied over, the valid bit set, and the tag is set equal to $0xfa$
6. Use the offset value $0xe$ t index into the cache block and return the requested block

A cache hit occurs when the tag matches and the valid bit is set. A more interesting case is a cache miss involving an overwrite.
Consider

Out of consideration of typing time, the full example will not be typed out and instead I defer the reader to find it here.

8.11.2 Handling Writes

What are ways to handle writes to the cache as to maximize performance i.e. minimize cache misses?

There are two things we can do:

1. write something
2. don't write anything

When do we pick one over the other? Some heuristics include

1. LRU cache: kick the least recently used block out of the cache[20]
2. Random: kick a random block out of the cache

[20] *Hashmap to doubly linked list, or ring buffer for fixed size*

PART

III

ECE355: Signal Analysis and Communication

Taught by Prof. Sunila Akbar

Section 9

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 (), independent is t
- Discrete: enclose in [], independent is n

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

Theorem 2

Energy for Complex Signals

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

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

Average Power for Complex Signals

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

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

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 23

Fundamental Period

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

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

Definition 24

Even signals

$$x(t) = x(-t) \tag{10.2}$$

Definition 25

Odd signals

$$x(t) = -x(t) \tag{10.3}$$

Theorem 3

Any signal can be broken into an even and odd component

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

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

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

Where C, a are complex numbers.

Two cases may occur.

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

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

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

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

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

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

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

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

$$P_{period} = \frac{E_{period}}{T_0} = 1 \tag{10.9}$$

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;

$$C = Ce^{at} \tag{10.10}$$

and

$$a = r + j\omega_0 \tag{10.11}$$

can be combined to give

Definition 26

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

Euler's relation can be used to simplify this to

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

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 ω_0	identical signals for distinct $\omega_0 \in \{\omega_0 \pm 2\pi i, i \in \mathbb{Z}\}$
Periodic for any ω_0	Periodic only if $\omega_0 = \frac{2\pi m}{N}$ for integers $N > 0, m$
Fundamental frequency ω_0	Fundamental frequency $\frac{\omega_0}{m}$
Fundamental period $\omega_0 = 0 \to$ undefined, otherwise $T_0 = \frac{2\pi}{\omega_0}$	Fundamental period $\omega_0 = 0 \to$ undefined, otherwise $T_0 = m\frac{2\pi}{\omega_0}$
	Since unique ω 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**[21] function, $\delta[n]$

[21] or unit sample

Definition 27

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

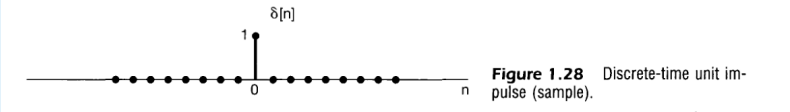


Figure 1.28 Discrete-time unit impulse (sample).

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

Definition 28

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

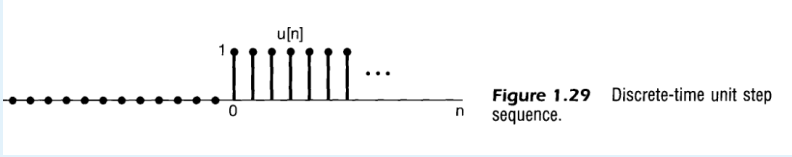


Figure 1.29 Discrete-time unit step sequence.

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

$$\delta[n] = u[n] - u[n-1] \tag{11.3}$$

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

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

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

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

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 $n \neq n_0$

$$x[n]\delta[n-n_0] = x[n_0]\delta[n-n_0] \tag{11.6}$$

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

Definition 29

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

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

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

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 30

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

(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 Δ short enough to not matter for any practical purpose

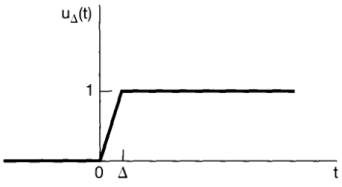


Figure 1.33 Continuous approximation to the unit step, $u_\Delta(t)$.

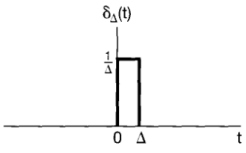


Figure 1.34 Derivative of $u_\Delta(t)$.

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

$$u(t) = \int_0^\infty \delta(t-\sigma)d\sigma \tag{11.10}$$

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,

$$x(t)\delta t(t-t_0) = x(t_0)\delta(t-t_0) \tag{11.11}$$

Subsection 11.2

Lecture 5

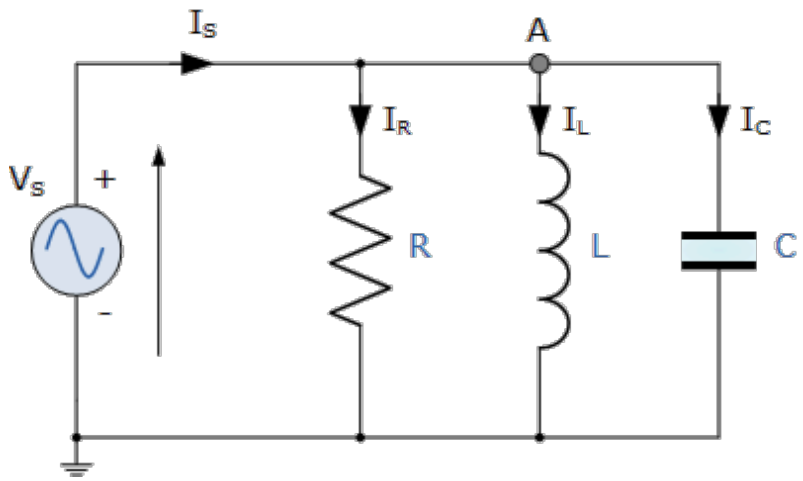
Definition 31 A **system** is a process that transforms a signal, i.e.

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

A similar definition can be applied for the discrete case

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;



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

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

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

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

11.2.1 Types of systems

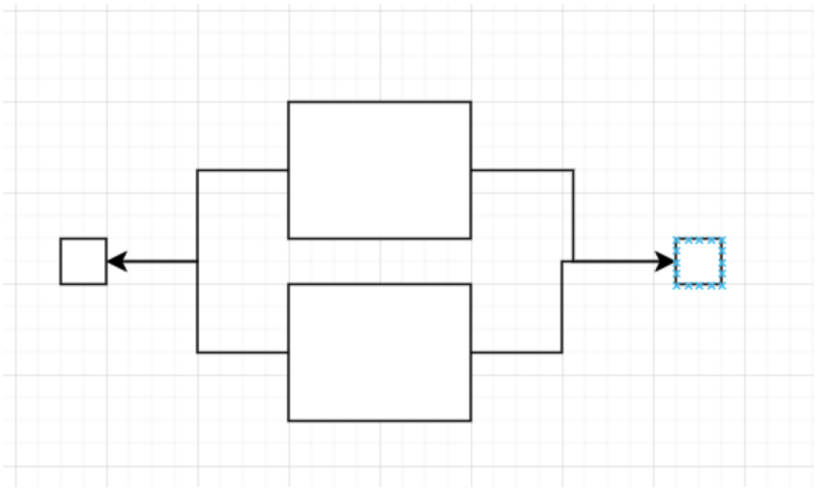


Figure 43. Parallel systems

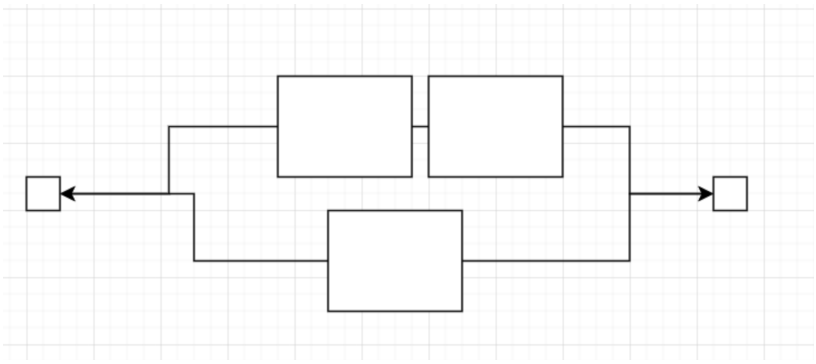


Figure 44. Series-parallel

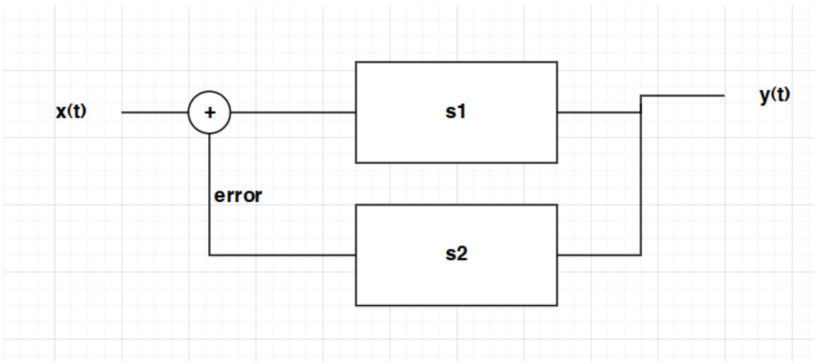


Figure 45. Feedback control system

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

11.2.2 System properties

Definition 32 **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,

$$y[n] = 2x[n] - x^2[n] \tag{11.15}$$

is a memoryless system, but

$$y[n] = 2x[n] - x^2[n-1] \tag{11.16}$$

is not.

Other simple memoryless systems include the identity system $x(t) = y(t)$
A system with memory is the *accumulator*

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

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

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

There can also be systems that are dependent on future values of the input and output[22].

[22] *PID go brr*

Definition 33 A system is *invertible* if distinct inputs lead to distinct outputs[23] For example, the identity system is invertible, but the accumulator is not.

[23] *Recall: MAT185 and matrix investability*

Definition 34 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 casual

Though casual systems are useful, non-casual 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 35 A system is *stable* is if the output is bounded if the input is bounded.

Definition 36 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 invarient if a time shift in the input signal results in an identical time shift in the output.
If

$$x[n] \xrightarrow{\text{sys}} y[n] \tag{11.19}$$

Then, for a time invariant system,

$$y[n - n_0] \xrightarrow{\text{sys}} x[n - n_0] \tag{11.20}$$

Definition 37 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

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

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.

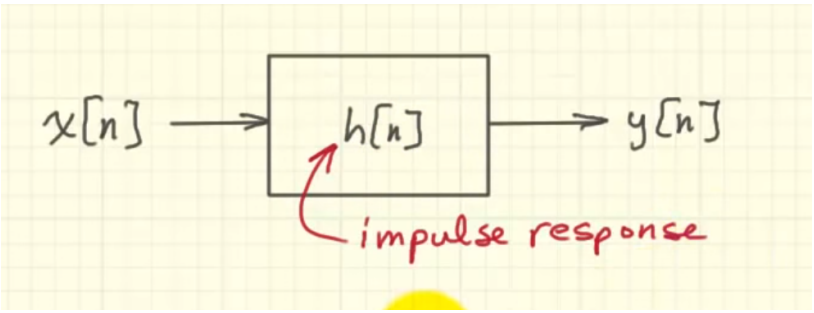
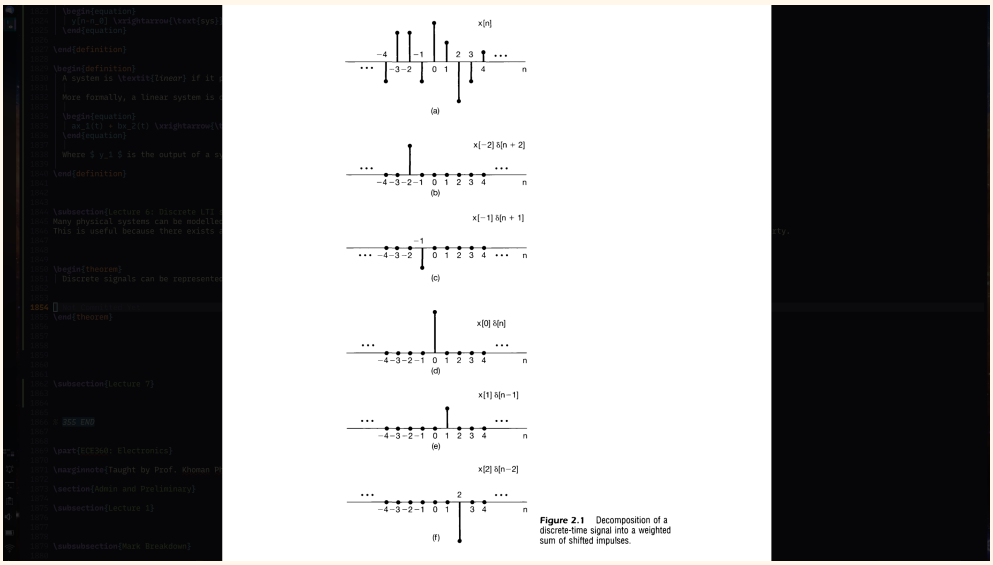


Figure 46. In this lecture the idea of LTI systems being a sum of $h[n]$ impulse responses will be

Theorem 6 Discrete signals can be represented as a summation of impulses.



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

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

Example As an example we can rewrite the unit step function as

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

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 38 **Convolution** sum

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

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

$$y[n] = x[n] * h[n] \tag{12.4}$$

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 $k = n$

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 it's duration is inconsequential for any real system.

Definition 39 **Sifting property** of continuous-time impulse

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

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

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

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

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

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

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

For notational convenience the τ subscript can be dropped and generalizing to all t

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

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

SUBSECTION 12.3

Lecture 12

A class of LTI systems that are really important are ones related through linear constant-coefficient differential equations. For example, consider

$$\frac{dy(t)}{dt} + 2y(t) = x(t) \tag{12.10}$$

This equation offers an implicit specification of the system, i.e. a relationship between input and output instead of an explicit expression; to find a explicit expression we must plug in initial conditions.

Given the exponential input

$$x(t) = Ke^{3t}u(T) \tag{12.11}$$

to (12.10), the complete solution to a differential equation of this type is given as the sum of a particular and homogeneous solution[24]

[24] *Recall ESC195*

$$y(t) = y_p(t) + y_h(t) \tag{12.12}$$

A common method for finding the particular solution is to first look for the *forced response*; an output signal with the same form as the input.

Take

$$y_p(t) = Ye^{3t} \tag{12.13}$$

and then doing some math we find $Y = \frac{K}{5}$ to arise at $y_p(t) = \frac{K}{5}e^{3t}, t > 0$.

To find the homogeneous solution we hypothesize a solution of form

$$y_h(t) = Ae^{st} \tag{12.14}$$

and then substitute into the differential equation to find

$$y(t) = Ae^{-2t} + \frac{K}{5}e^{3t}, \qquad t > 0 \tag{12.15}$$

The constants A, K can then be found by substituting in the initial conditions.

More generally speaking, a Nth order linear constant-coefficient differential equation is given by

$$\sum_{k=0}^{N} a_k \frac{d^k y(t)}{dt^k} = \sum_{k=0}^{M} b_k \frac{d^k x(t)}{dt^k} \tag{12.16}$$

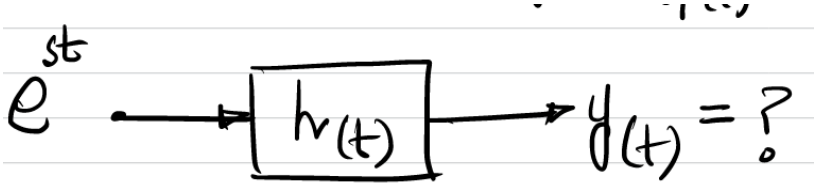
The discrete time counterpart is

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k] \tag{12.17}$$

With the solution $y[n]$ being the sum of the particular solution to the problem above and a solution to the homogeneous equation

$$\sum_{k=0}^{N} a_k y[n-k] = 0 \tag{12.18}$$

A important special case of LCCDEs are LTIs.
Consider a response to an arbitrary exponential input signal



Definition 40 **Laplace transform of** h, or the transfer function of LTI systems

$$y(t) = \int_{-\infty}^{\infty} h(\tau)e^{-s\tau}d\tau = \int_{0}^{1} e^{-s\tau}d\tau = \begin{cases} \frac{1}{s}(1-e^{-s}) & s \neq 0 \\ 1 & s = 0 \end{cases} \tag{12.19}$$

Theorem 8 If $x(t) = e^{st}$ and H_s exists, $y(t) = H(s)e^s t$
This can be interpreted in two ways

1. Same constant exponential as input e^{st} (eigenfunction of LTI system)
2. Scaled by constant $H(s)$ (eigenvalue)

Corollary
If the input to an LTI system with an impulse response $h(t)$ is

$$x(t) = \sum_k a_k e^{s_k t} \tag{12.20}$$

then it's output is given by

$$y(t) = \sum_k a_k H_{s_k} e^{s_k t} \tag{12.21}$$

Since almost all periodic signals can be expressed as a sum of harmonically related complex exponentials, i.e.

$$x(t) = \sum_{k=-\infty}^{\infty} a_k e^{jk\omega_t} = \sum_{k=-\infty}^{\infty} a_k e^{j\frac{2\overline{n}k}{T}t} \tag{12.22}$$

Where the k denotes the $|k|$th harmonic component and a_0 being the constant component[25]

[25] *In circuit analysis this is the constant component*

SUBSECTION 12.4

Lecture 15: Which periodic signals have a Fourier representation?

We have the following expressions for synthesis and analysis, respectively

$$x(t) = \sum_{k=-\infty}^{\infty} a_k e^{kj\omega_t} \tag{12.23}$$

$$a_k = \frac{1}{T}\int_T x(t)e^{-kj\omega_t}dt \tag{12.24}$$

SUBSECTION 12.5

Fourier series representation of continuous periodic systems

$$x(t) = \cos\omega_o t \Leftrightarrow e^{j\omega_o t} \tag{12.25}$$

Both of the signals are periodic with fundamental frequency ω_o with associated harmonically related complex exponentials

$$\Phi_k(t) = e^{jk\omega_o t}, \qquad k = 0, \pm 1, \pm 2 \tag{12.26}$$

TABLE 4.1 PROPERTIES OF THE FOURIER TRANSFORM

Section	Property	Aperiodic signal	Fourier transform
		$x(t)$ $y(t)$	$X(j\omega)$ $Y(j\omega)$
4.3.1	Linearity	$ax(t) + by(t)$	$aX(j\omega) + bY(j\omega)$
4.3.2	Time Shifting	$x(t - t_0)$	$e^{-j\omega t_0}X(j\omega)$
4.3.6	Frequency Shifting	$e^{j\omega_0 t}x(t)$	$X(j(\omega - \omega_0))$
4.3.3	Conjugation	$x^*(t)$	$X^*(-j\omega)$
4.3.5	Time Reversal	$x(-t)$	$X(-j\omega)$
4.3.5	Time and Frequency Scaling	$x(at)$	$\frac{1}{\lvert a\rvert}X\left(\frac{j\omega}{a}\right)$
4.4	Convolution	$x(t) * y(t)$	$X(j\omega)Y(j\omega)$
4.5	Multiplication	$x(t)y(t)$	$\frac{1}{2\pi}\int_{-\infty}^{+\infty}X(j\theta)Y(j(\omega-\theta))d\theta$
4.3.4	Differentiation in Time	$\frac{d}{dt}x(t)$	$j\omega X(j\omega)$
4.3.4	Integration	$\int_{-\infty}^{t}x(t)dt$	$\frac{1}{j\omega}X(j\omega) + \pi X(0)\delta(\omega)$
4.3.6	Differentiation in Frequency	$tx(t)$	$j\frac{d}{d\omega}X(j\omega)$
4.3.3	Conjugate Symmetry for Real Signals	$x(t)$ real	$X(j\omega) = X^*(-j\omega)$ $\Re e\{X(j\omega)\} = \Re e\{X(-j\omega)\}$ $\Im m\{X(j\omega)\} = -\Im m\{X(-j\omega)\}$ $\lvert X(j\omega)\rvert = \lvert X(-j\omega)\rvert$ $\measuredangle X(j\omega) = -\measuredangle X(-j\omega)$
4.3.3	Symmetry for Real and Even Signals	$x(t)$ real and even	$X(j\omega)$ real and even
4.3.3	Symmetry for Real and Odd Signals	$x(t)$ real and odd	$X(j\omega)$ purely imaginary and odd
4.3.3	Even-Odd Decomposition for Real Signals	$x_e(t) = \mathcal{E}v\{x(t)\}$ [$x(t)$ real] $x_o(t) = \mathcal{O}d\{x(t)\}$ [$x(t)$ real]	$\Re e\{X(j\omega)\}$ $j\Im m\{X(j\omega)\}$
4.3.7	Parseval's Relation for Aperiodic Signals $\int_{-\infty}^{+\infty}\lvert x(t)\rvert^2 dt = \frac{1}{2\pi}\int_{-\infty}^{+\infty}\lvert X(j\omega)\rvert^2 d\omega$		

Figure 47. a pretty good table

PART

IV

ECE358: Foundations of Computing

Taught by Prof. Shurui Zhou

Section 13

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??
- Θ

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

Subsection 13.2

Complexities

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 41

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

$$O(g(n)) = \{f(n) : \exists \quad c, n_0 \quad s.t. \quad 0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_o\} \tag{13.1}$$

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

$$n! \leq n \cdot n \cdot n \cdot n \ldots n = n^n \Rightarrow n! \in O(n^n) \tag{13.2}$$

□

Definition 42 Big Ω notation (lower bound)
$h(n)$ is an asymptotic lower bound for $f(n)$ if:

$$\Omega(h(n)) = \{f(n) : \exists \quad c, n_0 > 0 \quad s.t. \quad 0 \leq c \cdot h(n) \leq f(n), \forall n \geq n_o\} \tag{13.3}$$

Proof **Find** Θ **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 $\frac{n}{2} \ldots n$ and then find the bound

$$\begin{aligned} f(n) &= 1 + 2 + 3 \ldots + n \\ &\geq \lceil \frac{n}{2} \rceil + (\left\lceil \frac{n}{2} \right\rceil + 1) + (\left\lceil \frac{n}{2} \right\rceil + 2) + \ldots n \quad n/2 \text{ times} \\ &\geq \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{n}{2} \right\rceil + \ldots \left\lceil \frac{n}{2} \right\rceil \\ &\geq \frac{n}{2} \cdot \frac{n}{2} \\ &= \frac{n^2}{4} \end{aligned} \tag{13.4}$$

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

□

Definition 43 Big Θ notation (asymptotically tight bound)

$$\Theta(g(n)) = \{f(n) : \exists \quad c_1 c_2, n_0 \quad s.t. \quad 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_o\} \tag{13.5}$$

Proof Prove that

$$\sum_{j=1}^{n} i^k = \Theta(n^{k+1}) \tag{13.6}$$

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

$$\begin{aligned} f(n) = \sum_{j=1}^{n} i^k &= 1^k + 2^k + \ldots n^k \\ &\leq n^k + n^k + \ldots n^k \\ &= n^{k+1} \end{aligned} \tag{13.7}$$

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

$$
\begin{aligned}
f(n) = \sum_{j=1}^{n} i^k &= 1^k + 2^k + \dots n^k \\
&= n^k + (n_1)^k + \dots 2^k + 1^k = \sum_{i=1}^{n} (n - i + 1)^k \\
&\geq \frac{n}{2}^k * n \geq \frac{n^{k+1}}{2^k} = \Omega(n^{k+1})
\end{aligned} \tag{13.8}
$$

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 9

Properties of asymptotes:
Note: $\wedge$ means AND
Transitivity [26]

$$(f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n))) \Rightarrow f(n) = \Theta(h(n)) \tag{13.9}$$

[26] *The following applies to* O, Θ, o, ω

Reflexivity[27]

$$f(n) = \Theta(f(n)) \tag{13.10}$$

[27] *The following applies to* O, Θ

Symmetry

$$f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n)) \tag{13.11}$$

Transpose Symmetry

$$
\begin{aligned}
f(n) = O(g(n)) &\iff g(n) = \Omega(f(n)) \\
f(n) = o(g(n)) &\iff g(n) = \omega(f(n))
\end{aligned} \tag{13.12}
$$

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.

