Puvodni google doc: , https://docs.google.com/document/d/1Svj0KUPPj1kCWNNPgPTKkiNgOe66P3AwvptjFh8OJfM/edit#
Pokud nekdo chce ziskat high-level prehled o computer science vcetne historie, vyvoje pocitacu a zakladnich logickych obvodu a jak kde zapadaji ruzne predmety co mame za sebou, doporucuji crashcourse computer science (az se budete nudit po statnicih, ted neprokrastinujte!): https://www.youtube.com/watch?v=tpIctyqH29Q&list=PLH2l6uzC4UEW0s7-KewFLBC1D0l6XRfye

Číselné soustavy

big endian = číslo uloženo po **bajtech** od nejvyššího bajtu po nejnižší (zleva doprava)
little endian = je to naopak -- používá Intel, obecne jsem nabyl dojmu ze je vice common

nibble = 4 bity - odpovídá jedné cifře hexadecimální soustavy (0 - 9, A - F (15))

desitkova -> dvojkova soustava:
- delime dvojkou, piseme si vysledky modula, pak je prepiseme od konce a mame binarni cislo :)
- nebo jdeme od nejvyssich mocnin -- najdeme nejvyssi mocninu obsazenou v tom cisle, odecteme ji od cisla zapiseme 1,
    - pak vezmeme nizsi mocninu : mensi nez cislo? npiseme 1 a odecteme : napisme 0 a jdeme na nizsi mocniny

binarni -> desitkova:
- easy

Zaporna cela cisla v pocitaci (nevnimejte pocet bitu, neni dulezity ted :D):
- **dvojkovy doplnek** - zaporna cisla reprezentovana od 1111111… - zalozeno na tom ze kdyz budeme pod sebou odecitat 0 binarni od binarni jednicky, bude tam prenos a dostaneme prave 11111…
    - vyhodne pro pocitani
    - mame pak nejvyssi rad rezervovana pro zaporna cisla, tedy kazde cislo zacinajici 1 je zaporne
    - to odpovida rozdeleni rozsahu na dve poloviny - nejvyssi kladne cislo je pak 01111 -> kdyz pricteme neco dostaneme se do zapornych

- tedy kdyz reprezentujeme cisla pomoci n bitu, dostaneme rozsah cisel od <-2^(n-1), 2^(n - 1) - 1> -- kladnych cisel o jedno “min” protoze nula

Scitani cisel ve dvojkove soustave:

- aktualni bity: Ai, Bi
- **jednoducha scitacka pro jeden bit:**
  - vezme prenos z nizsiho radu Ci a aktualni bity scitanych cisel
  - vyplivne soucet bitu Si a prenos do vyssiho radu Ci + 1
- **ripple carry adder**
  - zretezenim jednoduchych scitacek muzeme vytvorit scitacku libovolne dlouhych cisel
  - jmenuje se
  - pomale -- jednotlive scitacky na sebe cekaji
- **carry look ahead adder** https://www.geeksforgeeks.org/carry-look-ahead-adder/

  - zalozena na analyze prenosu do vyssich radu
  - pro hodne bitu by byla komplikovana HW struktura, proto se pak taky nejak retezi ty pro min bitu
  - situace s prenosem (carry):
    - generovani prenosu
      - 1 + 1
      - bool variable: Gi = Ai and Bi (= Ai*Bi)
      - prenos vznikne bez ohledu na prenos nizsiho radu, tj ovlivnuje vysledek ale nikoliv dalsi prenos
    - propagace prenosu
      - 1 + 0, 0 + 1
      - bool variable: Pi = Ai xor Bi
      - prenos ovlivni vysledek a jede dal
    - vysledek Si = Pi xor Ci
      - pokud propagujeme a zaroven dostavame prenos, mame 0
      - pokud propagujeme ale nedostavame prenos mame 1
      - pokud nepropagujeme (0 + 0) a dostavame prenos mame 1
      - pokud nepropagujeme a nedostaveme prenos mame 0
    - prenos Ci+1 = Gi + PiCi (+ je boolean addition tedy or)
      - bud generujeme, nebo dostaneme prenos a navic prenasime dal
    - pak pro 4 bitovou look-ahead scitacku mame nasledujici vztahy:

The carry output Boolean function of each stage in a 4 stage carry look-ahead adder can be expressed as

$$
\begin{aligned}
C_1 &= G_0 + P_0 C_{in} \\
C_2 &= G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_{in} \\
C_3 &= G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{in} \\
C_4 &= G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{in}
\end{aligned}
$$

