**Programozás, algoritmusok, adatstruktúrák**

**1. Rekurzió.**

* Rekurziónak azt nevezzük, amikor egy függvény vagy eljárás önmagát hívja meg. Rekurzió használatával nagyon sok problémát elegánsan és röviden oldhatunk meg, azonban nem mindig a leghatékonyabb megközelítés
* leggyakrabban a backtrack típusú algoritmusokban szokták használni
* minden rekurzió ciklussá alakítható, illetve minden ciklus rekurzióvá
* előfordulthat, hogy a hívás túl sokszor fordul elő, ez végtelen rekurziót eredményez, ennek jellegzetessége a verem túlcsordulása, másnéven *stack overflow*
* Veges szamu lepes utan meg kell allitani.
* A rekurziv fuggvenyhivas eroforras igenyes, mivel minden hivasnal helyet kell foglalni a parametereknek es a fuggvenyeknek a veremben.

**2. Vermek, sorok.**

A verem olyan homogén, szekvenciális adatszerkezet, amely esetében az adatszerkezetben tárolt elemek sorozatához csak az egyik végénél férhetünk hozzá. Bővíthetjük a vermet egy újabb elemmel, ekkor azt mondjuk, hogy adatot helyezünk a verem tetejére, vagy kiolvashatjuk a verem tetején lévő elemet. Ez tehát azt jelenti, hogy leghamarabb az adatszerkezetben legutoljára elhelyezett elemhez férhetünk hozzá, illetve a benne legelőször elhelyezett adatot olvashatjuk ki legutoljára (LIFO: Last In First Out).

* a **verem** egy LIFO típusú struktúra, mely véges számú és azonos típusú adatot tárol
* a veremre értelmezett műveletek: **push, pop, isFull, isEmpty**
* szokásos megvalósítása egy véges méretű, összefüggő memóriaterület és egy veremmutató segítségével történik
* mindig az egyi végéről töltjük fel, míg a veremmutató a legfelső elemre mutat
* Használat: pl Undo művelet, backtracking feladatok, memória kezelés

A sor szintén homogén és szekvenciális, de esetében a bővítés az elemek sorozatának egyik, míg az olvasás a másik végén lehetséges. Az adatszerkezetbe elhelyezhetünk új elemet az egyik végén, de elem kiolvasására csak az adatszerkezet másik végén van lehetőség. Ez azt eredményezi, hogy az adatszerkezetbe legelőször elhelyezett elem olvasható ki leghamarabb, és a legutoljára beírt adathoz férhetünk hozzá legutoljára (FIFO: First In First Out). Használat: pl CPU ütemezés, aszinkron adat átvitel

* a **sor** egy FIFO típusú struktúra, melynél mindig a legelsőként elhelyezett elemet vehetjük ki legelőször, így nem érhető el minden eleme közvetlenül
* a soron értelmezett műveletek: **put, top, get, isEmpty**

**3. Láncolt listák, körkörösen láncolt listák, fák.**

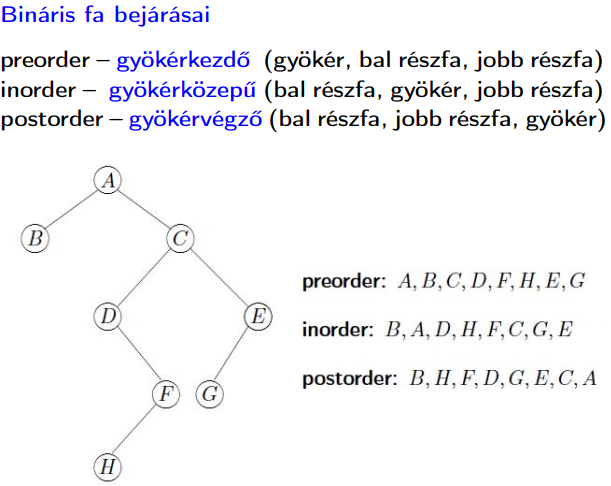
A láncolt lista olyan adatszerkezet, amelyben az objektumok lineáris sorrendben követik egymást. A tömböknél a lineáris sorrendet a tömbindexek határozzák meg, ezzel szemben a láncolt listákban mutatók valósítják meg az elemek lineáris elrendezettségét: a lista minden objektuma tartalmaz egy mutatót, amely a következő objektumra mutat.

A ciklikus listát az jellemzi, hogy a fej előző mutatója a lista végére, a vége köv pointere pedig a fejre mutat. A ciklikus listát ezért úgy is tekinthetjük, mint egy elemekből alkotott gyűrűt.