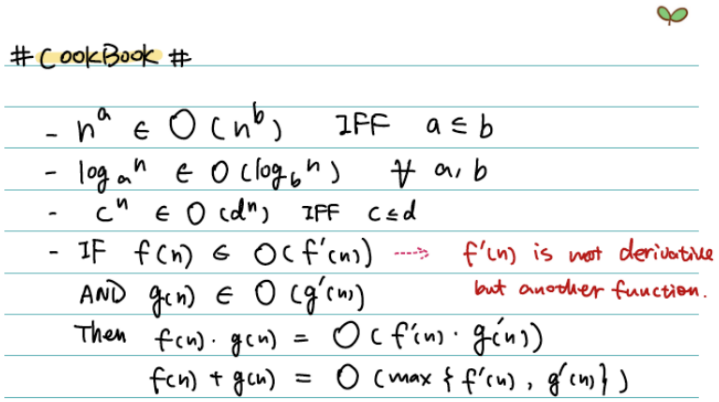


Figure 48. Complexity Cookbook

Subsection 13.3

Lecture 3: Logs & Sums

13.3.1 Functional Iteration

Recall:

$$a = b^c \Leftrightarrow log_b a = c \quad (13.13)$$

$f^{(i)}(n)$ denotes a function iteratively applied i times to value n.
For example, a function may be defined as:

$$f^{(i)}(n) = \begin{cases} f(n) & \text{if } i = 0 \\ f(f^{(i-1)}(n)) & \text{if } i > 0 \end{cases} \quad (13.14)$$

Given (13.14) we see that

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

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

$$lg^*(n) = \min\{i \geq 0 : lg^{(i)} n \leq 1\} \quad (13.15)$$

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

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

Summations & Series

Proof | Proof for a finite geometric sum:

$$\sum_{k=0}^{n} x^k = S$$
$$S = 1 + x + x^2 \dots x^n$$
$$xS = x + x^2 + x^3 \dots x^{n+1} \tag{13.16}$$
$$S = \frac{1 - x^{n+1}}{1 - x}$$

□

$$\sum_{i=1}^{\infty} x^i = \frac{1}{1-x} \quad \text{if } |x| < 1 \tag{13.17}$$

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \quad \text{if } |x| < 1 \tag{13.18}$$

Proof Begin by differentiating both sides over x

$$\sum_{k=0}^{\infty} x^k = \frac{1}{(1-x)} \quad \text{if } |x| < 1 \tag{13.19}$$

$$\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2} \tag{13.20}$$

And then multiply both sides by x, therefore (13.18) follows. □

Telescoping Series

$$\sum_{k=1}^{n} a_k - a_{k-1} = a_n - a_0 \tag{13.21}$$

Proof Write it out and cancel out terms

$$(a_1 - a_0) + (a_2 - a_1) \dots (a_n - a_{n-1}) = a_n - a_0 \tag{13.22}$$

Therefore the sum telescopes □

Another telescoping series may be proved similarly:

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} \xrightarrow{math} \sum_{k=1}^{n-1} (\frac{1}{k} - \frac{1}{k+1}) = (1 - \frac{1}{2}) + \dots (\frac{1}{n-1} - \frac{1}{n}) = a_o - a_n \tag{13.23}$$

Subsection 13.4

Lecture 4: Induction & Contradiction

13.4.1 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. $0, 1, 2, 3, 12, \ldots$ 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 $P(n) = 1 + 2 + 3 + \ldots n = \frac{n(n+1)}{2}$

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 $P(k+1) = \frac{(n+1)(n+2)}{2}$): $P(k+1) = \underbrace{1 + 2 + \ldots + n}_{\frac{n(n+1)}{2}} + (n+1) = \ldots = \frac{n^2+3n+2}{2} = \frac{(n+1)(n+2)}{2}$

□

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)

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

Proof

1. Basis:
$$n = 0, |S| = 0, |2^S| = 1 = 2^0 \tag{13.24}$$
$$n = 1, |S| = 1, |2^S| = 2 = 2^1 \tag{13.25}$$
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 $B = S \not\{a\}$ 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 $2^B \cup \{a\}, whichis 2^n$.

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

□

The same kind of argument can be applied to problems such as the Towers of Hanoi and the tiling problem.

13.4.2 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

$$\sqrt{2} = \frac{a}{b} \tag{13.26}$$

Where a, b **have no common factors**.
We can square both sides

$$2 = \frac{a^2}{b^2} \to a^2 = 2b^2 \tag{13.27}$$

Therefore a^2 is even.
Let $a = 2c$

$$2^2c^2 = 2b^2 \to b^2 = 2c^2 \tag{13.28}$$

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). □

Subsection 13.5

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 origianl but smaller in size, solve them resurisively, then combine them to create a solution to the original problem.

Definition 44

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;
$T(n) = T(n-1) + T(n-2)$ 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 45 Pseudocode for mergesort is given by:

```
mergesort(A, p, r)
        if p < r
                q = [(p+r)/2]
                mergesort(A, p, q)  // N/2
                mergesort(A, q+1, r)  // N/2
                merge(A, p, q, r) // merge the sorted
                ↪  subarrays
```

This mergesort partitions in half each time[28].
In the worst case we will compare $N-1$ times, so $O(N)$ worst case.
Proving that merge sort is $\Omega(N)$

[28] *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:

$$T(N) = \begin{cases} O(1) & n = 1 \\ T(N) = 2T(\frac{N}{2} + \Theta(N) \end{cases} \tag{13.29}$$

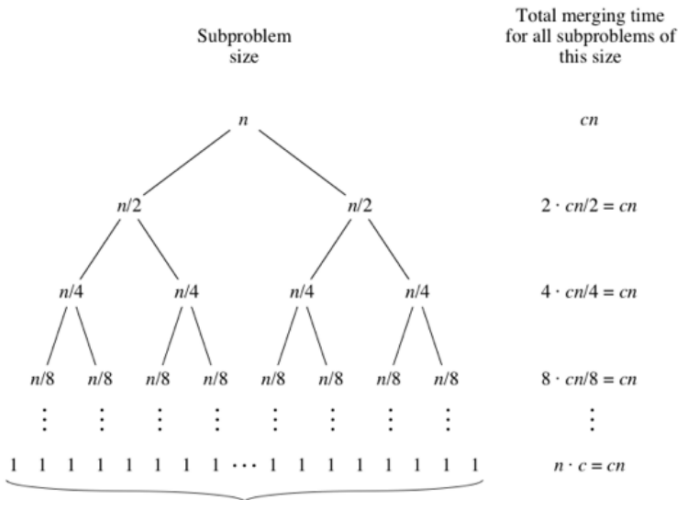
Note that $\Theta(N)$ is for the merge operation

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

- Recurrence trees
- Substitution
- Master theorem

13.5.1 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

$$N \cdot logN \tag{13.30}$$

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

13.5.2 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.

$$T(n) = 2T(\frac{n}{2}) + N \tag{13.31}$$

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 $T(1) = 1$, $T(2) = 2T(1) + 2 = 4, T(3) = 2T(1) + 3 = 5, T(4) = 2T(2) + 4 \ldots$.

   So $T(n)$ is therefore independent of $T(1)$, so we can use two bases, $T(2), T(3)$. Since $T(2) \leq C * 2 \log 2 = 2C, T(3) \leq C \cdot \log$
2. Hypothesis: Assume that the upper bound holds for all possible $m < n$, let $m = \lfloor \frac{n}{2} \rfloor$. This yields $T(\frac{n}{2}) \leq C \cdot \lfloor \frac{n}{2} \rfloor \cdot \log \lfloor \frac{n}{2} \rfloor$
3. Inductive step: substitute hypothesis into recurrence yields

$$T(N) \leq C \cdot (C \cdot \left\lfloor \frac{N}{2} \right\rfloor \cdot \log \left\lfloor \frac{N}{2} \right\rfloor + N = cN \log N - (1-c)N \leq Cn \log n \tag{13.32}$$

□

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

$$T(N) \leq 2C \cdot \left\lfloor \frac{n}{2} \right\rfloor + n = cn + n = (c+1)n \tag{13.33}$$

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

13.5.3 Master Theorem

Definition 46 The master method applies to recurrences of the form

$$T(n) = aT(\frac{n}{b}) + f(n) \qquad \text{where } a \geq 1, b > 1, f \text{asymptotically positive} \tag{13.34}$$

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 $f(n) = \Theta(n^{\log_b a})$, 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 $af(\frac{n}{b}) \leq cf(n)$ 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 a ware of. In each example we compare $f(n)$ with $F = n^{\log_b a}$ and take the larger of each as the solution 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^{ε}. 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.

$$\log_b a = \log_{\frac{2}{3}} 1 = 0 \tag{13.35}$$

$$f(n) = \Theta(n^0) \tag{13.36}$$

So

$$T(n) = O\log(n) \tag{13.37}$$

□

Subsection 13.6

Lecture 6

13.6.1 Graphs

Definition 47 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.

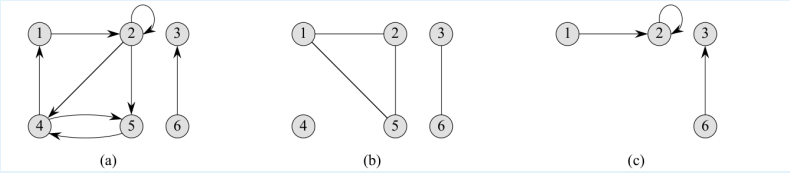


Figure 49. (a) directed graph, (b) undirected graph, (c) a subgraph of (a)

Some conventions:

- Edges are denoted by (u, v)[29] 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 $(v_0, v_1, \ldots v_k)$ from from vertex to another such that each vertex is incident[30] 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

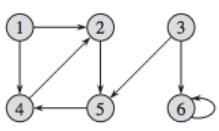
[29] *not* $\{u, v\}$

[30] *with the exception of start/end vertices*

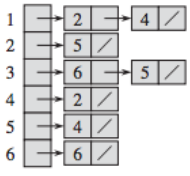
  - A path forms a cycle if the first and last vertices are the same
- A directed graph with no self-loops is **simple**
- A graph with no simple cycles is acyclic
- An undirected graph is **connected** if there exists a path between any two vertices
- A directed graph is **strongly connected** if every vertex is reachable from every other vertex
- Two graphs V, V' are **isomorphic** if there exists a bijection[31] 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 $E' = \{(u, v) \in E | u, v \in V'\}$
- 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

[31] $f : V \to V'$, *i.e. we can relabel the vertices of* V *to be those of* V' *and the two graphs would be identical*

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



(a)



(b)

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

(c)

Table 4. Time complexities of graph representations

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

13.6.2 Trees

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

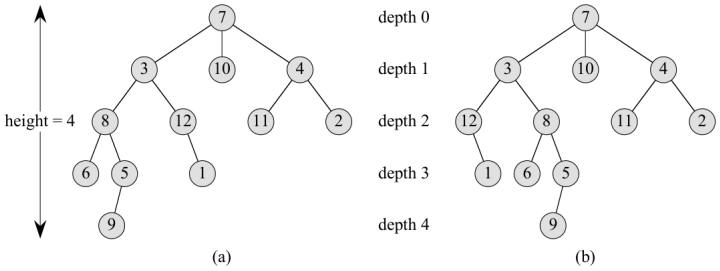


Figure B.6 Rooted and ordered trees. **(a)** A rooted tree with height 4. The tree is drawn in a standard way: the root (node 7) is at the top, its children (nodes with depth 1) are beneath it, their children (nodes with depth 2) are beneath them, and so forth. If the tree is ordered, the relative left-to-right 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 48 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 10 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 13.7

Lecture 7, 8: Probability and Counting

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

Definition 49 Probability distribution $Pr\{\}$: 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 50

$$Pr\{A \cup B\} \leq Pr\{A\} + Pr\{B\}$$

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

Definition 51

Baye's thereom

$$Pr\{A \mid B\} = \frac{Pr\{B \mid A\} Pr\{A\}}{Pr\{B\}} = \frac{Pr\{A\} Pr\{B|A\}}{Pr\{A\} Pr\{B|A\} + Pr\left\{\overline{A}\right\} Pr\left\{B|\overline{A}\right\}} \quad (13.38)$$

The expected value of a random variable is

$$E[X] = \int_{-\infty}^{\infty} xPr\{X = x\}\, dx \quad (13.39)$$

And in the discrete case,

$$E[X] = \sum_{x \in \mathbb{Z}} xPr\{X = x\} \quad (13.40)$$

The variance is

$$Var[X] = E\left\{(X - E[X])^2\right\} = E[X^2] - E^2[X] \quad (13.41)$$

The **standard deviation,** σ**,** is the positive square root of the variance.

Subsection 13.8

Lecture 9: Heapsort

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

This should be review from ESC190

Definition 52

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 it's ability to be represented as an array with `Parent(i) = floor(i/2)`, `Left(i) = 2iRight(i) = 2i+1`

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[32], 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.

[32] *which we can tell by looking at the node's children*

```
MAXHEAPIFY(A, i)
        l = LEFT(i)
        r = RIGHT(i)
        if l <=A.size and A[l] > A[i]
                largest = l
        else largest = i
        if r <= A.size and A[r] > A[largest]
                largest = r
        if largest != i
                swap A[i], A[largest]
                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

$$T(n) =\leq T(\frac{2n}{3}) + \Theta(1) \tag{13.42}$$

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.

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

Subsection 13.9

Placeholder for Quick, counting, selection, and radix sort

I'm going to skip these since these are fairly trivial and easily found online or in the textbook. I also didn't go to class.

Subsection 13.10

BSTs

TODO: Notes on red-black trees, etc.

Subsection 13.11

Amortized Analysis

Definition 53

- **Average cost**: mean across all inputs

$$\frac{1}{T-0}\int_0^T c(x)dx \tag{13.43}$$

- **Expected cost**: expectation over all inputs w.r.t a probability distribution

$$\int_0^T p(x)c(d)dx \tag{13.44}$$

- **Amortized Cost**: average cost over a *sequence* of operations

$$\frac{1}{n}\sum_{i=1}^{n} c(\frac{1}{n}) \tag{13.45}$$

The aggregate analysis method is simple and is as follows

1. Given a function or operation $f(x)$ and a sequence $\{x_1, x_2, \dots x_n\}$
2. Determine the total cost of the sequence $\sum_{i=1}^{n} c(x_i) \equiv T(n)$
3. The amortized cost is $\frac{T(n)}{n}$

The complicated but more powerful way to deal with this is the accounting method.

1. Declare a cost $\hat{c}$ for each operation/function call
2. Describe a procedure for how the cost will be allocated
3. Assert a credit invariant, i.e. that the credit in an object must be greater than or equal to 0
4. Prove the credit invariant
5. Using the credit invariant argue why the credit never goes negative
6. The amortized cost is $O(\hat{c})$

Let's consider a contrived example.

Example

Consider a data structure that has a cost of

$$c(x) = \begin{cases} x & \text{if x is a power of 2} \\ 1 & \text{otherwise} \end{cases} \tag{13.46}$$

Determine the amortized cost over the sequence $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
The aggregate method for solving this problem is trivial; simply compute the summation to find

$$\sum_{x=1}^{n} c(x) = \dots (2n-1) \leq 3n \tag{13.47}$$

What's more interesting is how we may apply the accounting method

1. Let cost $\hat{c} = 3$
2. If x is not a power of 2, uses 1 to pay for the operation and store the remaining 2
3. If x is a power of 2, store 2 in credit and use x from previous iterations to pay for the operation
4. (Steps 3, 4, 5) Prove the credit invariant. When $x = 2^m$ all elements $\in (2^{m-1}, 2^m]$ have 2 stored. The math can be a little handwavy but just draw out the ranges and show base cases etc.
5. And the amortized cost is $O(\hat{c}) = O(3) = O(1)$

What amortized cost is really telling us is that the cost of all operations as a whole is never larger than some value, though on the individual level it may fluctuate a little bit.

$$\frac{1}{n}\sum_{i=1}^{n}\hat{c} \geq \frac{1}{n}\sum_{i=1}^{n}c(x_i) \qquad \forall n \tag{13.48}$$

We can think of this is that the cumulative work stored in the data structure (amortized) being an upper bound on the actual cost.

$$\frac{1}{n}\sum_{i=1}^{n}(\hat{c} - c(x_i)) \geq 0 \qquad \forall n \tag{13.49}$$

The credit invariant is used to prove the above inequality.

If $c(x_i) \leq \hat{c}$ then we good and can store anything left over If $c(x_i) > \hat{c}$ then we need to use credit from previous iterations And the credit invariant is used to argue that we never use too much credit.

Example Consider a FIFO queue implemented using two stacks S_{in}, S_{out}. The stacks have the operations `Push(Q, x)` (pushes x onto S_{in}), `copy(Q)` (pops all elements from S_{in} and pushes onto S_{out}), and `pop(Q)` (calls copy and then pops off S_{out})

operation	actual cost	amortized cost
push	$O(1)$	$O(1)$
copy	$O(n)$	$O(1)$
pop	$O(n)$	$O(1)$

- Push: 1 pays for push onto S_{in}
- 2 pays for the eventual copy
- 1 pays for the pop from S_{out}

The credit invariant is therefore

1. Elements in S_{in} have 3 in stored credit
2. Elements in S_{out} have 1 in stored credit

The amortized cost of each operation is $O(1)$ for all

Example Consider a sequence of stack operations on a stack which never exceeds k elements. After k operations, we copy the stack

Proof

1. Let cost = 2. Push: pay 1, store 1. Pop: pay 1, store 1, Copy: pay k
2. credit invariant holds in stack because the number of credit stored on the stack equals the number of operations since the last copy. Since after each operation +1 is stored, then since the stack never has more than k elements after k operations we will have k stored and can therefore pay for the copy

□

Subsection 13.12

Skewed Heaps

Consider a *skewed heap* data structure which implements a `SkewHeapMerge` operation for merging heap-ordered trees by merging the rightmost paths of two trees without keeping any explicit balance conditions. Good performance of skewed heaps is guaranteed by a rebalancing step which swaps all the left and right children of nodes along the rightmost path (except for the last one). Since there is no balance condition the trees won't have $O(\log n)$ worst case performance but they do have good amortized performance.

SkewHeapMerge(h, h')