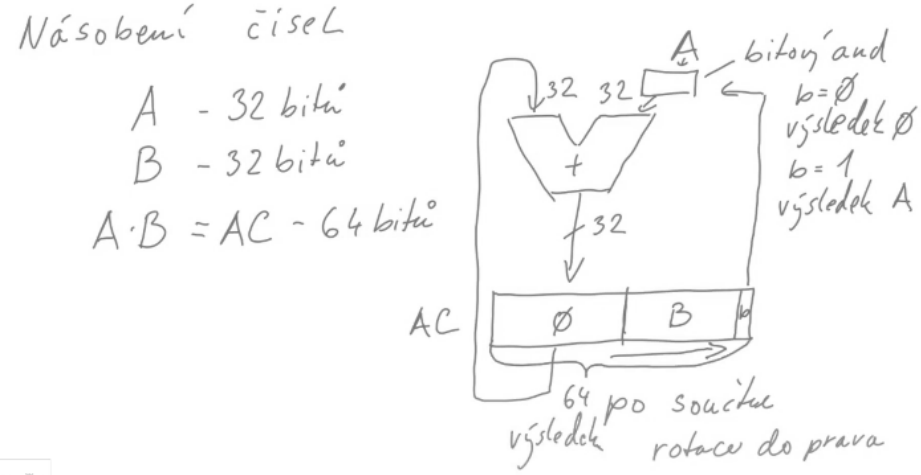
From the above Boolean equations we can observe that C_4 does not have to wait for C_3 and C_2 to propagate but actually C_4 is propagated at the same time as C_3 and C_2. Since the Boolean expression for each carry output is the sum of products so these can be implemented with one level of AND gates followed by an OR gate.

-

- **stromova scitacka**
    - slozitost 2logn -- nebudu zabredavat :D -- ale v podstate se ty zavislostui nejak skladaji do stromu nebo tak neco

Nasobeni celych cisel

- podobne jako nasobeni pod sebou
- pokud nasobime A a B -- 32 bitu
- nasobime do akumulatoru ktery ma 64 bitu
- inicializace: akumulator jsou nuly | B
    - podle posledniho bitu (tedy na zacatku posledniho bitu B) provedeme and s ackem (to nam jej bud vynuluje nebo zkopiruje -- v nasobeni odpovida Bi*A)
    - pricteme vysledek and k leve casti akumulatoru a zapiseme
    - provedeme rotaci akumulatoru doprava
    - opakujeme
- po 32 rotacich mame vysledek ulozeny v akumulatoru
- je to pomaly alg, ve skutecnosti se to pry dela jinak :D
- obrazek niz -- je z toho videa prevedl jsem do cernobile aby slo tisknout :)



Realna (desetinna) cisla

Realne cislo s konecnym rozvojem se neda vzdy zapsat binarnim s konecnym rozvojem ale naopak se da vzdy!

Standart IEEE 754 -- 32 bitovy float

- 1 bit **znamenko**, 8 bitu **exponent** a 23 bitu **mantissy** (hodnoty cisla)
- normalizovana cisla:
    - exponent je zvetsen o 127
    - mantissa obsahuje skrytou 1 (protoze vetsina cisel jsou 1.neco)
- nestandardizovana cisla:
    - exponent 00000:
        - cisla mensi nez e^-127 a 0
    - exponent 255
        - nekonecno -- mantissa musi byt 0
        - jinak je to NaN
- prevod do IEEE 754:
    - exponent:
        - pokud cislo je mensi nez 1:
            - nasobime dvema dokud nedostaneme cislo vetsi nez 1 -> pak exponent je e^-(pocet nasobeni)
        - jinak:
            - to same akorat delime :)
        - nebo udelame exp = floor(log2(cislo)) :D
    - mantissa:
        - porovnavame mocniny dvojky od -1 … -n s cislem co nam zbyde po predchozim alg.
        - pokud je cislo mensi -> 0
        - pokud je vetsi nez mocnina -> zapiseme 1 a od cisla odecteme mocninu

Scitani a odcitani:

- prevedeme na stejny exponent a secteme
- kdyztak musime upravit exponent a tedy i mantissu (aby carka byla na spravnem miste)

Základní architektura pocitace

CPU je spojen sbernici s RAM (= random access memory = muzeme cist odkudkoliv) sbernice:

- adresni cast - s kterou casti pameti pracujeme
- datovou -- prenasi data
- ridici - co delame - cteme nebo zapisujeme

Harvard architektura RAM - rozdelena na instrukce a data

Von Neumann - nerozlisuje instrukce a data v pameti
- dnes rozsirena

CPU
registry = rychla pamet v CPU
dulezity registr: PC (program counter) nebo se mu taky rika IP (instruction pointer) = ukazuje kde v programu se nachazime (ukazuje kde v pameti jsme)

ALU (Arithmetic logic unit)
- dela zakladni aritmeticke a logicke operace - kombinace ruznych obvodu a gatu
- jeden ze zakladnich stavebnich bloku CPU

Vykonavani instrukce:
1. fetch - nacte instrukci podle IP do registru, IP++
2. decode - typ instrukce a parametry
3. execute - vykonani -- vysledek se uklada do pameti nebo do registru

=> pro pipelining se pak da jeste rozdelit na vice fazi, zvlaste execute na ALU operaci a zapis do pameti
Instrukce se vykonavaji dokud neprijde interrupt (preruseni):
1. IP se ulozi do nejakeho registru pro navrat z preruseni
2. IP se nastavi podle preruseni

Take muze dojit k vyjimce pri vykonavani (deleni nulou) ci nacitani instrukce (spatna cast pameti)

Procesory podle typu instrukci:
CISC
- intel
- ruzne dlouhe instrukce

RISC
- MIPS, ARM, RISC-V
- 32 bitu = jedna instrukce
- vyhoda - instrukce stejne dlouhe a IP tedy trivialni implementace, povolene jen skoky po 4 bajtech
- nevyhoda: nevejde se tam 64 bitove cislo -- musi se provest dvema instrukcemi

MIPS instrukce:

Instrukce	Syntax	Operace	Význam
Add	add $d, $s, $t	$d = $s + $t;	Add: Sečte dva registry $s + $t a výsledek uloží do registru $d
Addi	addi $t, $s, C	$t = $s + C;	Add immediate: Sečte hodnotu v $s a znaménkově rozšířenou přímou hodnotu, a výsledek uloží do $t
Sub	sub $d,$s,$t	$d = $s - $t	Subtract: Odečte znaménkově obsah registru $t od $s a výsledek uloží do $d
Bne	bne $s, $t, offset	if $s != $t go to PC+4+4*offset; else go to PC+4	Branch on not equal: Skáče pokud si registry $s a $t nejsou rovny
Beq	beq $s, $t, offset	if $s == $t go to PC+4+4*offset; else go to PC+4	Branch on equal: Skáče pokud si registry $s a $t jsou rovny
slt	slt $d,$s,$t	$d = ($s < $t)	Set on less than: Nastavi registr $d, pokud plati podminka $s < $t
jump	j C	PC = (PC ∧ 0xf0000000) ∨ 4*C	Jump: Skáče bezpodmíněčně na návěstí C
lw	lw $t,C($s)	$t = Memory[$s + C]	Load word: Načte slovo z paměti a uloží jej do registru $t
sw	sw $t,C($s)	Memory[$s + C] = $t	Store word: Uloží obsah registru $t do paměti
lui	lui $t,C	$t = C « 16	Load upper immediate: Uloží předanou přímou hodnotu C do horní části registru. Registr je 32-bitový, C je 16-bitová.
la	la $at, LabelAddr	lui $at, LabelAddr[31:16]; ori $at,$at, LabelAddr[15:0]	Load Address: 32-bitové návěstí uloží do registru $at. Jedná se o pseudoinstrukci - tzn. při překladu se rozloží na dílčí instrukce.

CACHE

Potrebujeme protoze procesor jede treba na 0.3 nsec, ale RAM ma prumerne zpozdeni 70 - 120 nsec

L1, L2, L3 cache -- zleva doprava se zvetsuje jejich pamet (32kb, 256 kb, 8 mb) a zaroven se zmensuje jejich rychlost (1.2, 4, 12 nsec)
pozn. cena L1 cache je million korun za GB :D (proto je tak mala)

Cache:

- sklada se z **bloku dat - 4, 8, 16 slov (slovo = 4 bajty)**
- pamet se nacita z oblasti v pameti o velikosti pocet slov * 4 bajtu, ktera zacina na adrese delitelne pocet slov * 4
- pak zaznam v cache:
    - tag -- prvni 4 bity, ktere jsou pro vsechna slova stejna
    - blok dat
    - priznakove bity
        - valid 1/0
- cache hit - pokud se v cache nachazi tag ktery se shoduje se zacatkem adresy co hledame, muzeme si z cache vytahnout dotycny bajt / skupinu bajtu
- miss - neni v cache, hledame v pameti
    - pak neco vyhodime (podle ruznych kriterii) a nahradime

Plne asociativni cache = data muzeme dat kam chceme do cache, cely tag se musi porovnat se vsemi tagy v cache -- hodne komparatoru a narocna hardware implementace

jednocestna cache = data musime ulozit do casti cache (mnoziny) ktera odpovida jejich prefixu a tedy i kdyz je jinak prazdna, musime treba prepsat uzitecnou hodnotu

dvoucestna cache = mnoziny jsou zdvojene -- muzeme si vybrat tu prazdnou

vyhoda “cestnych” caches oproti plne asociativni je mensi pocet kompratoru

- kdyz nenajdeme neco v L1 cache hledame v L2, pak v L3 a pak kdyztak v RAM
- L3 cache byva spolecna pro jadra, musi se tedy synchronizovat!

Zapis do cache:
- write through - nezapiseme rovnou do cache ale cekame na zapis o uroven vyse
- write back - napisi se data pouze do cache a nastavi se dirty bit, ze jsme data modifikovali => data se pak paralelne s praci procesoru kopiruji dal => muze nastat problem s konzistenci

Strankovani

k cemu? -> aby kazdy proces mohl mit svuj vlastni virtualni svet (pamet)
Mame tabulky pro kazdy proces co znaci, kde co je v realne RAM
-- vetsinou vicevrstve.
Vic detailu asi v OSY nebo jinych materialech, nechce se mi to vypisovat :D

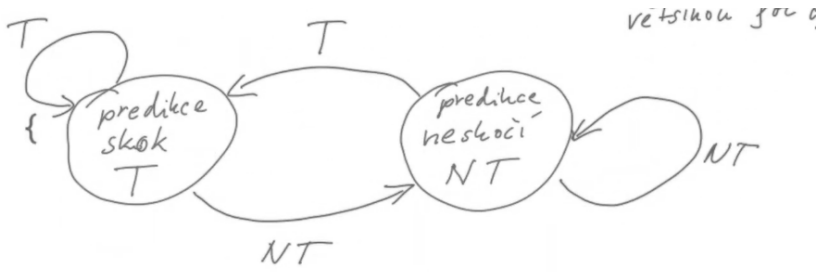
PIPELINING

Rozdelime vykonavani instrukci na vice fazi: fetch, decode, execute, memory access (instrukce load/store - cteni a zapis do pameti) a write back (zapis vysledku do registru, load) -> pak muzeme instrukce paralelizovat -> zacneme fetch instrukce, pak delame decode ale mezitim uz delame fetch dalsi instrukce

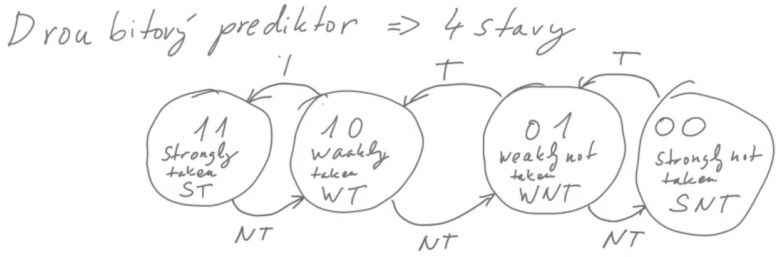
Muze nastat problem se synchronizaci - npr stara hodnota v registru - **hazard**
- resime:
    - pozastavenim vykonavani dane instrukce dokud nema aktualni data
    - forwarding - z mem nebo write back preposleme rovnou do execute

Predikce skoku

Skok muze predstavovat problem, protoze procesory (zvalste ty slozitejsi) si dopredu nacitaji velke mnozstvi instrukci -> muzeme predikovat, jestli instrukce povede ke skoku nebo ne
T = taken (skocili jsme)
NT = not taken (neskocili jsme)
Jednobitovy prediktor:
- automat
- kdyz jsme skocili, predikujeme skok, kdyz jsme neskocili predikujeme ze neskocime :D
- nefunguje spatne, protoze ve for cyklu mnohokrat za sebou skaceme