```
1
2  if h == NIL
3      return NIL
4  if h' == NIL
5      return NIL
6  if h.key ≤ h'.key
7      return SkewHeapMerge(h', h)
8  h.right = SkewHeapMerge(h.right, h')
9  SWAP(h.left, h.right)
10 return h
```

Example Show that a `SkewedMeapMerge`, `Insert`, and `DeleteMin` uses amortized $O(\log_2 n)$ time.

1. Allocate $\hat{c} = c\log(n)$ per operation (since we want to prove that it's $O(\log n)$ amortized)
2. Define $WT(x)$ as the number of nodes in the subtree rooted at x. Define a heavy node as a node with $WT(x) \geq WT(x.parent)$. Note that this doesn't include the root. A light node is exactly the opposite.
   - If all nodes on the rightmost path are light, then the cost of SkewHeapMerge is minimized
   - Define a misplaced node as a heavy node on the rightmost path

   Find the number of light nodes on any path from the root to the leaf.

Subsection 13.13

Minimum Spanning Trees

Definition 54 A minimum spanning tree is an acyclic subset of edges in a graph which connects all the vertices and has the minimum total weight[33]

[33] *i.e. sum of edge cost*

Two common algorithms for solving the minimum spanning tree problem are Kruskal's algorithm and Prim's algirthm. They are similar in that they both attempt to greedily add edges to the minimum spanning tree, but whereas Kruskal's algorithm grows a forest by adding the least weight edge in the graph that connects two distinct components, Prim's algorithm grows a single tree by adding the least-weight edge connecting the tree to a vertex outside fo the tree.

Definition 55 **Terminology**

- **Cut** $(S, V - S)$ is a partition of V[34]
- A cut **respects** a set A if no edge crosses the cut
- A **light edge** is the edge crossing a cut with the minimum weight
- A **safe edge** is any light edge crossing a cut $S, V - S$ of G that respects the MST A

[34] *see it as dividing V into two new sets $S, S - V$*

MST-Kruskal(G, w)
```
1  A = {0}
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4       sort G.E by weight w in nondecreasing order
5       for each edge (u, v) ∈ G.E
6            if FIND-SET(u) ≠ FIND-SET(v)
7                 A = A ∪ {(u, v)}
8                 UNION(u, v)
9  return A
```

In summary Kruskal's algorithm adds edges $(u, v)$to the MST in nondecreasing order of weight, *if* (u, v) do not belong to the same tree (as to prevent creating a cycle).

The runtime of these algorithms can be improved using a priority queue on the edge weights. However if the graph is dense ($E = V^2$) an array can be better an array can be better

MST-Prim(G, w, r)
```
1  for each vertex u ∈ G.V
2       u.key = ∞
3       u.π = NIL
4  Q = G.V
5  while Q ≠ {0}
6       u = EXTRACT-MIN(Q)
7       for each vertex v ∈ G.Adj[u]
8            u.key = w(u, v)
9            u.π = u
```

Prim's algorithm builds a MST set A that always forms a single tree. Each step adds a light edge to A that connects it to an isolated vertex. Since this only adds edges that are safe for A and A will eventually contain all vertices in V, A forms a minimum spanning tree.

Subsection 13.14
Shortest Paths

Common algorithms for solving the shortest path problem are Dijkstra's algorithm, Bellman-Ford's algorithm, and A*.

Dijkstra's algorithm and A* were discussed in ESC190 and such will not be discussed here. Bellman-Ford's algorithm, unlike Dijkstra, can handle negative weight edges[35]

[35] *though still fail to produce a shortest path as long as no negative-weight cycles are reachable, but can detect and report its existence.*

Definition 56

Relaxation
For each vertex $v \in V$, we maintain a *shortest path estimate* $v.d$ describing the upper bound on the weight of a shortest path from source to v, i.e. $s \rightsquigarrow v$. The initial state of $v.d$ will just be ∞

π denotes the predecessor of a vertex in a shortest path from s to v

initalize-single-source(G, s)
```
1  for each v ∈ G.V
2       v = d = ∞
3       v.π = NIL
4  s.d = 0
```

relaxing an edge (u, v) means to check if the shortest path to v so far can be improved by going through u, and updating $v.d, v.\pi$ if so.

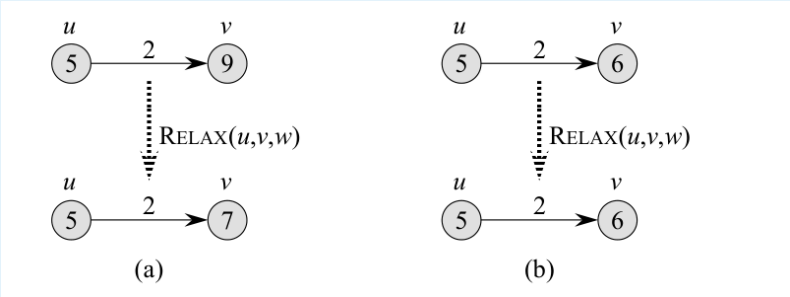


Figure 50. Relaxing an edge

RELAX(u, v, w)
1 **if** $v.d > u.d + w(u, v)$
2 $\quad v.d = u.d + w(u, v)$
3 $\quad v.\pi = u$

An issue with relaxation-based algorithms is that order matters; continuing to relax in a graph with negative weight cycles can cause an infinite loop, and a poor choice of order can lead to many unnecessary relaxations. Bellman-Ford proposes proposes to label edges E in a topological order $e_1 ... e_n$ and then relax them in that order $|V| - 1$ times.

All of the presented algorithms use only RELAX to change shortest-path estimates and predecessors. I'll also include the Dijkstra algo using RELAX later as well.

Definition 57

Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
1 INITALIZE-SINGLE-SOURCE(G,s)
2 **for** $1 \leq i \leq |G.V| - 1$
3 $\quad$ **for** each edge $(u, v) \in G.E$
4 $\quad\quad$ RELAX(u,v,w)
5 **for** each edge $(u, v) \in G.E$
6 $\quad$ **if** $v.d > u.d + w(u, v)$
7 $\quad\quad$ **return FALSE**
8 **return TRUE**

The algorithm makes $|V| - 1$ passes over the graph, attempting to relax each edge $|V| - 1$ times. Briefly we can see that the algorithm is correct by observing that the longest acyclic path in a graph with $|V|$ vertices has length $|V| - 1$. And the algorithm maintains the invariant that the i-th pass $v.d$ is at most the weight of every path from $s \rightsquigarrow v$ using at most i edges. So by the $|V| - 1$-th pass each edge is maximally relaxed, i.e. $v.d = \delta(s, v)$ for all $v \in V$, where $\delta(u, v)$ denotes the actual shortest path weight

It then checks for a negative weight cycle reachable from the source, the proof for how this works can be found in CLRS Pg. 653[36]

In scenarios where we are working with DAGs we can use a topological sort to order the edges and relax them in that order – which would result in a linear time algorithm $\Theta(V + E)$. There are no negative-weight cycles possible in a DAG so shortest paths will always be well-defined

[36] *But more or less the approach is to build a contradiction around how the sum of edge weights in a negative cycle is* ≤ 0 *and the conditions on* $v.d$ *being equal to* $u.d + w(u, v)$

DAG-BELLMAN-FORD(G, w, s)

```
1  sort G.V topologically
2  INITALIZE-SINGLE-SOURCE(G,s)
3  for each vertex u ∈ G.V taken in topological order
4      for each vertex v ∈ G.Adj[u]
5          RELAX(u,v,w)
```

Recall: topological ordering of DAG is a linear ordering of its vertices such that if $u \rightsquigarrow v$ then u comes before v in the ordering, i.e. a graph traversal such that each node v is visited only after all its dependencies are visited. An algorithm for topologically sorting a graph is just DFS with a colouring concept; WHITE for unvisited, GREY for visiting, and BLACK for visited. Time is simply a shared variable that increments for each call to DFS. The starting time of a vertex is the time when it is first discovered, and the finishing time is the time when it is colored black. The topological ordering is the reverse of the finishing times.

TOPOLOGICAL-SORT(G)

```
1  call DFS(G) to compute finishing times f[v] for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

Dijkstra's algorithm is then just a special case of the Bellman-Ford algorithm for a graph with no negative edge weights that relaxes greedily edges radially outwards from the source.

DIJKSTRA(G, w, s)

```
1  INITALIZE-SINGLE-SOURCE(G,s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q) (min on v.d)
6      S = S ∪ {u}
7      for each v ∈ G.Adj[u]
8          RELAX(u,v,w)
```

Subsection 13.15

Splay Trees

Consider the weight dictionary problem where we want to maintain a set of keys $K = \{k_1, k_2 \dots k_n\}$ with associated values $V = \{v_1, v_2 \dots v_n\}$ and access frequencies $W = \{w_1, w_2, \dots w_n\}$ and we want to access high-access-frequency item faster than low-frequency items without sacrificing efficiency on INSERT, SEARCH, and DELETE. If no insert/delete operations are needed across the tree during runtime one can build the static optimal search tree i.e. the one that minimizes $P = \sum_i w_i d_i$ where w_i is the weight of a path and d_i be the depth of node x_i ahead of time in $O(n^3)$ with DP. To motivate the study of splay trees, let's think of the case where we would want to remove from the BST and have different access frequencies during runtime; a more challenging problem. A data structure that achieves an amortized time within a constant multiple of the theoretic lower bound is the **splay tree**; a self-organizing data structure, or one that moves x closer to the entry point (in a BST the root) whenever x is accessed.

Definition 58 A splay tree is an ordered binary tree that satisfies

- every element in the left subtree of x is $\leq x$
- every element in the right subtree of x is $\geq x$

Across which we can perform the splay procedure

```
    splay(x)
        while x is not the root do
            let p be the parent of x
            if p is the root then
(1)             rotate at p and stop
                    — now x is at the root
            else if x and p are both right children or left children then
(2)             rotate at parent(p)
(2)             rotate at p
                    — now x is where its grandparent was
            else { x is a left child and p a right child, or vice versa }
(3)             rotate at p
(3)             rotate at the new parent of x (its former grandparent)
                    — now x is where its grandparent was
```

splay is a procedure that moves x to the root of the tree by rotating x and it's parent. There are a maximum of $\frac{h}{2}$ steps or h rotations if the height of node x is h (Each step in splay is actually two rotations)
The other BST operations are therefore implemented as follows

- locate(x): search for x in the tree and then call splay on x
- insert(x): insert x into the tree as a leaf and then call splay on the newly inserted x
- delete(x) : Delete an element as usual. Leaf nodes can simply be removed; internal nodes are replaced by their in-order successor and then the successor is removed. Then call splay on the parent of the deleted node.

Note that there is no explicit balance condition and as such the worst-case complexity can be $O(n)$ but the *amortized* complexity of each operation is $O(log(n))$

This splay operation causes items that are accessed most frequently to move closer to the root.

Definitions:

Proof Prove that splay has an amortized cost of $O(log(n))$.

- Let $\text{WT}(x)$ be the weight of node x, i.e the number of nodes in the subtree rooted at x including x itself.
- Let $\text{rank}(x) = \log \text{WT}(x)$

Credit invariant: assert that there are always $\text{rank}(x)$ credits on x, $\forall x \in$ tree. 1 credit will be charged in each rotate operation and only $O(\log n)$ credits are needed in addition to maintain this invarient.
Claim: for each step in the splay operation the number of additional credits is at most $3(\text{new rank}(x) - \text{old rank}(x))$ and maybe one more for the last step; any other credits can be taken from the tree. Consider the rank of x as the splay operation proceeds; $\text{rank}_0, \text{rank}_1, \text{rank}_2, ...\text{rank}_k$. Therefore the total number of credits needed is

$$1 + 3(\text{rank}_k - \text{rank}_{k-1} \dots 3(\text{rank}_1 - \text{rank}_0) \tag{13.50}$$

Cancelling out terms,

$$= 1 + 3(\text{rank}_k - \text{rank}_0) \tag{13.51}$$

Since $\text{rank}_k = \log \text{WT}(x)$,

$$1 + 3(\text{RANK}_k - \text{RANK}_0) \leq 1 + 3\log n \tag{13.52}$$

Now let's analyze it with respect to the three rotation cases in SPLAY;

1. if x's parent is the root
2. if x and its parent are both right or left children
3. or if x is a left child and its parent is a right child or vice versa

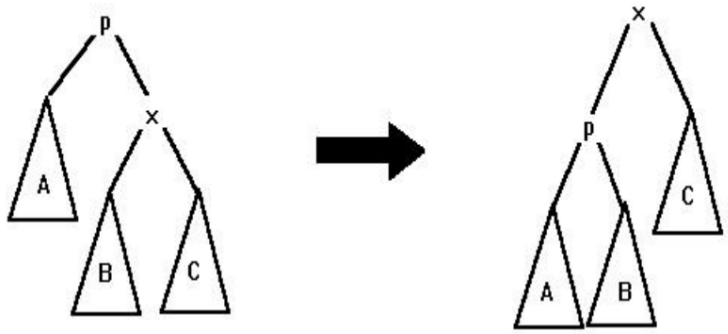


Figure 51. Case 1: x's parent is the root

This becomes a case of just counting the new and old ranks of x and its parent p

$$\text{OLD RANK}(x) = \log(1 + |B| + |C|) \leq \text{NEW RANK}(x) = \log(2 + |B| + |C| + |A|) \tag{13.53}$$

$$\text{OLD RANK}(p) = \log(2 + |A| + |B| + |C|) \geq \text{NEW RANK}(p) = \log(1 + |A| + |C|) \tag{13.54}$$

p can only decrease in rank. x already has $\text{OLD RANK}(x)$ credits, so we only need $\text{NEW RANK}(x) - \text{OLD RANK}(x)$, which is only 1/3 of the allocated credits. One more credit may be used to pay for the rotation, which can be taken from the +1 taken for the last step.

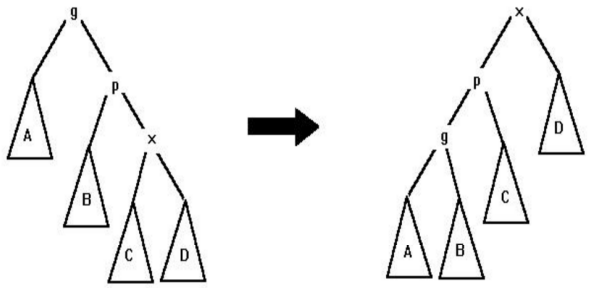


Figure 52. Case 2: x, p both left or right children

We have

$$\text{OLD RANK}(x) + \text{OLD RANK}(p) + \text{OLD RANK}(g) \tag{13.55}$$

Available and we need

$$\text{NEW RANK}(x) + \text{NEW RANK}(p) + \text{NEW RANK}(g) \tag{13.56}$$

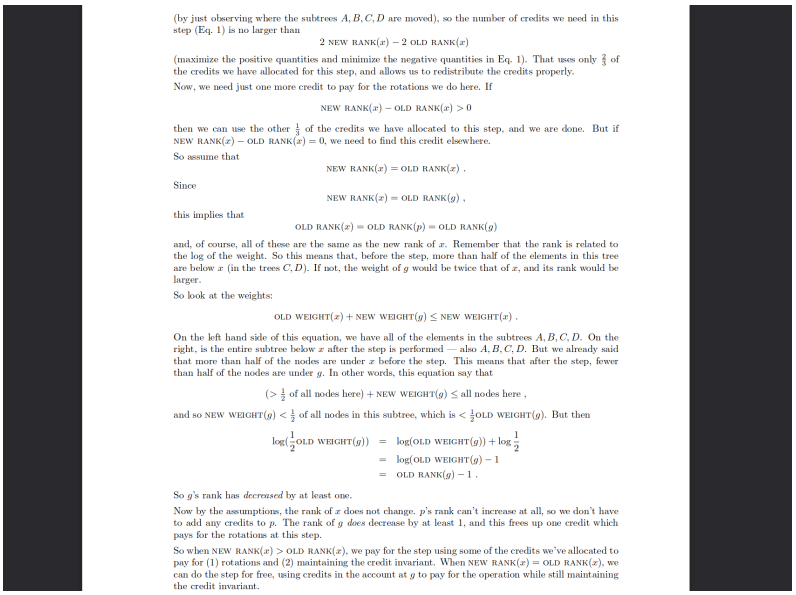
Some math can be done to find that this is still bounded by our claim, i.e. $1 + 3(\text{NEW RANK}(x) - \text{OLD RANK}(x))$
Here are the screenshotted notes (since I don't want to write it out)

Since NEW RANK(x) = OLD RANK(g)

$$\text{NEW RANK}(p) + \text{NEW RANK}(g) - \text{OLD RANK}(x) - \text{OLD RANK}(p) \tag{1}$$

credits to what we have. But we note that

$$\begin{aligned} \text{NEW RANK}(p) &\leq \text{NEW RANK}(x) \\ \text{NEW RANK}(g) &\leq \text{NEW RANK}(x) \\ \text{OLD RANK}(p) &\geq \text{OLD RANK}(x) \end{aligned}$$



(by just observing where the subtrees A, B, C, D are moved), so the number of credits we need in this step (Eq. 1) is no larger than

$$2\ \text{NEW RANK}(x) - 2\ \text{OLD RANK}(x)$$

(maximize the positive quantities and minimize the negative quantities in Eq. 1). That uses only $\frac{2}{3}$ of the credits we have allocated for this step, and allows us to redistribute the credits properly.

Now, we need just one more credit to pay for the rotations we do here. If

$$\text{NEW RANK}(x) - \text{OLD RANK}(x) > 0$$

then we can use the other $\frac{1}{3}$ of the credits we have allocated to this step, and we are done. But if NEW RANK(x) − OLD RANK$(x) = 0$, we need to find this credit elsewhere.

So assume that

$$\text{NEW RANK}(x) = \text{OLD RANK}(x)\ .$$

Since

$$\text{NEW RANK}(x) = \text{OLD RANK}(g)\ ,$$

this implies that

$$\text{OLD RANK}(x) = \text{OLD RANK}(p) = \text{OLD RANK}(g)$$

and, of course, all of these are the same as the new rank of x. Remember that the rank is related to the log of the weight. So this means that, before the step, more than half of the elements in this tree are below x (in the trees C, D). If not, the weight of g would be twice that of x, and its rank would be larger.

So look at the weights:

$$\text{OLD WEIGHT}(x) + \text{NEW WEIGHT}(g) \leq \text{NEW WEIGHT}(x)\ .$$

On the left hand side of this equation, we have all of the elements in the subtrees A, B, C, D. On the right, is the entire subtree below x after the step is performed — also A, B, C, D. But we already said that more than half of the nodes are under x before the step. This means that after the step, fewer than half of the nodes are under g. In other words, this equation say that

$$(> \tfrac{1}{2} \text{ of all nodes here}) + \text{NEW WEIGHT}(g) \leq \text{all nodes here}\ ,$$

and so NEW WEIGHT$(g) < \frac{1}{2}$ of all nodes in this subtree, which is $< \frac{1}{2}$OLD WEIGHT(g). But then

$$\begin{aligned} \log(\tfrac{1}{2}\text{OLD WEIGHT}(g)) &= \log(\text{OLD WEIGHT}(g)) + \log\tfrac{1}{2} \\ &= \log(\text{OLD WEIGHT}(g) - 1 \\ &= \text{OLD RANK}(g) - 1\ . \end{aligned}$$

So g's rank has *decreased* by at least one.

Now by the assumptions, the rank of x does not change. p's rank can't increase at all, so we don't have to add any credits to p. The rank of g *does* decrease by at least 1, and this frees up one credit which pays for the rotations at this step.

So when NEW RANK$(x) >$ OLD RANK(x), we pay for the step using some of the credits we've allocated to pay for (1) rotations and (2) maintaining the credit invariant. When NEW RANK$(x) =$ OLD RANK(x), we can do the step for free, using credits in the account at g to pay for the operation while still maintaining the credit invariant.

A very similar proof follows for case 3.

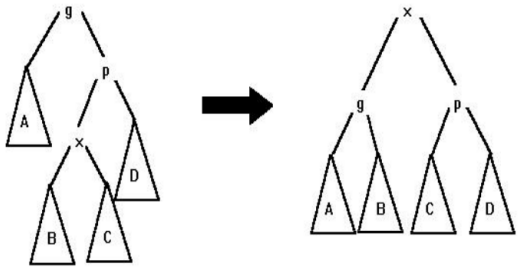


Figure 4: Case 3 of the SPLAY procedure

Figure 53. Case 3

One edge case to consider is when the new rank of x is the same of the old rank x. This can be solved by considering the ranks of g and p

Suppose that

$$\text{OLD RANK}(x) = \text{NEW RANK}(x),$$

so that we have 0 credits to do this step. It is still the case that this implies

$$\text{OLD RANK}(x) = \text{OLD RANK}(p) = \text{OLD RANK}(g)\ ,$$

so more than half of the nodes in this subtree are originally under x. But after the rotations, we see that some of the nodes under x are now under g (B) and some are under p (C). There are two possibilities.

Either: One of the nodes g and p now has $> \frac{1}{2}$ of the weight, which means that the other has $< \frac{1}{2}$ the weight and therefore drops in rank. That frees up one credit.

Or: If g and p have about the same weight, then *both* of them have dropped in rank (since each node has lost half of its weight), and we still get at least one credit freed. So, in this case too, we can take the cost of the rotations from the credits already on the tree.

Think of it this way. Let's say that the entire subtree has a bit more than 2^k nodes, and x, p, g originally had rank k. That means that most of these 2^k nodes are under x. Now after the rotation, we split these nodes between g and p. Either one of them has about 2^k nodes — which implies that the other has much less (and so a rank lower than k) — or both of them have about 2^{k-1} nodes — so both of them have a lower rank.

To be sure that you understand the proof, you should try filling in the rest of the details of this case.

Therefore the cost of a splay operation is at most $1 + 3 \log n$ credits, giving it an amortized cost of $O(\log n)$.

□

Proof The time complexities of $\text{LOCATE}(x)$, $\text{INSERT}(x)$, and $\text{DELETE}(x)$ follow from the previous proof and are all $O(log(n))$ □

As a general comment we use credit immediately when the tree is almost balanced; paying for single rotations or when the rotations make it more unbalanced. In other cases the rotation will use credits to make the tree more balanced.

Subsection 13.16

Maximum Flow

We can interpret a directed graph as a flow network and use it to answer questions about, well, flow. For example an early application of this problem was to solve an issue the US army was experience: given a train network, what is the minimum set of routes to be bombed to cause maximum disruption. More formally, one can imagine a directed graph[37] with vertices representing conduit junctions and each edge having a certain capacity. The maximum flow problem seeks to find the greatest rate at which material can be moved from a source to a sink through a network without violating the capacity constraint of any edge.

[37] *representing some sort of flow system i.e. pipes in a sewer system or current through a wire*

Definition 59

Flow network

A flow network $G = (V, E)$ is a directed graph where each edge (u, v) has a capacity $c(u, v) > 0$. For the time being we will consider only graphs without self-loops and no reverse edges.

The source s and sink t are distinguished vertices in V such that there is a path between to two, i.e. $s \to t$. Also assume that for each vertex $v \in V$ there exists a path from s to v and a path from v to t; $s \to v \to t$

The **flow** in G is then more formally a function

$$f : V \times V \to \mathbb{R} \tag{13.57}$$

That satisfies the capacity

$$0 \leq f(u, v) \leq c(u, v) \qquad \forall (u, v) \in V \tag{13.58}$$

And flow conservation constraints

$$\sum^{v \in V} f(v,u) = \sum^{n=v \in V} f(u,v) \qquad \forall u \in V - \{s,t\} \tag{13.59}$$

$f(u,v)$ is the flow from vertex u to vertex v, and the value of f is defined as the difference between the sum of flow into the source and the sum of flow out of the source.

$$|f| = \sum^{v \in V} f(s,v) - \sum^{v \in V} f(v,s) \tag{13.60}$$

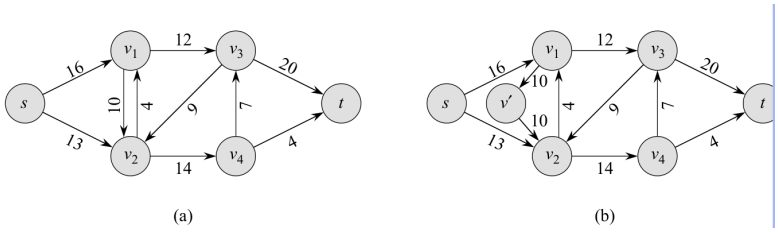


Figure 54. A network with antiparallel (reverse) edges ($v_1 \to v_2$)

Antiparallel edges can be handled by building an equivalent network with no antiparallel edges (See: 54) via splitting an antiparallel edge into a new vertex and two edges.

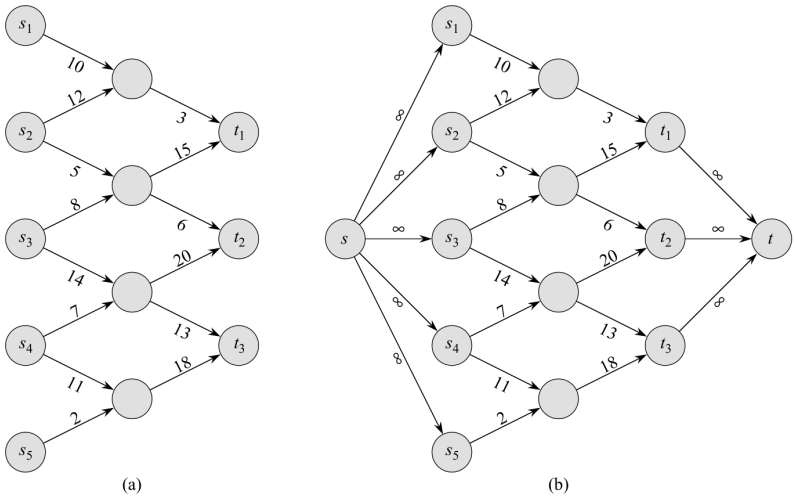


Figure 55. Multiple sources & sinks

Multiple sources and sinks can be modelled by attaching a **supersource** and **supersink** with infinite capacity between each of the existing sources and sinks (55).

One method for solving the maximum flow problem is the **Ford-Fulkerson** method[38]

[38] *It actually encompasses a number of implementations with different runtimes*

Definition 60

FORD-FULKERSON-METHOD(G, s, t)
1 $f = 0$
2 **while** $\exists$ augmenting path $p \in$ residual network G_f
3 augment flow f along b
4 **return** f

Definition 61

Residual networks G_f are graphs that represents the how much we can still change the flow on the edges of G, i.e $c_f(u,v) = c(u,v) - f(u,v)$. If this capacity is negative the edge is omitted from the residual network. Note that it may be sometimes necessary to send back flow along an edge (or decreasing the flow along an edge). The maximum amount of backflow that can be supported along the edge (u,v) is the current flow along the edge. This can be accomplished by adding a new edge in the opposite direction (v,u) with capacity equal to $f(u,v)$
So, more formally,

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise} \end{cases} \tag{13.61}$$

The **augmentation** of flow f by f', $f \uparrow f'$ is given by

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases} \tag{13.62}$$

Lemma 3 $f \uparrow f'$ is a flow in G with value $|f + f'| = |f| + |f'|$

Lemma 4 Flow is conserved, i.e.

$$(f \uparrow f')(u,v) = f(u,v) + f'(u,v) - f'(v,u) \leq f(u,v) + f'(u,v) \leq f(u,v) + c_f(u,v) = c(u,v) \tag{13.63}$$

$$\sum^{v \in V} (f \uparrow f')(u,v) = \sum^{v \in V} (f \uparrow f')(v,u) \tag{13.64}$$

Definition 62 An **augmenting path** p is a simple path from s to t in the residual network G_f.

Definition 63 The **Edmonds-Karp** algorithm is an instance of the Ford-Fulkerson method that chooses the shortest (in terms of edge count) augmenting path using BFS and can be show to terminate after $O(VE)$ augmentations and has $O(VE^2)$ runtime (since BFS is $O(V+E)$)
Suppose p is a path.

$$C_p(p) = min\{c_f(u,v) : (u,v) \text{ on path } p\} \tag{13.65}$$

Section 14

Theory of Computation

Subsection 14.1

Finite Automata

Definition 64 **Deterministic Finite Automata (DFAs)**[39]
Given a sequence of inputs they can accept (true) or reject (false).
More technically, a DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

- A set of states Q
- An input alphabet Σ (domain of input)
- Transition function $\delta : Q \times \Sigma \to Q$ which takes in a state and some input $\in \Sigma$ to transition to some other state
- An initial state $q_o \in Q$
- A set of accepting states $F \subseteq Q$

[39] *Glorified if-statements*

Example Create a DFA that accepts strings of the form `"abc"`

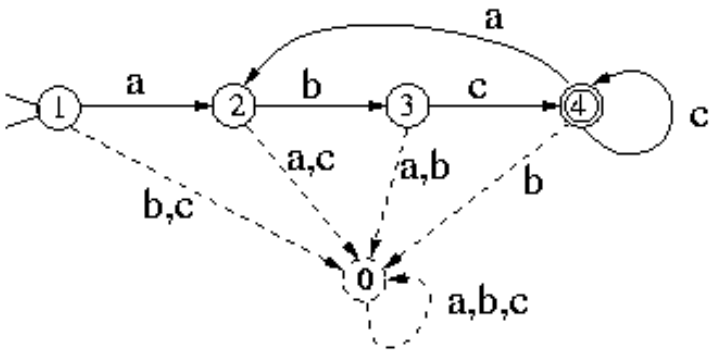


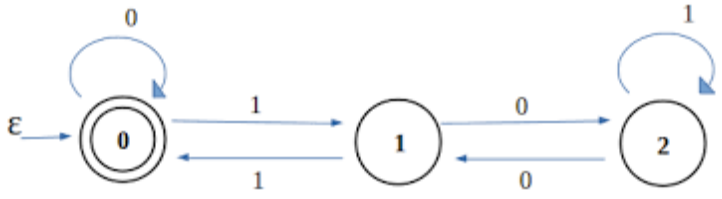
Figure 56. Note non-accepting state 0 w/ loop. Accepting state q_3 indicated via double circle

Example Create a DFA that accepts all inputs with an even number of 0s. $\Sigma = \{0\}$

$$q_0 \overset{0}{\leftrightarrow} q_1 \tag{14.1}$$

where q_0 is the accepting state

Example Create a DFA on binary strings that only accepts on strings with a number of 0s that is a multiple of 3



Definition 65 **Non-deterministic Finite Automata (NFA)**
These are the same as DFAs except we can take an additional input ε which is the empty string. On ε the NFA branches and computes each branch in parallel. If *any* branch reaches

an accepting state the NFA accepts.

Example Create an NFA that accepts on any input with either a multiple of 2 or 3 0s. This is simply the two DFAs we found earlier connected via a branch on q_0.

Example Find a NFA that accepts on either an even number of 0s or exactly two 1s

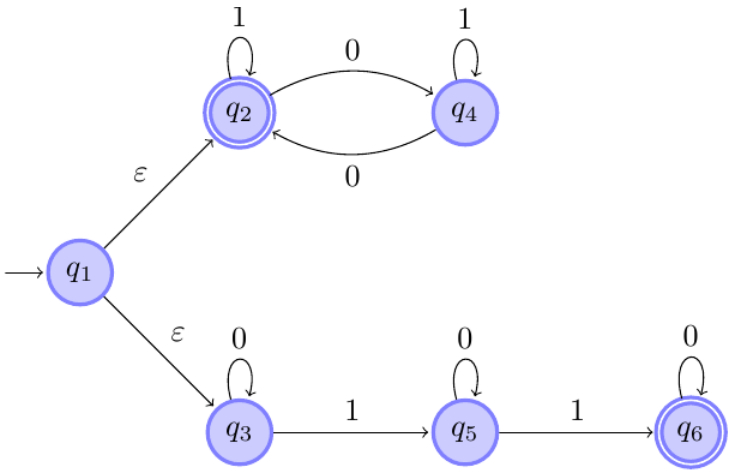


Figure 57. Note how this is just two DFAs connected together

Theorem 11 All NFAs can be converted to an equivalent DFA[40]

[40] *NFA doesn't provide any additional computability*

DFAs and NFAs are memoryless, for example a DFA or NFA is incapable of accepting on a string e.x

$$O^n I^n \qquad for \qquad n \geq 0 \tag{14.2}$$

sequences like `000111` for n = 3, etc

Subsection 14.2

Turning Machines

In order to address this Turing machines were introduced as basically a DFA with infinite memory.

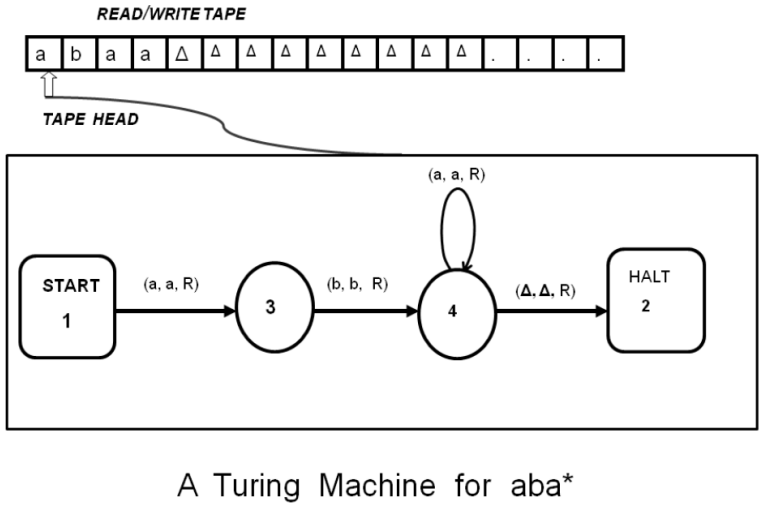


Figure 58. A Turing machine for aba

And Turing machines can compute anything[41]

[41] *except for some things like the halting problem etc. TLDR if a problem is not computable by turning machine it is not computable by any finite means. E.g. printing all real numbers between 0 to 1; this is an uncountable infinity but the turning machine tape is countable; larger infinity*

Definition 66 **Nondeterministic Turing Machine (NTM)**
Or: a turing machine with a NFA head. Like how a DFA can be converted into a NFA (which would be faster because of parallel computation), a NTM has much better complexity compared to a regular TM.

Subsection 14.3
Complexity Classes

Definition 67 **Complexity Classes**

- P: **P**olynomial number of steps to compute on a TM
- NP: Non-deterministic Polynomial: takes a polynomial number of steps to compute on a NTM.

Theorem 12 All problems in NP can be verified on polynomial time on a TM, since verifying is like going backwards; since when computing one has to go through all branches on a NTM but when verifying we only have to take one path.

Subsection 14.4
NP Completeness

Definition 68 **Reductions**

$$A \leq_p B \tag{14.3}$$

(14.3) means that A is *polynomially reducible* to B. There are about seven equivalent ways

to interpret this.

1. There exists a polynomial time function f such that $\alpha \in A \Leftrightarrow f(\alpha) \in B$
   - The contrapositive can also apply; $f(\alpha) \notin B \Rightarrow \alpha \notin A$
   - If we can solve it in B-space then we can solve it in A-space
2. We can decide $B \Leftrightarrow$ we can decide A, i.e. that A
3. Runtime of $A \leq$ runtime $B + O(n^k)$ (some polynomial factor)

Hard: Takes a long time to compute

Definition 69 **NP-Complete**

A decision problem L s.t. $L \in NP$ such that

1. $L \in NP$
2. $L' \leq_p L$ for all $L' \in NP$ (NP-hard)

Informally, a problem is NP-Complete if it is at least as hard as any other problem in NP

Suppose $A \leq_p B \Rightarrow A \leq B + O(n^k)$

$$\begin{aligned} &\text{if } B \in P \Rightarrow A \in P \\ &\text{if } A \in P \Rightarrow B \in ? \\ &\text{if } B \in NPC \Rightarrow A \in ? \\ &\text{if } A \in NPC \Rightarrow L' \leq_p A \leq_p B \Rightarrow B \in NP\text{-hard} \end{aligned} \tag{14.4}$$

14.4.1 Solving NPC problems

Given some decision problem A

1. Showing $A \in NP$
   - Provide a *certificate* of $A \in NP$: the evidence that the problem is an instance of A
   - Provide a *verification procedure* to check of the certificate is really an instance of A
   - Argue why the verification procedure is polynomial time
2. Showing $A \in NP$-hard
   - Determine which NPC problem L' we will use (usually provided on exam)
   - Show that $L' \leq_p A$; transform a known NPC problem to an unknown (i.e. NP-hard)

Showing that A is NP-Complete requires proving both of the above.

Example

- Ham-Cycle: Given $G = (V, E)$, find a simple *cycle* that travels through all vertices.
- Ham-Path: Given $G = (V, E)$, find a simple *path* that travels through all vertices.

1. Show that Ham-Cycle is in NP
   - **Certificate**: a cycle $C = \{v_0, v_1 \dots v_n\}$
   - **Verification**:
     - Check that all of the edges in the cycle are in the graph; $\forall i, (v_i, v_{i+1}) \in E$. This is $O(VE)$
     - Check that all vertices appear exactly once $O(V)$
     - This verification procedure is clearly of polynomial time

Proof can be fairly hand wavy for these simple complexities

2. Show that Ham-Cycle $\in$ NP-hard via Ham-Path

$$\text{Ham-Path} \leq_p \text{Ham-Cycle} \tag{14.5}$$

We note that we can transform the known NPC problem (that we proved in the previous part of this example) into an unknown by adding an additional vertex x to G to form G'. Then, for each vertex v of G, add an edge to (v, x). This will guarantee forming a ham-cycle. Construct $G' = (V', E')$, $V' = V \cup x$

$$E' = E \cup (v, x), (x, v) | v \in V \tag{14.6}$$

Claim: $< G > \in$ Ham-Path $\Leftrightarrow < G' > \in$ Ham-Cycle

- $\Rightarrow$ proof: Suppose G is an instance of Ham-Path ($\exists V = v_o \rightarrow v_1 \rightarrow v_n \in G$), therefore $x \rightarrow v_0 \rightarrow v_1 \rightarrow v_n \rightarrow x$ is a ham-cycle in G. So G' is an instance of a ham-cycle
- $\Leftarrow$ proof: Suppose G' is an instance of Ham-Cycle, then $\exists v_0 \rightarrow v_1 \rightarrow v_n \rightarrow x \rightarrow v_0$; a known cycle in G'
- We can remove x to form a ham-path in G
- And therefore G is an instance of ham-path

Definition 70

Combinatorial Equivalence Checking
Given boolean formulas $A(x_1, x_2, x_3 \dots x_n)$ and $B(x_1, x_2, x_3 \dots x_n)$, does there exist an assignment such that $A \neq B$?
`Formula-SAT`: Given a boolean formula $F(y_1, \dots y_k)$, is there an assignment such that $F = 1$?

Example

1) $CEC \in NP$.

- Certificate: assignment $x_1, \dots x_n$
- Verification: Evaluate A, B and check if $A \neq B$

2) $CEC \in NP$-hard via Formula-SAT

$$\text{Formula-SAT} \leq_p \text{CEC} \tag{14.7}$$

Given a boolean formula F, construct $A = F$ and $B = 0$. Then, makke a claim that $< F > \in Formula - SAT \Leftrightarrow < A, B > \in CEC$.

- $\Rightarrow$ proof: there exists an

Example

Prove that Half-3-CNF-SAT is NP-Complete. Given a 3-CNF formula Φ with n variables and m clauses, does there exists a truth assignemnt to Φ that causes exactly $m/2$ of the clauses to evaluate to true and exactly $\frac{m}{2}$ of the clauses to evaluate to false. Now that Φ can contain duplicate clauses and tautological classes like $x \vee \neg x \vee y$
1: Show that Half-3-CNF-SAT is in NP
Certificate: an assignment of true, false sequence to Φ. Verification: evaluate all clauses and make sure exactly half are 1 and the other half is 0. This is $O(m)$
2: Show that Half-3-CNF-SAT is NP-hard via 3-CNF-SAT

$$\text{3-CNF-SAT} \leq_p \text{Half-3-CNF-SAT} \tag{14.8}$$

Attempt: construct $\Phi' = \Phi \wedge (p \vee q \vee r)^m$. This is incorrect.
Attempt: construct

$$\Phi' = \Phi \wedge (p \vee q \vee r)^m \wedge (p \vee q \vee r)^{2m} \tag{14.9}$$

Claim:

$$< \Phi > \in 3 - SAT \Leftrightarrow < \Phi' > \in Half - 3 - CNF - SAT \tag{14.10}$$

Example Graph-3-colourability: Given $G = (V, E)$, can you assign one of three colours to each vertex such that no two adjacent vertices have the same colour?
k-clique-cover: Given $G = (V, E)$ and k, can we partition the vertices v into $v_1, \dots v_k$ such that each v_i is a clique. This is known to be NP-Hard
1) Certificate: an assignment of colours to edge vertices. Verification: iterate over each edge and check that the vertices on each end of the vertex are not of the same colour.
2) 3-col is NP-hard via k-clique-cover. In particular we will do this via 3-clique-cover

$$3 - clique - cover \leq_p 3 - col \tag{14.11}$$

Given $G = (V, E)$ is an instance of 3-clique-cover, construct $G' = (V', E')$ is the complement of G - $V' = V$, $E' = \{(u, v) | (u, v) \notin E\}$. [42]
Claim:

[42] *Note that this is in polynomial time. Remember to put this on the solution for exams!*

$$< G > \in 3 - clique - cover \Leftrightarrow < G' > \in 3 - col \tag{14.12}$$

Forwards: suppose G has a 3-clique-cover. So then we color each set v_i a different color. Then $u.color = v.color \Rightarrow (u, v,) \in E \Rightarrow (u, v) \notin E'$. So G is 3-colourable
Backwards: suppose G' is 3-colourable. $\exists$ coloring such that $u.color = v.color \Rightarrow (u, v,) \notin E' \Rightarrow (u, v) \in E$. Therefore let all nodes of the same color be a clique; G.

Example Partition: Let P be a set of positive integers. P is an instance of Partition if

$$\exists P_1, P_2 \subseteq P \text{ such that } P_1 \cup P_2 = P \text{ and } ¶_1 \cap P_2 = \{\} \text{ and } \sum_{x \in P_1} x = \sum_{x \in P_2} x. \tag{14.13}$$

Subset-sum: Given a set of positive integers X and positive integer t, does there exist $S' \subseteq S$ s.t. $\sum S' = t$
Showing that this is NP is fairly trivial so we'll skip it. Showing that it is NP-hard is trickier. We will take the approach of showing that Partition is NP-Hard via Subset-sum

$$\text{Subset-sum} \leq_p \text{Partition} \tag{14.14}$$

Given set S and target t, construct $x = \sum S$, $P = S \cup \{x - 2t\}$.
Claim:

$$< s, t > \in subset - sum \Leftrightarrow < P > \in Partition \tag{14.15}$$

Forwards: Suppose $\exists T_1 \subseteq A$ st. $\sum T_1 = t$. Let

PART

V

ECE360: Electronics

Taught by Prof. Khoman Phang

Section 15

Admin and Preliminary

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)[43].

operation	actual cost	amortized cost
push	$O(1)$	$O(1)$
pop	$O(1)$	$O(1)$
copy	$O(n)$	$O(1)$

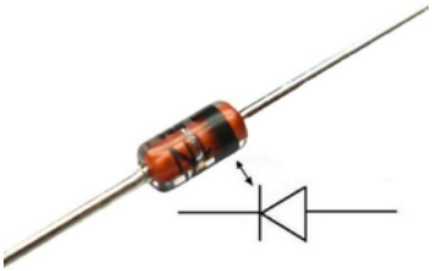
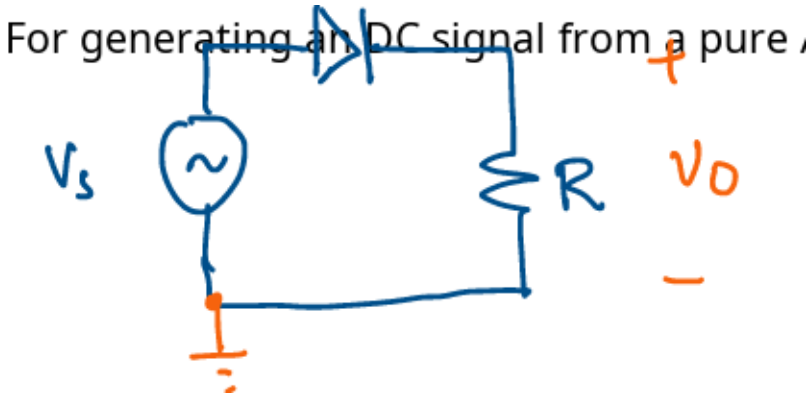
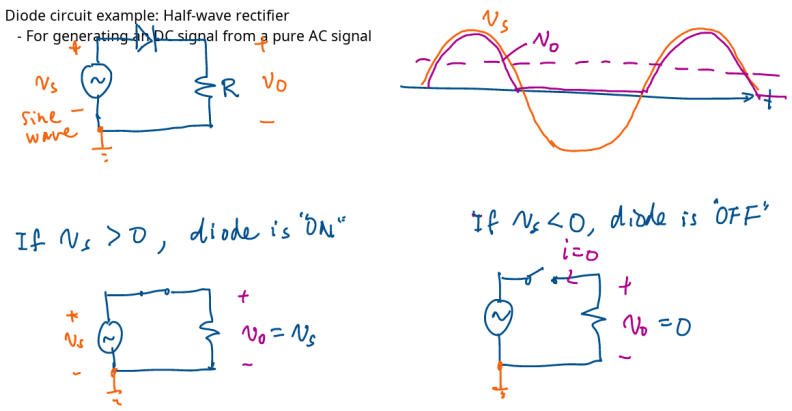


Figure 59. A diode and its symbol

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





Section 16

Diodes

Subsection 16.1

Lecture 2

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

- Off ↔ reverse bias
- On ↔ forwards bias

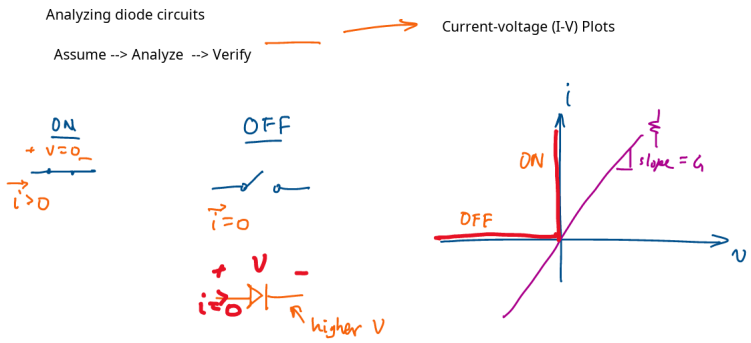


Figure 60. General steps for analyzing non-linear circuits. Note plotting out expected response

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

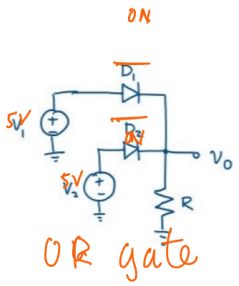
Analysis Examples

'0' '1'

Example 1:

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

What is the function of this circuit?