Dvoubitovy prediktor:



“dele trva jej presvedcit”

Moderni CPU vyuzivaji pro predikci **perceptron**:
vstupem je registr s historii poslednich n skoku
Prediktoru **muze byt vic a muzeme se rozhodovat ktery pouzit**

Pripojeni periferii I/O

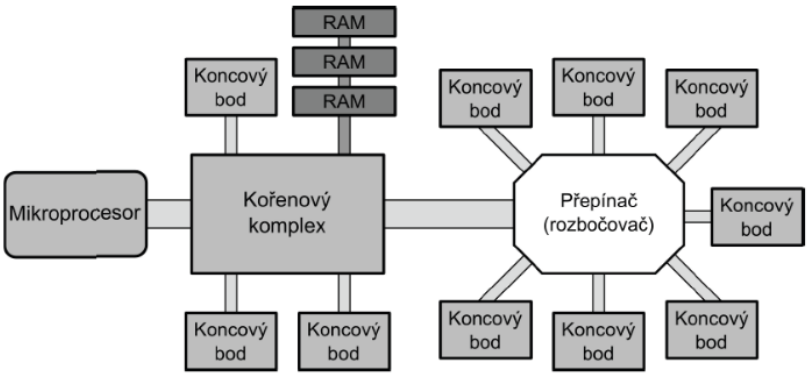
pro pripojeni periferiim:

- intel -- instrukce in out (spis zpetna kompaibilita) + **sdileni pameti**
- risc -- jen **sdileni pameti (DMA** - direct memory access)

central address decoder - vi, ktera cast pameti RAM je pro periferie, zna cely system
X
kazda periferie ma svuj decoder - zna svou cast pisecku :D
-> napr LED se rozsviti podle toho co si precte ve svoji casti pameti

- Ruzne sbernice (aka draty k periferiim, menenim napeti vysilaji signal):
  - asynchronni - bez hodin + seriove - prenasime bit po bitu
    - napr SATA -- pripojeni HDD
    - PCI-e (express)
  - paralelni:
    - vice dratu paralelne
    - oproti seriovym umi fungovat v nizsich frekvencich protoze se musi poresit synchronizace dat mezi jednotlivymi draty
    - pr PCI - komunikace pouze jednim smerem (half duplex) - povel - odpoved
      - na jedne sbernici muze byt vice zarizeni s jinou adresou
  - PCI-e (express) - seriova full duplex (vysila i prijima zaroven) - vodic navic

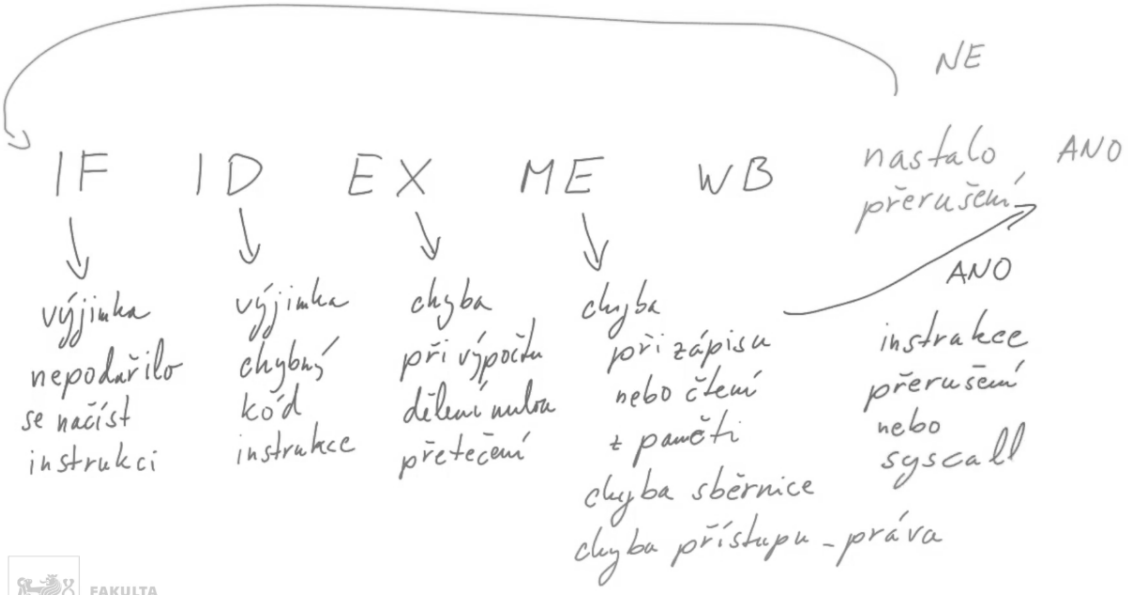
- oproti PCI daleko rychlejsi
- point - to - point
    - na CPU je napojen switch ktery komunikuje s periferiemi
- na obrazku:



-

Vyjimky a preruseni

Jake mohou vzniknout:



Komunikace s periferii:

- CPU udela preruseni, specifikuje co chce od periferie
- periferie pracuje
- pak da prerusenim vedet CPU ze je done

Zpracovani preruseni:

- **vnejsi** preruseni
    - pri zpracovani nesmi byt naruseny uzivatelske registry, pro program je neviditelne

    - asynchronni
    - IP (PC) uzivatelskeho programu je zkopirovan do jineho registru - na konci je zpet nastaven
    - pri pipeliningu je dokoncena pipeline
- **vyjimky**
    - konec programu - nemuzeme dal pokracovat nebylo by korektni
    - chyba operacniho systemu - modra smrt
    - zpracovani je synchronni

Predavani parametru funkcim v C

Pouziva se **zasobnik** (stack) - jak jinak :D
implementace zasobniku:

- push: vezme registr esp, mensi o 4 a pak ulozi pozadovanou hodnotu na adresu na kterou ukazuje esp
- pop: zvetsime esp o 4 a precteme hodnotu byla v pameti kam ukazova esp

Volani funkce:

- skocime do funkce pomoci specialni instrukce (zapamatuje se na zasobnik kde jsme byli predtim)
- pak se ve funkci nejprve zvetsi zasobnik o potrebne misto
- parametry pri volani funkce se ukladaji na zasobnik ci u mipsu prvni 4 se ukladaji do registru
- navratova hodnota se da do registru rax (v mipsu aspon)

Procesory x86

prehled procesoru a jejich historie - asi zbytecne si pamatovat :D
prehled instrukci intelu --taky si nebudu pamatovat

Preklad C

1. **preprocessing** - textove nahrazeni maker (mohou bt parametrizovane), includy se nakopiruji
2. **lexikalni analyza**
    a. text je rozdelen na lexy -- zakladni stavebni kameny -- cisla, konstanty, symboly
    b. pravidla jsou vlastne regularni vyrazy -- postupujeme po rade po jednotlivych regularnich vyrazech -- od special symbolu po promenne atd
3. **syntakticka analyza**
    a. zalozena na pravidlech bezkontextove gramatiky
    b. pomoci prepisovacich pravidel se da dobre zadefinovat napr priorita operaci
    c. stvori odvozovaci strom

4. **semanticka analyza**
    a. vytvori mezikod -- neco jako bytecode, multiplatformni “assembler” instrukce
        i. pr. ocislovane trojice arg1 op arg2
    b. vyuziva stomu vytvoreneho syntaktickou analyzou -- prochazi strom rekurzivne, kdyz dojede nad listy, muze uz udelat prvni jednoduche prikazy, z nich hodnoty pak muze skladat do dalsich atd
5. **generovani kodu -- prevod mezikodu na assembler**
    a. rozhodneme, co se da do jakych registru a jak se dana operace mezikodu mapuje na instrukci konkretniho assembleru
    b. optimalizace:
        i. primo na trojicich mezikodu - hledani zavislosti, opakujici se trojice atd
        ii. pri prevodu na assembler
    c. dostaneme .o soubor
6. **linkovani**
    a. nezname symboly se musi sestavit
    b. hledaji se zavislosti, symboly ktere musime importovat
    c. **static linking** - vsechno se hodi za sebe, prepocitaji se adresy aby odpovidaly sestavenemu programu
        i. program muze byt velky, knihovna se kompiluje nekolikrat
        ii. nepotrebuje zadne vnejsi zavislosti ke svemu spusteni (uzivatel nemusi instalovat zadne knihovny)
    d. dynamic linking
        i. knihovny nejsou pridany do knihoven
        ii. nevyhoda -- musi byt nainstalovana knihovna
        iii. vytvori se stub u volani funkci z dynamicke knihovny
            1. pri prvnim volani se zjisti kde je knihovna, nacte se a zavede se spravna adresa jejich funkci