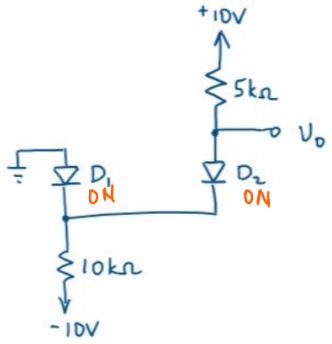


V_1	D_1	V_2	D_2	V_o
5V	ON	5V	ON	5V
5	ON	0	OFF	5V
0	OFF	5	ON	5V
0	OFF	0	OFF	0V

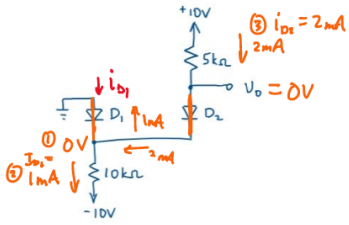
Example 2:

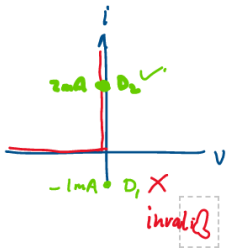
Find output voltage Vo

Assume --> Analyze --> Verify

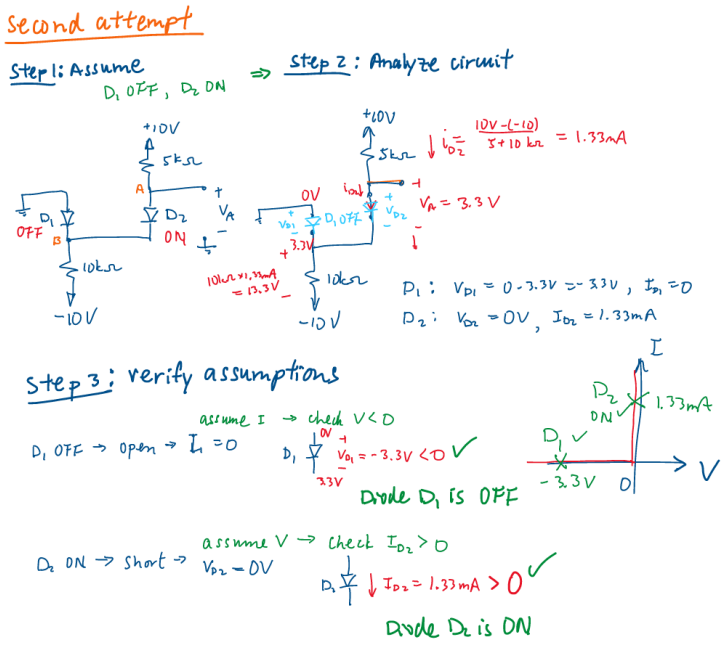


Assume D_1 ON
D_2 ON





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



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.

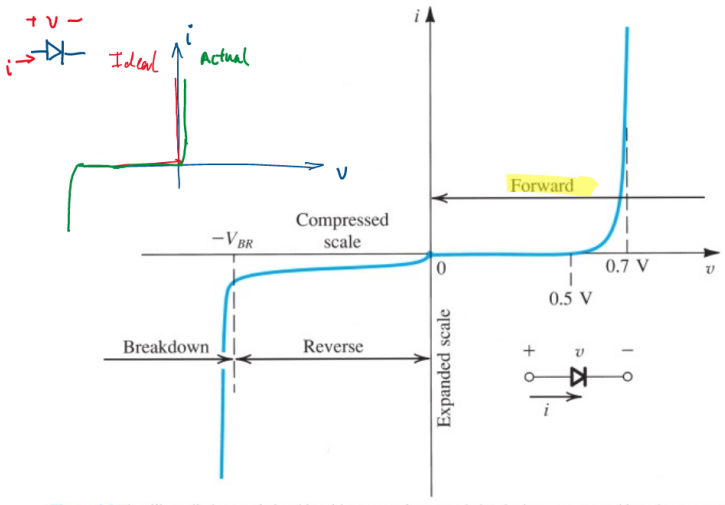


Figure 4.8 The silicon diode i–v relationship with some scales expanded and others compressed in order to reveal details.

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 13 **Forward Bias**

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

Where:

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

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.

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 $10^{-9}to10^{-15}A$ and scales with temperature, doubling with every approximately 5^oC increase in temperature.
Note: the ideal diode equation can be rearranged to find an expression for voltages

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

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

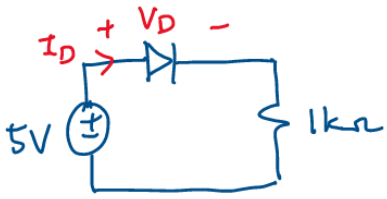
Using the ideal diode equation we can find the relationship between voltages and currents as they pass through the diode.

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

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

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.

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

Ohm's law can then be applied at the resistor

$$V_r = IR = I_o R = 5V - V_D \Rightarrow 5 - V_D = I_o R \tag{16.7}$$

V_D is the voltage across the diode

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:

- $V_o = 0.736V$

- $I_D = 4.264mA$

Section 17

Lecture 4 & 5: Forward conducting diodes

The exponential model accurately describes the diode outside of the breakdown region, though it's 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

$$I_D = I_S e^{V_D/V_T} \tag{17.1}$$

Where

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

Another equation may be produced via Kirchhoff's law

$$I_D = \frac{V_{DD} - v_d}{R} \tag{17.2}$$

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.

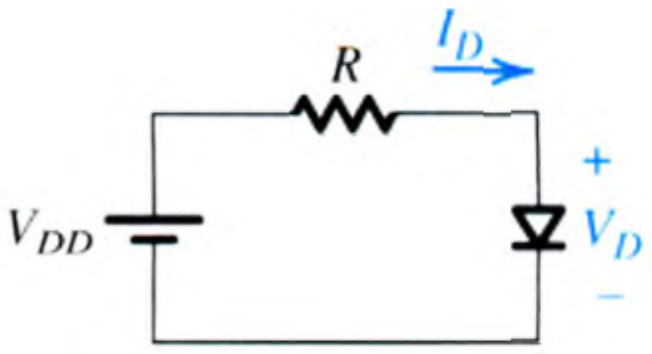
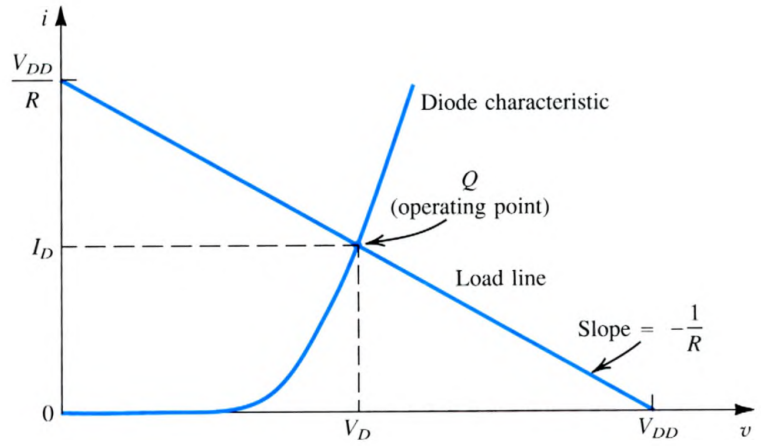


Figure 61. Simple example circuit with diode

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. 61). $V_{DD} = 5V, R = 1k\Omega$, and at $V_D 0.7V$, $I_D = 1mA$

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

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

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

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

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

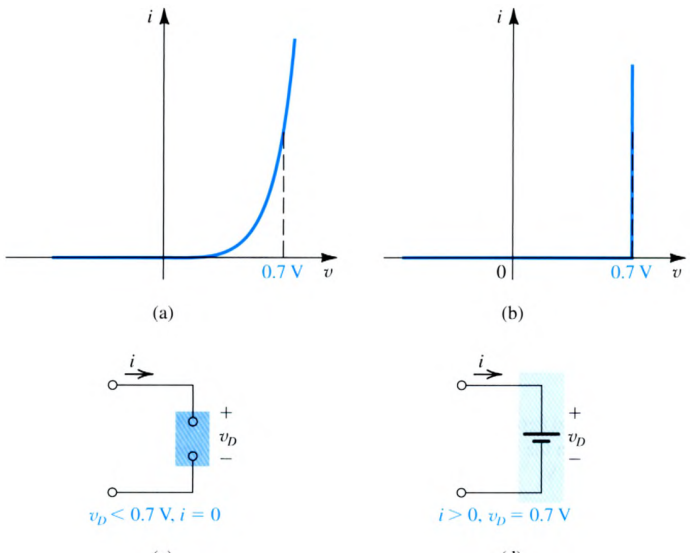
$$V_2 = 0.738V \Rightarrow I_D = 4.3mA, V_D = 0.738V \tag{17.5}$$

- This states that for a decade[44] 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 the this behaviour becomes significant is called the **cut-in voltage**

[44] *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.

$$\begin{aligned} V_D &= 0V \\ I_D &= \frac{5-0}{1} = 5mA \end{aligned} \tag{17.6}$$

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

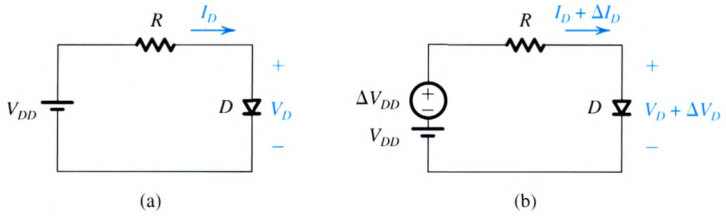
Section 18

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 ΔV_{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[45] the results are as follows: [46]

[43] *recall: passive sign convention*

[45] It is 11:17pm and I have two more lectures to catch up to today

Definition 71 **Small signal approximation**

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

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

$$r_d = \frac{V_T}{I_D} \tag{18.2}$$

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 eliminated all independent dc sources[47] 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 $\Delta V_D \lessapprox 5mV$

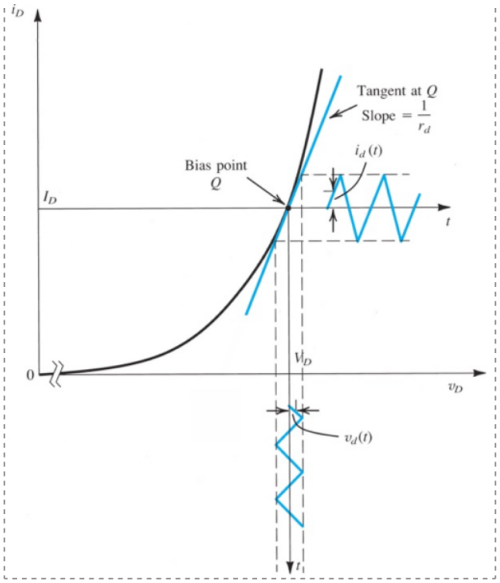
[46] Small signal analysis can be performed separately from the dc bias analysis because of the linearization of diode characteristics in the small-signal approximation

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;

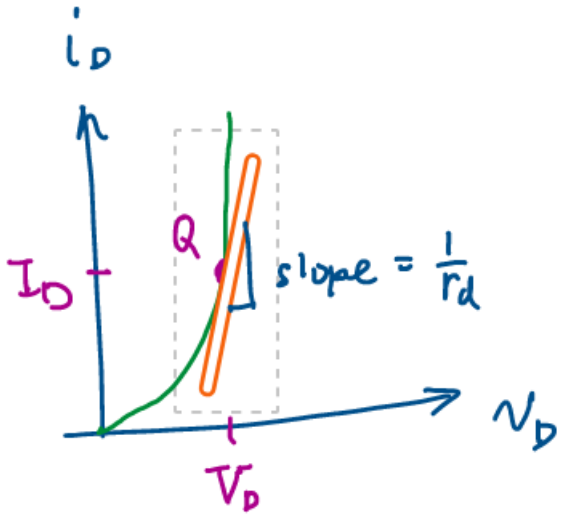
$$f(a+b+c\ldots) = f(a) + f(b) + f(c) + \ldots \tag{19.1}$$

Here we can think of the function as

$$f(V_o + v_d) = f(V_o) + f(v_d) \tag{19.2}$$

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

$$i_D = I_s e^{\frac{V_D}{V_T}} \tag{19.3}$$

$$\text{slope} = \frac{di_D}{dv_o} = I_S(\frac{1}{v_T})e^{\frac{V_D}{V_T}} \tag{19.4}$$

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

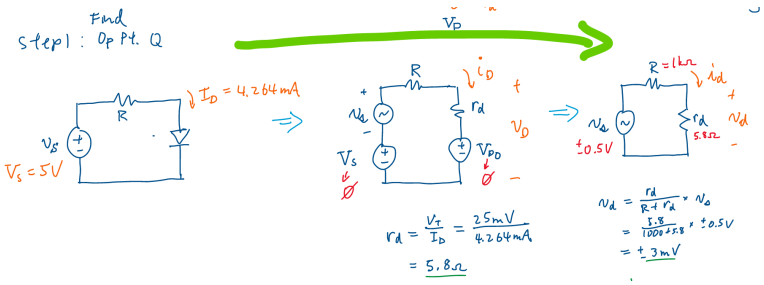
$$\text{slope} = I_S e^{V_D/V_T}(\frac{1}{V_T}) \implies \frac{I_D}{V_T} = \frac{1}{r_d} \tag{19.5}$$

And therefore

$$r_d = \frac{V_T}{I_D} \tag{19.6}$$

□

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

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

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 $\pm 5mV$

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

e^x can be expanded to a power series

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \\ &\approx 1 + x \qquad \text{if } \frac{x^2}{3!} \ll x \to x \ll 2 \end{aligned} \tag{19.9}$$

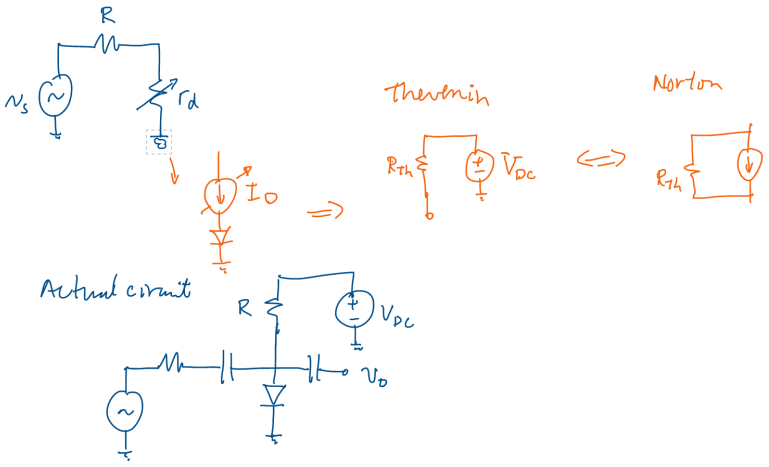
Therefore $\frac{v_d}{V_T} \ll 2$, so $v_d \ll 2V_T = 50mV$
So make

$$|v_d| < 5mV \tag{19.10}$$

□

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



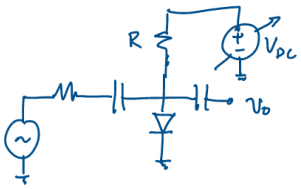
This will be covered more next lecture. TLDR: can use small signals and then build up a circuit to get the intended Q using biases.

Section 20

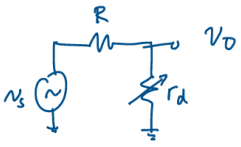
Lecture 7

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

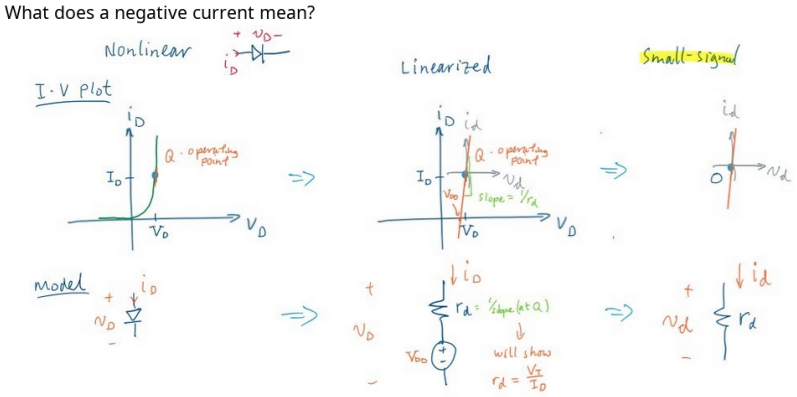
Practical circuit



Circuit concept
(Small-signal)



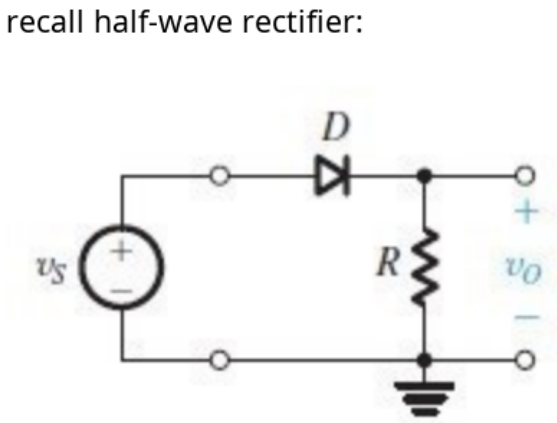
- The capacitors in this circuit can be treated like short circuits because they are large (which is why we don't see them in the small-signal circuit).
- The constant voltage offset source gets zeroed out in the small-signal circuit because it is a constant, thereby becoming a ground.
- Current sources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



Section 21

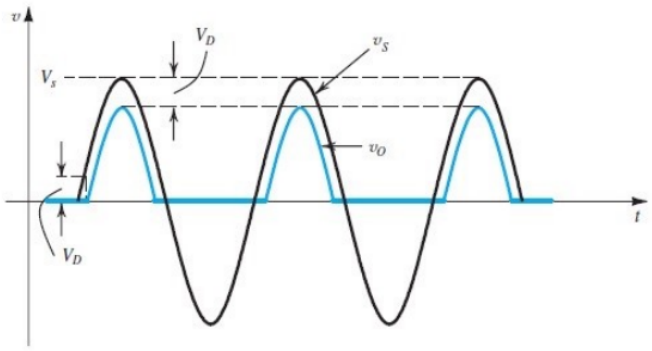
Rectifiers

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



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

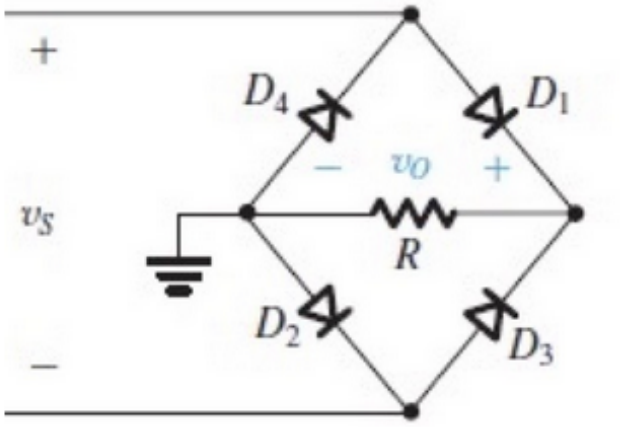


Figure 62. When the signal is positive it will flow from diode 1 to the load and then to diode 2. When it is the negative polarity it will go through diode 3 to the load and then return through diode 4. Note how the direction remains the same, and how we selectively connect the ground to the top or bottom terminal depending on the input polarity.

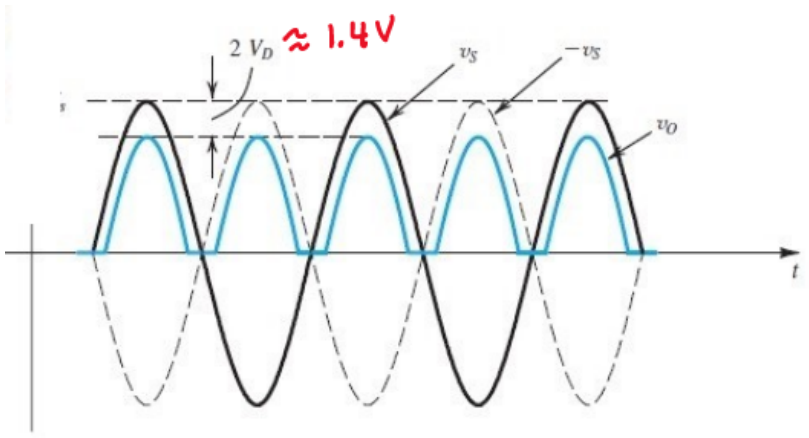


Figure 63. Comparing the two there is a little more loss but at least we get the full wave now

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

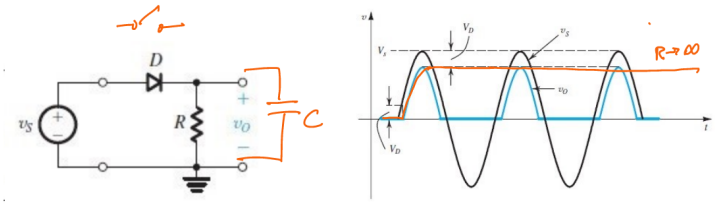
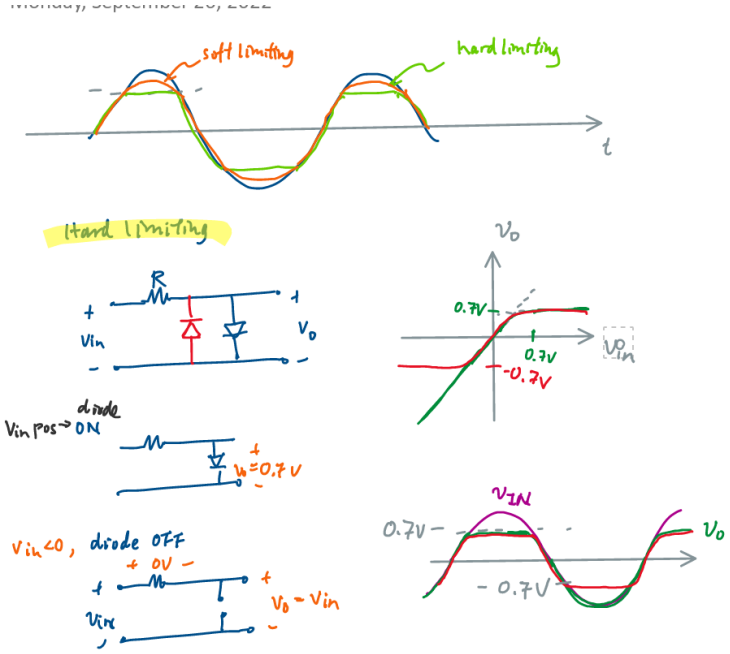


Figure 64. Capacitor size and frequency must be tuned as to avoid voltage drop (recall RC circuits. Note that the time constant $\propto RC$, so a bigger resistor (slower rate of discharge) or bigger capacitor (bigger energy storage) can reduce voltage drop)

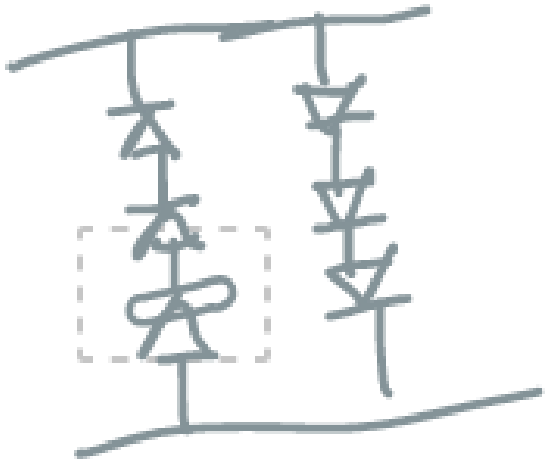
Subsection 21.1

Lecture 8

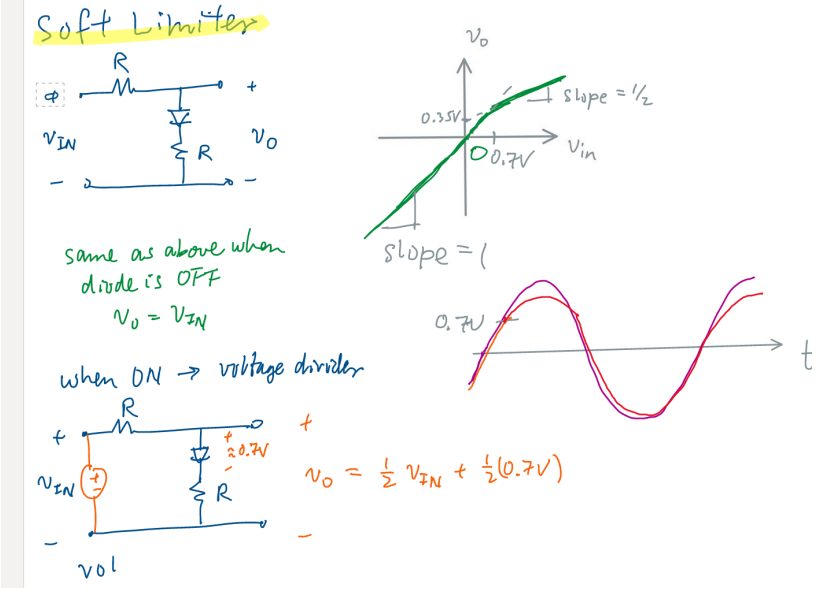
Hard limiting circuits include a pair of diodes which cause only a maximum of $0.7V$ signal to pass through.



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

$$v_o = \frac{R}{R+R} v_{in} = \frac{1}{2} v_{in} \tag{21.1}$$

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

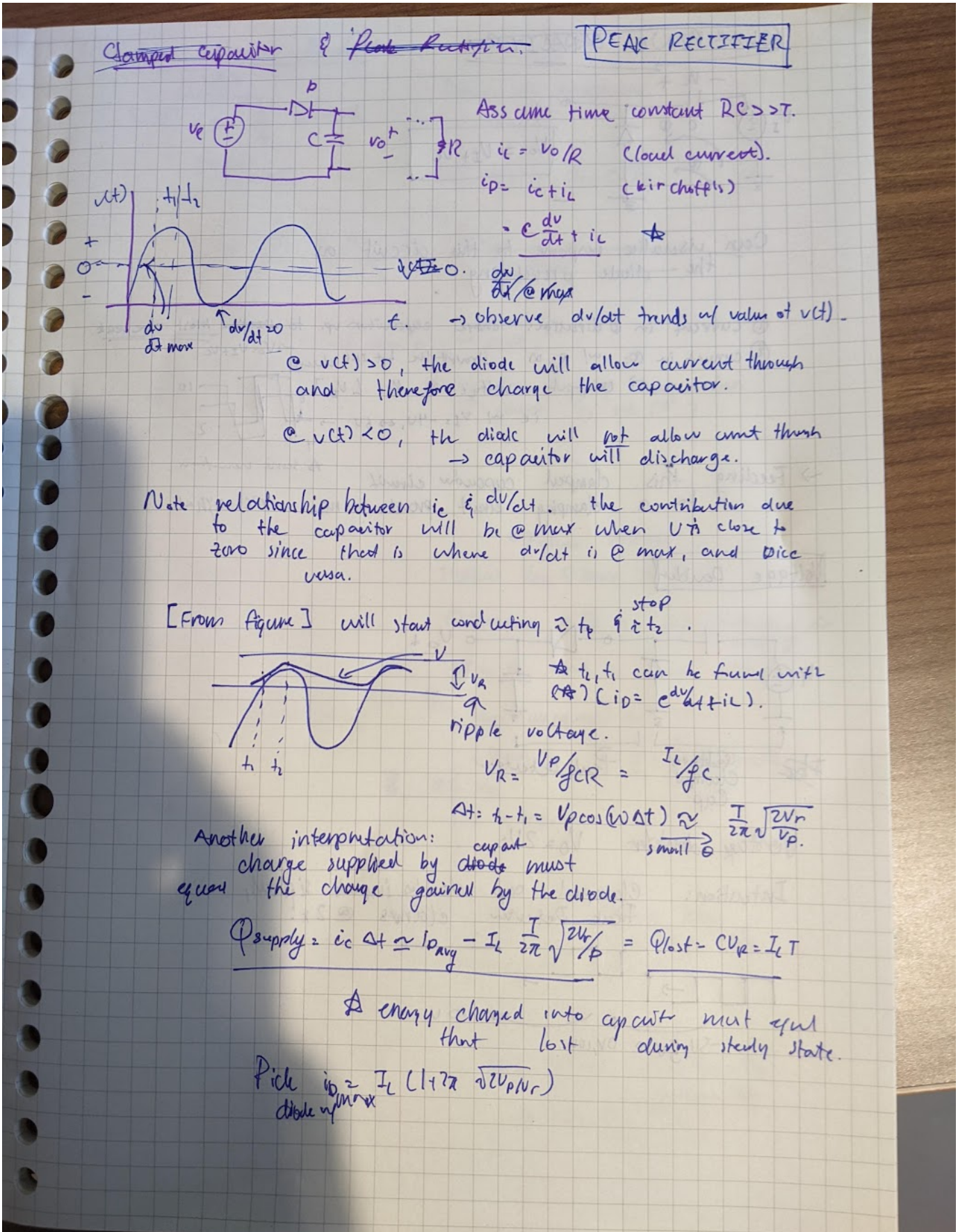
$$v_o = \frac{1}{2} v_{in} + \frac{1}{2}(v_d = 0.7V) \tag{21.2}$$

$$v_o = \frac{1}{2} v_{in} + \frac{1}{2} 0.7V \tag{21.3}$$

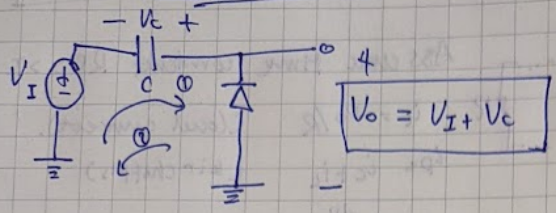
Subsection 21.2

Lecture 9

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



CLAMPING CAPACITOR



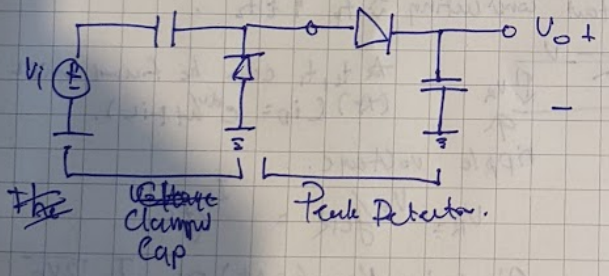
Can visualize response to this circuit as the diode preventing.

① current in ① direction: charge capacitor up. to most -ve peak.
→ Vo = VI + Vc

② current in ②: w/ $R_L = \infty$, stays at like but ideally.
→ output voltage @ Vc: [VI] 6
ie w/ VI: -4V, ⇔ 6V →
-4
10
2
As same waveform.

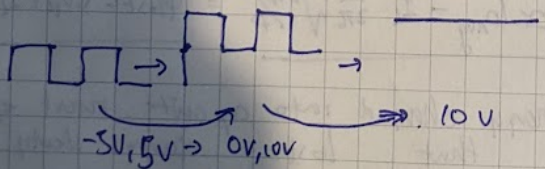
→ Feeding this clamped capacitor circuit to a clamping circuit provides a nice DC signal.

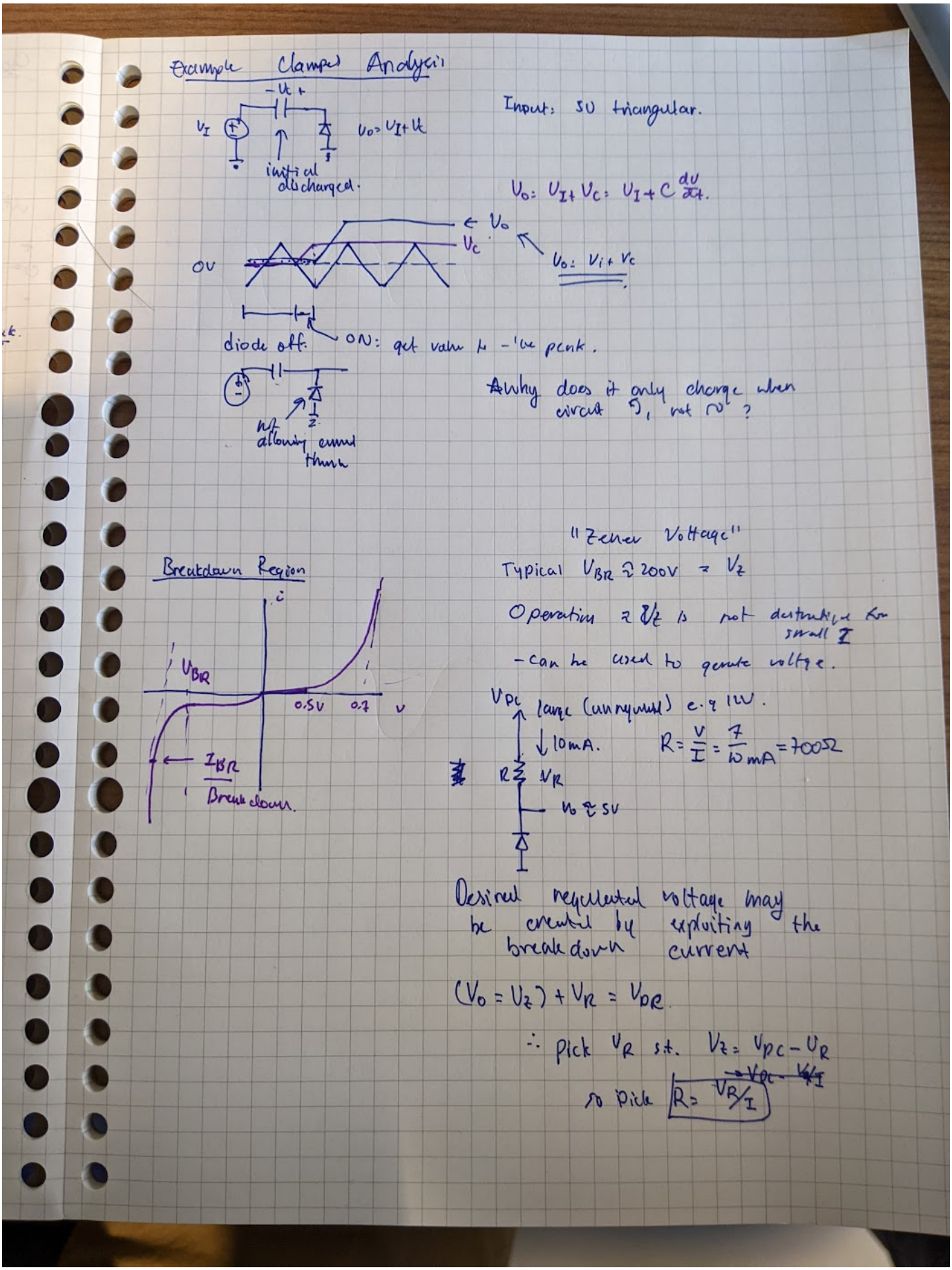
Voltage Doubler



Voltage doubler $V_O = 2V_i$

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

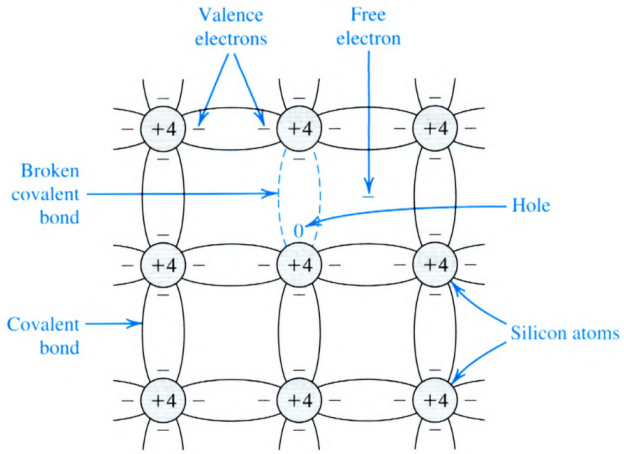



Example Clamper Analysis
Input: 5V triangular.
initial discharged.
$V_o = V_I + V_C = V_I + C\frac{dV}{dt}$.
$V_o = V_i + V_c$
0V
diode off:
ON: get value $\downarrow$ -'ve peak.
Why does it only charge when circuit $\sqcap$, not $\sim$?
not allowing current through
"Zener Voltage"
Typical $V_{BR} \approx 200V = V_z$
Operation $\approx V_z$ is not destructive for small I
- can be used to generate voltage.
Breakdown Region
V_{BR}
0.5V
0.7
v
I_{BR}
Breakdown.
V_{pc} large (unregulated) e.g. 12V.
10mA.
$R = \frac{V}{I} = \frac{7}{10\,mA} = 700\Omega$
R
V_R
$V_o \approx 5V$
Desired regulated voltage may be created by exploiting the breakdown current
$(V_o = V_z) + V_R = V_{DR}$
$\therefore$ pick V_R s.t. $V_z = V_{pc} - V_R$
$\Rightarrow$ pick $R = V_R/I$

Subsection 21.3

Lecture 10

The most commonly used semiconductor in electronics is silicon. At an 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



$$n = p = n_i \tag{21.4}$$

Where n_i denotes the number of free electrons in a unit volume of silicon at a given temperature.

Theorem 14 The number of free electrons and holes in a unit volume (cm^3) of silicon at a given temperature T is given by

$$n_i = BT^{\frac{3}{2}} e^{\frac{-E_g}{2kT}} \tag{21.5}$$

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 $E_g = 1.12eV$ 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 15 For n type silicon, the concentration of free electrons is given by

$$p_n n_n = n_i^2 \Leftrightarrow p_n \approx \frac{n_i^2}{N_P} \tag{21.6}$$

For p type silicon, the concentration of free electrons is given by

$$p_p n_p = n_i^2 \Leftrightarrow p_p \approx \frac{n_i^2}{N_A} \tag{21.7}$$

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 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 μ_p being a constant[48]. 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

[47] since we already accounted for them in step 1

$$I_{s,p} = Aqpv_{p-drift} = Aqp\mu_p E \Rightarrow J_p = qp\mu_p E \tag{21.8}$$

The current due to free electrons is found similarly and can be used to find the total drift current density

$$I_{s,n} = -Aqnv_{n-drift} \Rightarrow J_{s,n} = qn\mu_n E \tag{21.9}$$

$$J_S = J_{s,p} + J_{s,n} = q(p\mu_o + n\mu_n)E = \frac{E}{\rho} \tag{21.10}$$

Where resistivity ρ is given by

$$\rho = \frac{1}{q(p\mu_p + n\mu_n)} \tag{21.11}$$

We now observe that (21.10) is a variant of Ohm's law and can be written as

$$\rho = \frac{E}{J} \tag{21.12}$$

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[49].

[48] $\mu_p = 480cm^2/V \cdot s$ *for intrinsic silicon*

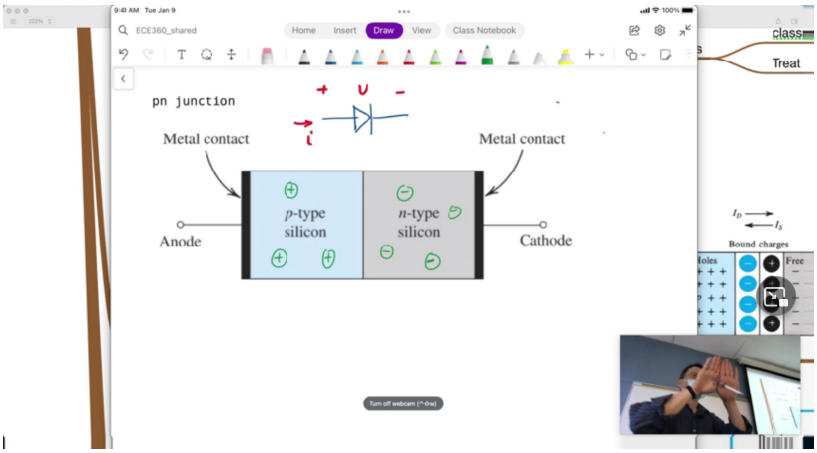
The magnitude of the current at any point is proportional to the slope of the concentration gradient, i.e. for holes

$$J_{D,p} = -qD_p \frac{dp(s)}{dx} \tag{21.13}$$

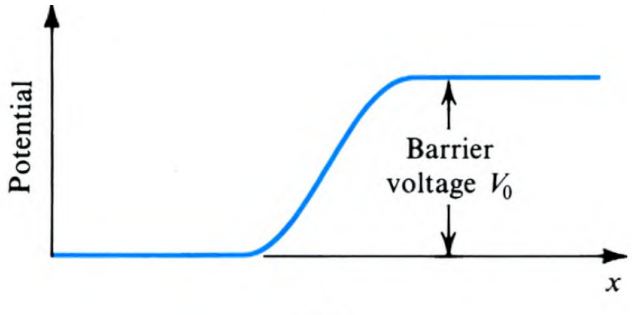
And for electrons

$$J_{D,n} = qD_n \frac{dp(s)}{dx} \tag{21.14}$$

This brings us to our first example of a semiconductor device: the $p-n$ junction diode.



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[50] dominates in the reverse bias region, while diffusion current dominates in the forward bias region.

[49] *Think of dropping ink into water*

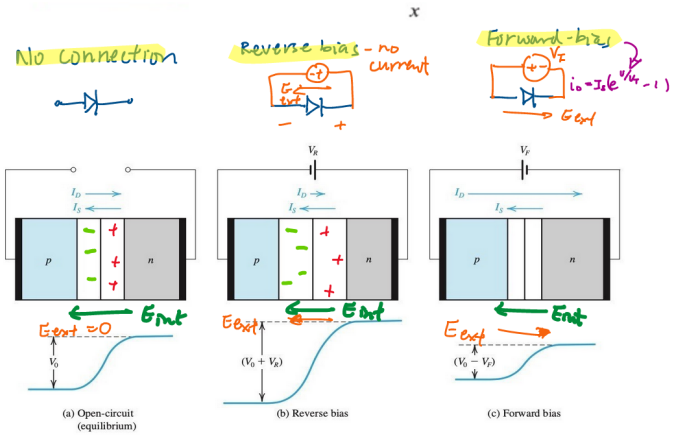


Figure 3.11 The *pn* junction in: **(a)** equilibrium; **(b)** reverse bias; **(c)** forward bias.

Skipping past the derivation, the expression for diode current is given by

$$i_D = I_s(e^{\frac{v_D}{v_T}-1}) \tag{21.15}$$

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 65. 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.

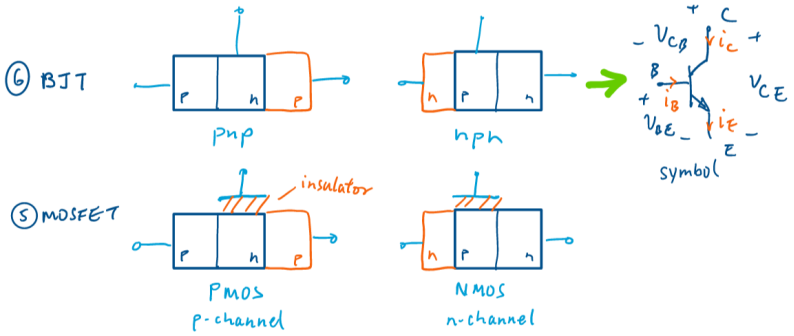


Figure 66. BJT transistors came first but are not commonly used now; they will be discussed later on in the course. Whereas BJTs directly make contact with the semiconductor, MOSFETs apply a field to induce a change in the device.

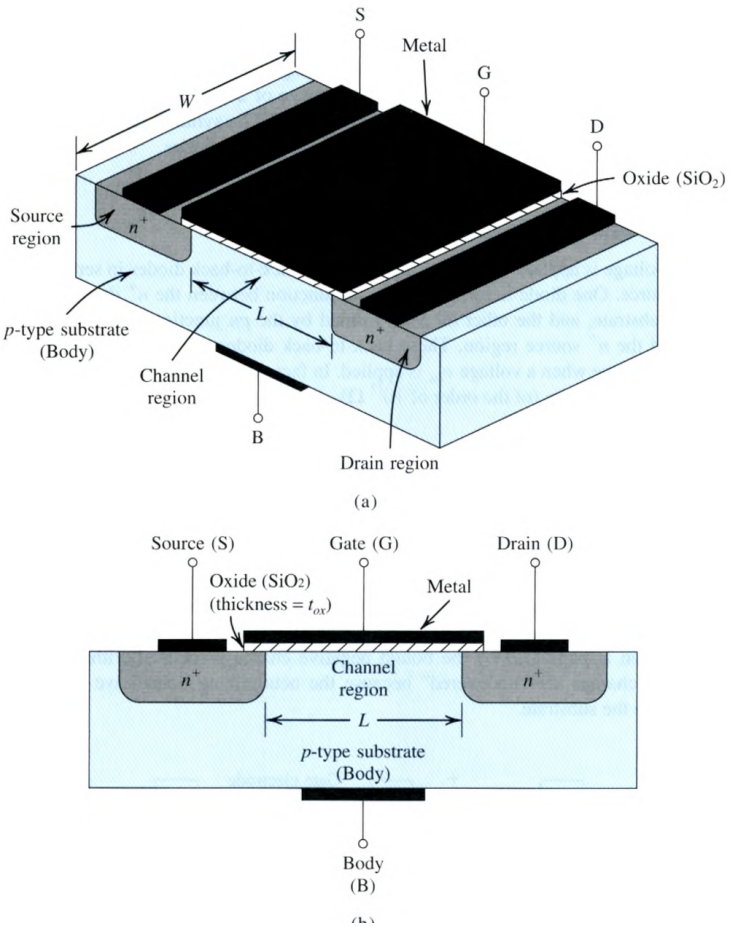


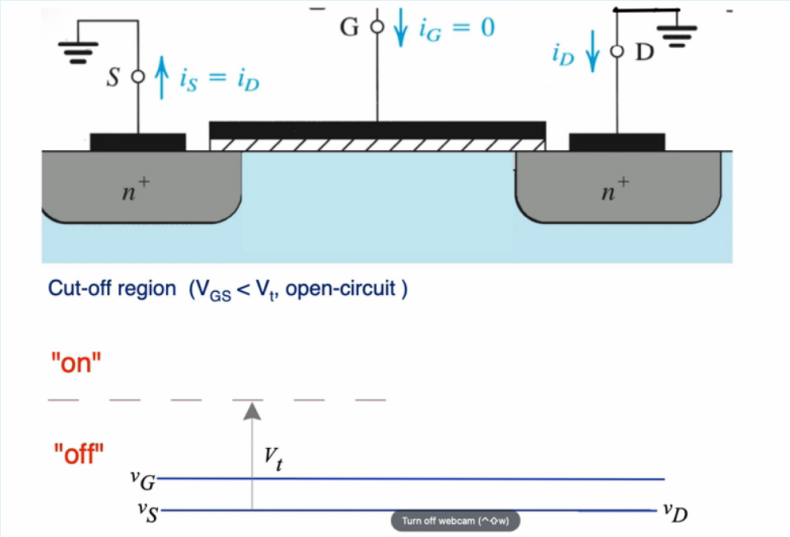
Figure 67. CMOS transitors

P transistors use holes as their primary charge carrier and allow current to pass if the base is negative with respect to the emitter[51]. Conversely, N transistors carry charge using electrons and allow current to pass if the base is positive with respect to the emitter[52]

Definition 72 **Cut-off region**

[50] *recall: movement of carriers due to electric field, electrons drifting back to n region*

[51] *For a PMOS this would be the application of a negative 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"[53] through which electrons may pass.

[53] *again, for a n-type*

Subsection 21.5

Lecture 12: MOSFET Operating Regions

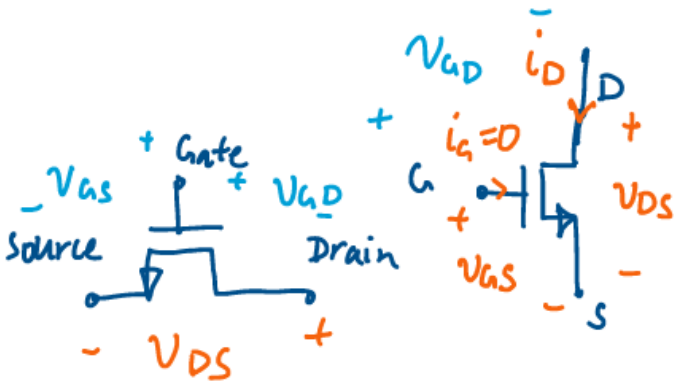


Figure 68. Labelled NMOS. The drain current can be thought of as the output.

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

-

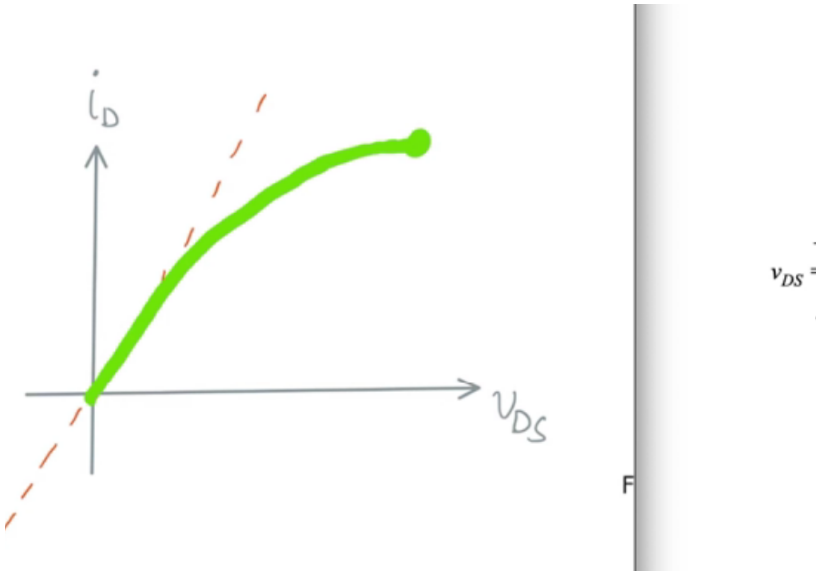
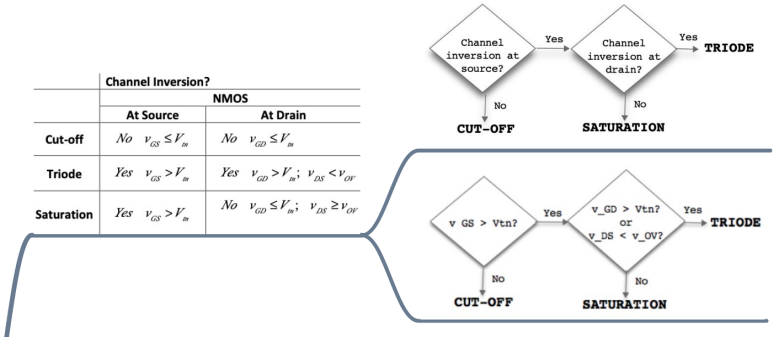


Figure 69. The n-channel will thin out as the drain voltage goes up; at some point we will hit a drain voltage equal to the threshold voltage and cause the channel to disappear

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.



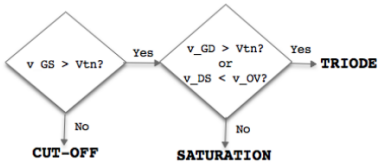
Remark For a NMOS the source is the terminal at the lower voltage and the drain is the terminal at the higher voltage. For a PMOS it is the opposite, i.e. the source is at higher voltage. It is conceivable that the source and drain could be swapped during operation.

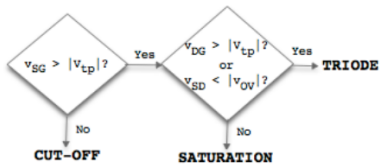
52 For a NMOS this would be the application of a positive field.

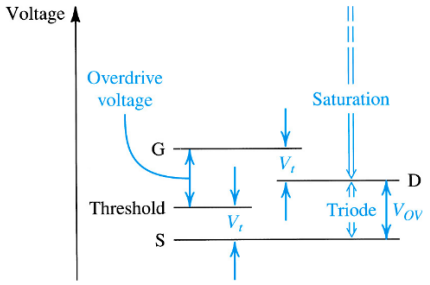
V_t is the threshold voltage. Note that since this is a symmetrical device, we can compare V_{GS} and V_{GD} w.r.t the source and drain

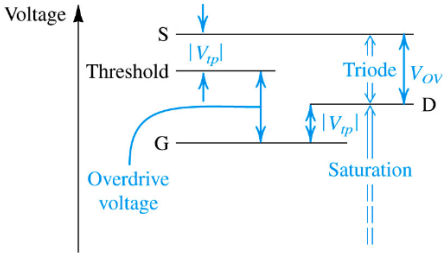
MOSFET Regions of Operation v_{OV} : **Overdrive voltage** $v_{OV} = v_{GS} - V_{tn}$

	Channel Inversion?			
	NMOS		PMOS	
	At Source	At Drain	At Source	At Drain
Cut-off	*No* $v_{GS} \le V_{tn}$	*No* $v_{GD} \le V_{tn}$	*No* $v_{SG} \le \lvert V_{tp}\rvert$	*No* $v_{DG} \le \lvert V_{tp}\rvert$
Triode	*Yes* $v_{GS} > V_{tn}$	*Yes* $v_{GD} > V_{tn}$; $v_{DS} < v_{OV}$	*Yes* $v_{SG} > \lvert V_{tp}\rvert$	*Yes* $v_{DG} > \lvert V_{tp}\rvert$; $v_{SD} < \lvert v_{OV}\rvert$
Saturation	*Yes* $v_{GS} > V_{tn}$	*No* $v_{GD} \le V_{tn}$; $v_{DS} \ge v_{OV}$	*Yes* $v_{SG} > \lvert V_{tp}\rvert$	*No* $v_{DG} \le \lvert V_{tp}\rvert$; $v_{SD} \ge \lvert v_{OV}\rvert$





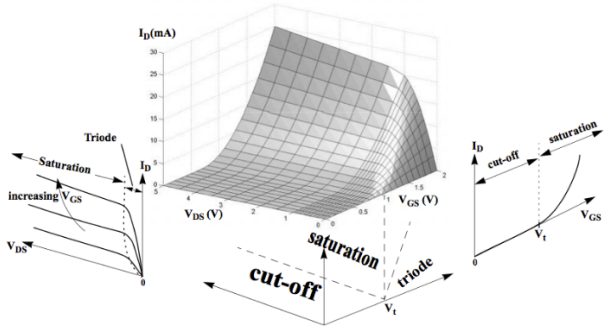




MOSFET Device Equations

Operating Region	NMOS	PMOS
Cut-off	$i_D = 0$	$i_D = 0$
Triode	$i_D = \mu_n C_{ox}\left(\frac{W}{L}\right)\left[(v_{GS} - V_{tn})v_{DS} - \frac{v_{DS}^2}{2}\right]$	$i_D = \mu_p C_{ox}\left(\frac{W}{L}\right)\left[(v_{SG} - \lvert V_{tp}\rvert)v_{SD} - \frac{v_{SD}^2}{2}\right]$
Saturation	$i_D = \frac{1}{2}\mu_n C_{ox}\left(\frac{W}{L}\right)(v_{GS} - V_{tn})^2[1 + \lambda v_{DS}]$	$i_D = \frac{1}{2}\mu_p C_{ox}\left(\frac{W}{L}\right)(v_{SG} - \lvert V_{tp}\rvert)^2[1 + \lvert\lambda\rvert v_{SD}]$

n-Channel MOSFET Current-Voltage Characteristic



Subsection 21.6

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

Taught by Prof. Sigil

Section 22

Complex Numbers

Subsection 22.1

Lecture 1

Consider a 2-vector $\vec{x} = (x, y) \in \mathcal{R}$. As complex numbers correspond to two-vectors

$$\vec{x} = (x, y) \leftrightarrow z = x + iy, i^2 = -1 \tag{22.1}$$

z is, therefore, a complex variable. What are the benefits of a complex number like z?

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 73 **Imaginary and Complex Numbers**

i is an imaginary number such that

$$i^2 = -1 \tag{22.2}$$

A complex number has the form:

$$z = x + iy \tag{22.3}$$

Definition 74 There are a number of operations we can perform on complex numbers.

Addition

$$z + z' = (x + x') + i(y + y') \tag{22.4}$$

Multiplication

$$zz' = (x + iy)(x' + iy') = (xx' - yy') + i(xy' + x'y) \tag{22.5}$$

Proof

Proof of (22.5):

$$\begin{aligned} zz' &= (x + iy)(x' + iy') \\ &= x + ixy' + iyx' + i^2yy' \\ &= xx' - yy' + i(xy' + yx') \end{aligned} \tag{22.6}$$

□

Magnitude

$$|z| = \sqrt{x^2 + y^2} \tag{22.7}$$

Conjugate
The complex conjugate has the properties:

- $\overline{z}z = |z|^2$
- $\overline{(z + z')} = \overline{z} + \overline{z}'$
- $\overline{z \cdot z'} = \overline{z} \cdot \overline{z}'$

We can define a new operation

$$\forall \text{complex} z, \exists \quad \text{complementary number } w \text{ such that } zw = wz = 1 \tag{22.8}$$

Denote

$$w = \frac{1}{z} = z^{-1} \tag{22.9}$$

Proof

Proof of (22.9): Find w s.t. $zw = 1$

$$\begin{aligned} zw &= 1 \\ w\overline{z}z &= \overline{z}z = |z|^2 > 0 \\ |z|^2 w &= \overline{z} \\ w &= \frac{\overline{z}}{|z|^2} \to Z^{-1} = \frac{\overline{z}}{|z|^2} \end{aligned} \tag{22.10}$$

□

Furthermore, there are operators that we can define on complex numbers.

Definition 75

Real and Imaginary Operators
Given $z = x + iy$, we can define the real and imaginary operators

$$x = Re\{z\} \tag{22.11}$$

$$y = Im\{z\} \tag{22.12}$$

Example

$$Im\left\{(1 = \sqrt{2}i)^{-}1\right\} \tag{22.13}$$

By (22.9), we have

$$Im\left\{z^{-1}\right\} = \frac{-Im\{z\}}{|z|^2} \tag{22.14}$$

And

$$Re\left\{z^{-1}\right\} = \frac{-Re\{z\}}{|z|^2} \tag{22.15}$$

Using these, for example, we find that the $Im = \frac{-\sqrt{2}}{3}$
We can get the real component in a similar way.

Here is an enumeration of absolute value properties for complex numbers:

$$|z \cdot w| = |z||w| \tag{22.16}$$

$$|z + w| \leq |z| + |w| \tag{22.17}$$

$$|\overline{z}| = |z| \tag{22.18}$$

$$|z + w|^2 = (\overline{x} + \overline{w})(z + w) = |z|^2 + |w|^2 + \overline{z}w + \overline{w}z \tag{22.19}$$

Proof

Note that $\overline{z}w + \overline{w}z = 2Re\{z\overline{w}\}$, by (22.19)
And so

$$|z + w|^2 \leq |z|^2 + |w|^2 + 2|z||w| = (|z| + |w|)^2 \tag{22.20}$$

□

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 16 **Polar Decomposition**

Complex numbers can be expressed in polar form as well

$$z = r(\cos\theta + i\sin\theta) \tag{22.21}$$

Where

$$r = |z| \qquad x = r\cos\theta \qquad y = r\sin\theta \qquad \theta = \tan^{-1}\left(\frac{y}{x}\right) \tag{22.22}$$

This has a number of useful properties

$$z \cdot z' = |z||z'|(\cos(\theta + \theta') + i\sin(\theta + \theta')) \tag{22.23}$$

$$\frac{z}{z'} = \frac{|z|}{|z'|}(\cos(\theta - \theta') + i\sin(\theta - \theta')) \tag{22.24}$$

Proof

Proof for (22.23):

$$\begin{aligned} z \cdot z' &= |z|(\cos(\theta + i\sin\theta)) \times |z'|(\cos\theta' + i\sin\theta') \\ &= |z||z'|(\cos\theta\cos\theta' + i\cos\theta\sin\theta' + i\sin\theta\cos\theta' - \sin\theta\sin\theta') \\ &= |z||z'|\left[\cos\theta\cos\theta' - \sin\theta\sin\theta' + i(\cos\theta\sin\theta' + \sin\theta\cos\theta')\right] \end{aligned} \tag{22.25}$$

And the proof follows □

Lemma 5 A corollary exists

$$z^2 = |z|^2(\cos 2\theta + i\sin 2\theta) \tag{22.26}$$

Theorem 17 **Moivre's Theorem**

$$z^n = |z|^n(\cos(n\theta) + i\sin(n\theta)) \tag{22.27}$$

More generally,

$$(\cos\theta + i\sin\theta)^n = \cos(n\theta) + i\sin(n\theta) \tag{22.28}$$

So we may define z to be the n^{th} root of w which implies that

Lemma 6 Every complex number has a n^{th} root $\forall n$

Proof

$$\text{Let } z = |w|^{\frac{1}{n}}(\cos\frac{\theta}{n} + i\sin\frac{\theta}{n}) \tag{22.29}$$

Then

$$w = |w|(\cos\theta + i\sin\theta), \text{ then } z^n = w \tag{22.30}$$

□

Intuition: define z to be $\frac{1}{n}$ and then take the n^{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[54].

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 $V_{GD<V_t}$

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 76 Given a domain $\mathbb{D} \in \mathbb{C}$, a function f is a rule such that

$$z \in \mathbb{D} \xrightarrow{f} w \in \mathbb{D} \leftrightarrow w = f(z) \tag{22.31}$$

Definition 77 We may define $\mathbb{D}$ to be the domain of f
Likewise, range is defined as

$$Ran\{f\} = \{w \in \mathbb{C} : \exists z \in D : f(z) = w\} \tag{22.32}$$

Example

$$f(z) = \frac{1}{z+i} \tag{22.33}$$

What is the maximum domain of f?

$$Dom\{f\} = \{z \in \mathbb{C} : |z| < -i\} \tag{22.34}$$

What is the range of f?

$$\frac{1}{z+i} = w \tag{22.35}$$

For which values of w can we solve this equation?

$$z = -i + \frac{1}{w} \tag{22.36}$$

So the range of the function is

$$Ran\{f\} = \{w \in \mathbb{C} : |w| \neq 0\} \tag{22.37}$$

Example

$$f(z) = z^2 + 1 \tag{22.38}$$

It is fairly clear that $Dom\{f\} = f \in \mathbb{C}$
The range can be found by solving for z in

$$z^2 + 1 = w \tag{22.39}$$

And so

$$Ran\{f\} = \{w \in \mathbb{C}\} \tag{22.40}$$

22.2.2 Exponential Functions

Definition 78 Given $z = x + iy$,

$$e^z = e^x(\cos y + i \sin y) = e^{Re\{z\}}(cos(Im\{z\}) + i \sin(Im\{z\})) \tag{22.41}$$

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) □

More properties:

- $Dom\{e^z\} = \mathbb{C}$
- $Ran\{e^z\} = \{\mathbb{C} \setminus \{0\}\}$
- $e^z = w \qquad \text{if } w \neq 0$

arg, or argument is the angle from the real axis to that on the complex plane. Usually denoted by θ

55

54 *They are part of a cyclic group*

$$\begin{aligned} z &= ln|w| + i \arg w \\ e^z &= e^{ln|w| + i \arg w} \\ &= e^{ln|w|} e^{i \arg w} \\ &= |w| \cos(\arg w) + i \sin(\arg w) \\ &= w \end{aligned} \tag{22.42}$$

Remark **Polar representation**

$$w = |w| e^{i \arg w} \tag{22.43}$$

Example **Find polar coordinates of** $z = i + 1$

$$r = |w| \quad \theta = \arg w \tag{22.44}$$

$$\begin{aligned} |z| &= \sqrt{1+i} = \sqrt{2} \\ \cos\theta &= \frac{1}{\sqrt{2}} \to \theta = \frac{\pi}{4} \\ z &= \sqrt{2}e^{i\pi/4} \end{aligned} \tag{22.45}$$

Example Find

$$(1+i)^{\frac{1}{3}} \tag{22.46}$$

Solution: $z = \sqrt{2}e^{\frac{i\pi}{4}} \to z^{1/3} = 2^{\frac{1}{6}}e^{i\pi/12}$

Definition 79 **Trig functions for complex numbers**

$$\cos z = \frac{1}{2}\left(e^{iz} + e^{-iz}\right) \tag{22.47}$$

Proof

$$\cos x = \frac{1}{2}(e^{ix} + e^{-ix}) = \frac{1}{2}\left(\cos x + i\sin x + \underbrace{\cos(-x)}_{\text{odd; }=\cos(x)} + \underbrace{i\sin(-x)}_{\text{even; }=-\sin(x)}\right) = \cos x \tag{22.48}$$

□

$$\sin z = \frac{1}{2}\left(e^{iz} - e^{-iz}\right) \tag{22.49}$$

And a similar proof follows for $\sin z$.
These have the following properties

$$\sin z|_{ImZ=0} = \sin x \tag{22.50}$$

$$\cos(z + 2\pi n) = \cos z \forall n \in \mathbb{Z} \tag{22.51}$$

$$\sin(z + 2\pi n) = \sin z \forall n \in \mathbb{Z} \tag{22.52}$$

Proof

$$\begin{aligned} \cos z + 2\pi n &= \frac{1}{2}(e^{i(z+2\pi n)} + e^{-i(z+2\pi n)}) \\ &= \frac{1}{2}(e^{iz}e^{i2\pi n} + e^{-iz}e^{-i2\pi n}) \\ &= \frac{1}{2}(e^{iz} + e^{-iz}) \\ &= \cos z \end{aligned} \tag{22.53}$$

□

The domain of $\{\cos z, \sin z\} = \mathbb{C}$
Range?
Solve $\cos z = w$ for z

$$\begin{aligned} \frac{1}{2}(e^{iz} + e^{-iz}) &= w \\ \dots \times 2e^{iz} \text{ on both sides} & \\ e^{2iz} - 2we^{iz} + 1 &= 0 \\ \dots \text{Let } S = e^{iz} & \\ S^2 - 2ws + 1 &= 0 \\ S = w \pm \sqrt{w^2 - 1} &\equiv u \end{aligned} \tag{22.54}$$

Now we note that $e^{iz} = u$ can be solved for z for any $u \neq 0$

$$u = 0 \leftrightarrow w = \pm\sqrt{w^2 - 1} \tag{22.55}$$

$$w^2 = w^2 - 1 \text{ impossible for } u \neq 0 \tag{22.56}$$

Therefore:

$$Ran\{\cos z\} = Ran\{\sin z\} = \mathbb{C} \tag{22.57}$$

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

$$\exp : e^z = e^x(\cos y + i \sin y) \tag{22.58}$$

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.

$$e^{z+2\pi mi} = e^z \tag{22.59}$$

Consider the equation

$$e^z = w \tag{22.60}$$

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 80

$$\log \equiv \log w = \ln |w| + i \arg w \tag{22.61}$$

$w \neq 0$

Proof

Proof that $\log w = \ln |w| + i \arg w$

$$\begin{aligned} w = &= |w| e^{i \arg w} \\ &= e^{\ln |w|} e^{i \arg w} \\ &= e^{\ln |w| + i \arg w} \\ \rightarrow e^z &= e^{\ln |w| + i \arg w} \\ \Rightarrow z &= \ln |w| + i \arg w \end{aligned} \tag{22.62}$$

□

Note that $\arg$ is a multivalued function, i.e

$$\arg w = \arg(w + 2\pi m_i), i \in \mathbb{Z}, \arg w \in [-\pi, \pi) \tag{22.63}$$

Example

$$e^z = e^5 \tag{22.64}$$

Solve for z

$$z = 5 + 2\pi m i, m \in \mathbb{Z} \tag{22.65}$$

Example

Solve $e^z = i$

$$i = e^{\frac{i\pi}{2}} \tag{22.66}$$

The solution is therefore

$$z = i(\pi/2 + 2\pi m), m \in \mathbb{Z} \tag{22.68}$$

Note that providing only a single solution is wrong; must provide all

$$|i| = 1; \arg i = \frac{\pi}{2} \tag{22.67}$$

The complex logarithm is a multivalued function (like $\arg$).

$$\log z = \ln |z| + i \arg z, \arg z \in [-\pi, \pi) \tag{22.69}$$

Note that $i \arg z$ demotes 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 $\arg w = \frac{3\pi}{4}$.

$$\arg(zw) = ? \tag{22.70}$$

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 2π [56]

$$\arg(zw) = \frac{17\pi}{12} - 2\pi = -\frac{7\pi}{12} \in [-\pi, \pi) \tag{22.71}$$

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.

[56] *since adding $\pm 2\pi$ doesn't change the angle, just rotates it around once*

$$\begin{aligned}\log zw &= \ln|zw| + i\arg zw \\ &\neq \ln|z| + \ln|w| + i\arg z + i\arg w\end{aligned} \tag{22.72}$$

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

$$\therefore \log(zw) = \log z + \log w \tag{22.73}$$

Example Compute $\log\left(\sqrt{3}+i\right)$.
Just apply the formula.

$$\log w = \ln|w| + i\arg w \tag{22.74}$$

$$\log\left(\sqrt{3}+i\right) = \ln\sqrt{4} + i\arg(\sqrt{3}+i) = \ln 2 + i(\frac{\pi}{6} + 2\pi m), m \in \mathbb{Z} \tag{22.75}$$

22.3.3 Powers

Definition 81

$$\forall a \neq 0, a^z \equiv e^{z\log a} \tag{22.76}$$

Example Complete $(1+i)^i$

$$\begin{aligned}(1+i)^i &= e^{i\log(1+i)} \\ \log(1+i) &= \ln\sqrt{2} + i(\frac{\pi}{4} + 2\pi m), m \in \mathbb{Z} \\ \ldots &= e^{-\frac{\pi}{4}-2\pi m} e^{i\ln\sqrt{2}}\end{aligned} \tag{22.77}$$

Subsection 22.4

Lecture 4

Definition 82

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

$$f : \mathbb{D} \to \mathbb{C} \qquad \text{to be the analytic of } z_o \in \mathbb{D} \tag{22.78}$$

Now, given f, we can define the complex derivative f'

$$f'(z_0) = \lim_{z\to z_0} \frac{f(z) - f(z_0)}{z - z_0} \qquad \text{exists} \tag{22.79}$$

Noting that

$$z \to z_0 \leftrightarrow |z - z_0| \to 0 \tag{22.80}$$

(22.79) can be rewritten as

$$f'(z_0) = \lim_{h\to 0} \frac{1}{h}(f(z_0+h) - f(z_0)) \tag{22.81}$$

$h = z - z_0; z = z_0 + h$

Example Find the complex derivative

$$f(z) = z^n \tag{22.82}$$

Proof

$$\begin{aligned} f'(z) &= \lim_{h\to 0} \frac{1}{h}\left((z+h)^n h - z^n\right) \\ &= \lim \frac{1}{h}\left(z^n = nz^{n-1}h + \binom{n}{2} z^{n-2}h^2 + \ldots + h^n - z^n\right) \\ &= \lim \frac{1}{h}\left(nz^{n-1} + \binom{n}{2} z^{n-2}h + \ldots h^{n-1}\right) \end{aligned} \tag{22.83}$$

... Cancel out terms that go to 0

$$\begin{aligned} &= (z^n)' \\ &= nz^{n-1} \end{aligned}$$

□

$z = x + iy, \overline{z} = x - iy$

Example Find the complex derivative

$$f(z) = \overline{z} \tag{22.84}$$

Proof

$$\begin{aligned} f'(z) &= \lim_{h\to 0} (\overline{z} + \overline{h} - \overline{z}) \\ &= \lim_{h\to 0} \frac{\overline{h}}{h} \end{aligned} \tag{22.85}$$

We then take the limit along the real and imaginary axis separately $h = h_1 + ih_2$

Real:

$$\lim_{h\to 0} \frac{h_1}{h_1} = 1 \tag{22.86}$$

Imaginary:

$$\lim_{h\to 0} \frac{-h_2}{h_2} = -1 \tag{22.87}$$

□

So the limit (22.85) does not exist

In the previous two examples we found that z^n is analytic and $\overline{z}$ is not.

Example

$$f(z) = e^z \tag{22.88}$$

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 $e^{x+h_1}, \cos(y+h_2), \sin(y+h_2)$

$$e^{x+h_1} cos(y+h_2) \times \tag{22.90}$$

$$\left(e^x + e^x \frac{h_1}{1!} + \text{higher order terms}\right) \times \tag{22.91}$$

$$\left(\cos y - \frac{\sin y}{1!} h_2 + \text{higher order terms}\right)$$

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

$$= e^x \cos y - e^x \sin y + e^x h_1 \cos y + \ldots \text{higher order terms} \tag{22.92}$$

And as a result

$$(e^z)' = lim_{h\to 0} \frac{1}{h} e^z h = e^z \tag{22.93}$$

□

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 $g \in$ domain of f

Example

$$(e^{z^3}) = e^{z^3}(z^3)' = 3z^2 e^{z^3} \tag{22.94}$$

Definition 83 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 18 **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[57]

[57] *Complex differentiable*

$$f(z) = u + iv \tag{22.96}$$

Then,

$$f'(z) = \frac{\partial u}{\partial x} + i\frac{\partial v}{\partial x} = \frac{\partial v}{\partial y} - i\frac{\partial u}{\partial y} \tag{22.97}$$

In particular we're interested in the this set of PDEs (which is called the Cauchy-Riemann equations)

$$\begin{aligned} \frac{\partial u}{\partial x} &= \frac{\partial v}{\partial y} \\ \frac{\partial u}{\partial y} &= -\frac{\partial v}{\partial x} \end{aligned} \tag{22.98}$$

The short form is as follows

$$u_x = v_y \qquad u_y = -v_x \tag{22.99}$$

If $f = u + iv$ and in $\mathbb{D}$, then u, v satisfy

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y} \tag{22.100}$$

And

$$\frac{\partial u}{\partial y} = -\frac{\partial v}{\partial x} \tag{22.101}$$

Proof

Since f is analytic, then

$$f'(z) = \lim_{h\to 0} \frac{1}{h}(f(z+h) - f(z)) \tag{22.102}$$

Which exists and is independent of the way $h \to 0$.
Take the limits of the real and imaginary parts of (22.102):
Real:

$$\lim_{h_1\to 0, h_1\in\mathbb{R}} \frac{1}{h_1}(f(z+h_1)-f(z)) = \lim \frac{1}{h_1}(f(x+h_1+iy)-f(x+iy)) = \partial_x f(z) \tag{22.103}$$

$$\lim_{ih_2\to 0, h_2\in\mathbb{R}} \frac{1}{h_2}(f(z+ih_2) - f(z)) = \lim \frac{1}{ih_2}(f(x+ih_2+iy) - f(x+iy)) = -\partial_y f(z) \tag{22.104}$$

Since this limit is independent of how $h \to 0$,

$$\partial_x f(z) = -i\partial_y f(z) \tag{22.105}$$

Recall that $f = u + iv$

$$\partial_x(u+iv) = -i\partial_y(u+iv) = -i\partial_y u + \partial_y v \tag{22.106}$$

Therefore

$$\partial_x u = \partial_y v, \partial_x v = -\partial_y u \tag{22.107}$$

□

Definition 84

Complex derivative:
Using the Cauchy-Riemann equations, we can define the complex derivative of f as

$$\frac{\partial f}{\partial z}\frac{1}{2}(\frac{\partial f}{\partial x} - i\frac{\partial f}{\partial y}) \tag{22.108}$$

$$\frac{\partial f}{\partial \overline{z}}\frac{1}{2}(\frac{\partial f}{\partial x} - i\frac{\partial f}{\partial y}) \tag{22.109}$$

Then, via the Cauchy-Riemann equations, we have

$$\frac{\partial f}{\partial \overline{z}} = 0 \tag{22.110}$$

Proof

Proof of (22.110):
Recall $f = u + iv$
Plug this into the LHS of the expression:

$$\frac{\partial u}{\partial x} + i(\frac{\partial v}{\partial y}) + i(\frac{\partial u}{\partial x} + i\frac{\partial v}{\partial y}) \Rightarrow \partial_x u - \partial_y v = 0, \partial_y u + \partial_x v = 0 \tag{22.111}$$

Which is the Cauchy-Riemann equations (22.107)

□

Example

Is $f(z) = e^{z^5} \cdot \sin z \cdot \overline{z}^3$ analytic?

$$\frac{\partial f}{\partial z} e^{z^5} \cdot \sin z e\overline{z}^2 \neq 0 \tag{22.112}$$

So f is not analytic

Example Is $f(z) = |z|^6$ analytic?

Proof

$$\frac{\partial y}{\partial \overline{z}} = \frac{\partial}{\partial z}(z\overline{z})^3 = z^3 \cdot 3\overline{z}^2 \neq 0 \tag{22.113}$$

□

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 19 Given $f(x, y)$ continuously differentiable and satisfies the Cauchy-Riemann equations, then f is analytical

Note abbreviation higher order terms $\Leftrightarrow$ H.O.T

Proof

$$\begin{aligned} f(x+h_1, y+h_2) - f(x,y) &\xrightarrow{\text{Taylor series}} \\ &= f(x,y) + \partial_x f(x,y)h_1 + \partial_y f(x,y)h_2 + \text{H.O.T} - f(x,y) \\ &\ldots \text{cancel terms} \ldots \\ &= \partial_x = \frac{\partial f}{\partial z} + \frac{\partial f}{\partial \overline{z}} \\ &= \partial_y = \frac{\partial f}{\partial \overline{z}} - \frac{\partial f}{\partial z}\frac{1}{i} \end{aligned} \tag{22.114}$$

And then plugging this back into the initial expression we get

$$\begin{aligned} f(x+h_1, y+h_2) - f(x,y) &= (\frac{\partial f}{\partial z}\frac{\partial f}{\partial \overline{z}})h_1 + (\frac{\partial f}{\partial \overline{z}}\frac{\partial f}{\partial z})\frac{1}{i}h_2 + \text{H.O.T} \\ &= \frac{\partial f}{\partial z}(h_1 + ih_2) + \text{H.O.T} \\ &= \frac{\partial f}{\partial z}h + \text{H.O.T} \end{aligned} \tag{22.115}$$

$$\begin{aligned} f'(z) &= \lim_{h\to 0}\frac{1}{h}(\frac{\partial f}{\partial z}h + \text{H.O.T}) \\ &= \frac{\partial f}{\partial z}z \text{ exists} \end{aligned}$$

So therefore f is analytic

□

Example Is $\sin z$ analytic?

$$\frac{\partial}{\partial \overline{z}}\sin z = 0 \rightarrow \sin z \text{ is analytic} \tag{22.116}$$

In 'slow motion',

$$\begin{aligned}\frac{\partial \sin z}{\partial \overline{z}} &= \frac{1}{2}\left(\frac{\partial \sin z}{\partial x} + i\frac{\partial \sin z}{\partial y}\right)\\ \sin z &= \frac{1}{2i}\left(e^{iz} - e^{-iz}\right)\\ e^{iz} &= e^{ix-y} = e^{-y}(\cos x + i\sin y)\\ \frac{\partial e^{iz}}{\partial \overline{z}} &= \frac{1}{2}(\partial_x + i\partial_y)\\ &= \ldots = 0\end{aligned} \tag{22.117}$$

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 $\overline{z}$. Then, by the theorem we proved, if $\frac{\partial f}{\partial \overline{z}} = 0$, then f is analytic and $f'(z) = \frac{\partial f}{\partial z}$

Subsection 22.5

Power series

Definition 85

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 20

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

$$\lim_{n\to\infty} |a_n|^{1/n} \tag{22.120}$$

exists, then

$$\frac{1}{R} = \lim_{n\to\infty} |a_n|^{1/n} \tag{22.121}$$

Lemma 7

If $\lim |\frac{a_{n+1}}{a_n}|$ exists, then

$$\frac{1}{R} = \lim_{n\to\infty} |a_{n+1}/a_n| \tag{22.122}$$

Example

Find the radius of convergence for

$$\sum_{n=0}^{\infty} \sqrt{n} z^n \tag{22.123}$$

Proof

Let $z_0 = 0, a_n = \sqrt{n}$

$$\lim_{n\to\infty} \sqrt{n}^{\frac{1}{n}} = \lim_{n\to\infty} n^{1/2n}$$
$$\ln n^{1/2n} = \frac{1}{2n}\ln(n) \text{ which } \to 0 \text{ as } n \to \infty \tag{22.124}$$
$$n^{\frac{1}{2n}} \to e^0 = 1 \Rightarrow R = \frac{1}{1} = 1$$

□

Theorem 21

If $\sum a_n z^n$ and $\sum b_n z^n$ have radius of conversion of at least R

1. $$\sum a_n z^n + \sum b_n z^n = \sum (a_n + b_n) z^n \tag{22.125}$$
   and has radius of convergence of at least R
2. $$\sum a_n z^n \cdot \sum b_n z^n \tag{22.126}$$
   has radius of convergence equal to R
3. $$\sum a_n z^n \tag{22.127}$$
   is complex differentiable (and therefore analytic) in $|z| < R$ and it's derivative is
   $$\sum a_n n z^{n-1} \tag{22.128}$$
   with radius of convergence R
4. $$a_n = \frac{1}{n! f^n|_0} \tag{22.129}$$
   where
   $$f(z) = \sum_{n=0}^{\infty} a_n z^n \tag{22.130}$$

Subsection 22.6

Lecture 5

PART

VII

ECE444: Software Engineering

Taught by Prof. Shurui Zhou

Section 23

Preliminary

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 86 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.

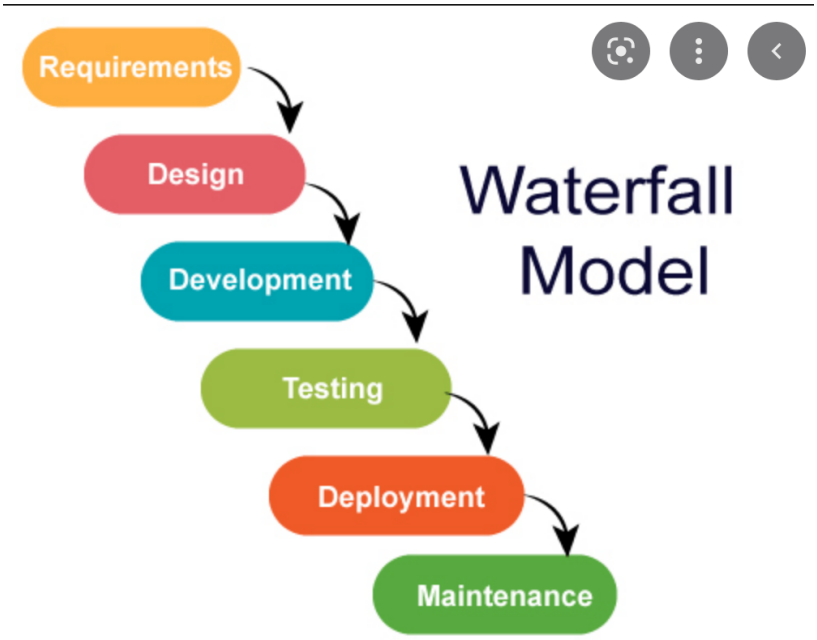


Figure 70. Waterfall method

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

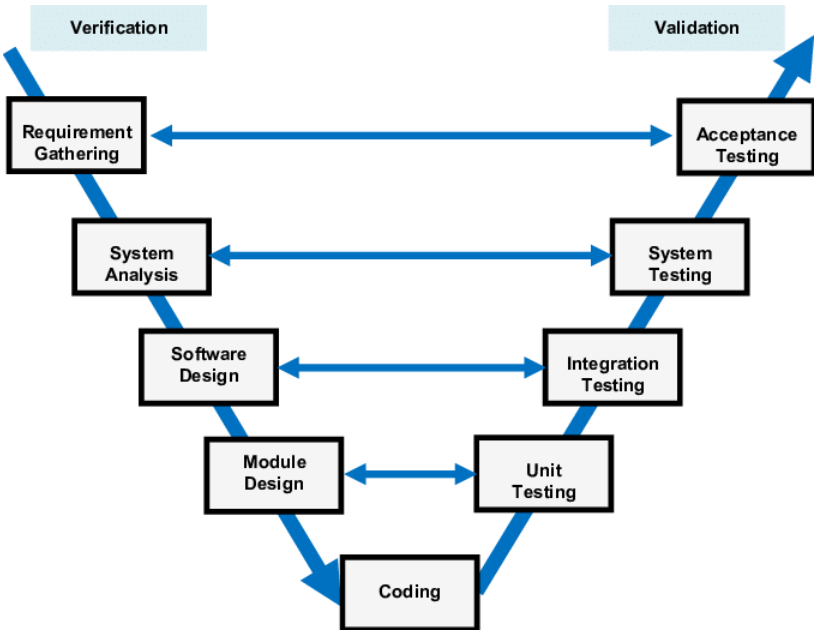


Figure 71. V model

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

ESC301: Seminar

What are notes?