A fa minden csúcsát egy objektummal ábrázoljuk. Akárcsak a listáknál, itt is feltételezzük azt, hogy minden csúcs tartalmaz egy kulcs mezőt. A további (érdeklődésünkre számot tartó) mezők, amelyek más csúcsokra mutató pointereket tartalmaznak, a fa típusától függően változnak.

* a **fák** körmentes, összefüggő gráfok
* a *gyökérelem* a fa azon eleme, melynek nincs őse
* a *levél* a fa azon eleme, melynek nincs utóda
* a *közbenső elemek* a fa nem gyökér- illetve levélelemei
* az *út* a gyökérből kiinduló
* Ha n csomopont van -> az elek szama n-1
* Egy csomopont melysege -> a gyokertol valo utvonal hossza
* Magassag -> a leghosszabb utvonal a gyokertol, vagy egy adott csomoponttol, a levelig

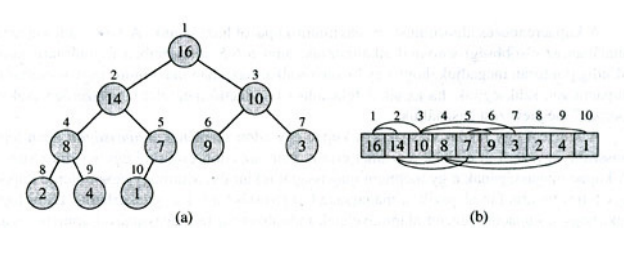
**4. Bináris fák és bináris keresőfák.**

* a **bináris fa** olyan speciális fa, mely esetén minden csúcs legfeljebb két gyerekkel rendelkezik
* A bináris fák az egyirányba láncolt lista általánosításai abban az értelemben, hogy egy helyett kettő rákövetkezője van minden faelemnek. Ezeket a rákövetkezőket bal- és jobboldali rákövetkezőknek hívjuk. Az általános bináris fa felépítését azonban nem végezhetjük olyan általánosan, mint tettük azt a láncolt listák esetén, azaz nem tudunk olyan függvényt/eljárást írni, amely egy adott helyen bővíti a fát, a fának ugyanis nem egy, vagy két vége van, hanem ahogy nő a fa, úgy nő azoknak a helyeknek a száma, amelyet a fa "végének" nevezhetnénk. A fa bővítése ezért csak akkor oldható meg automatikusan, ha a bővítés során bevezetünk valamilyen szabályt, amely szabály aztán pontosan meghatározza, hogy egy adott elemet hová kell beszúrni a fában
* ****
* egy speciális esete a **kiegyensúlyozott bináris fa**, mely esetén minden csúcsnak pontosan 0 vagy 2 gyereke lehet, illetve bármely csúcspont két részfájának magassága közti különbség legfeljebb egy
* a **keresőfa** olyan fa, melynek bármely elemére igaz, hogy az elem baloldali részfájában az összes kulcs kissebb, a jobboldali részfájában az összes elem kulcsa nagyobb mint az adott csúcs kulcsa
* keresőfa tulajdonságai: • minden csúcsához tartozik egy kulcs (különböző csúcsokhoz különböző kulcsok)   
   • minden csúcs esetében a bal gyerek kulcsa kisebb, a jobb gyerek kulcsa nagyobb, mint a csúcsé
* Inorder bejárás: növekvő sorba rendezve adja meg a kulcsokat.

**5. Kupacok (heap), binomiális fák.**

**-** (bináris) **kupac** majdnem teljes bináris fa egy tömbben ábrázolva

- A majdnem teljes bináris fa azt jelenti, hogy csak a legalsó szinten hiányozhatnak elemek, és azok is csak egy adott elemtől kezdődően. Az elemeket egydimenziós tömbben tároljuk, szinteken haladva balról jobbra. Például itt a fában a csúcsok mellett az illető elem sorszáma szerepel a tömbben. A tömb első eleme 1- es indexű.

****

Az elemeket az A tömbben őrizzük: A1, A2, . . . , An, ahol

- a kupacméret[A] ≤ n a kupac maximális elemszáma   
- A1 a gyökér.   
- Ai szülőjének az indexe: Szülő(i) = i/2   
- Ai bal gyerekének az indexe: Bal(i) = 2i   
- Ai jobb gyerekének az indexe: Jobb(i) = 2i + 1  
- kupactulajdonsag: ha **B** csúcs fia az **A** csúcsnak, akkor kulcs(A) >= kulcs(B), ebben az esetben a kupacot *max-kupacnak* nevezzük, ha az egyenloseget megforditjuk, *min-kupac*.  
- **alkalmzasok**: kupacrendezes, grafalgoritmusokban sokszor gyorsitja az algoritmust.

A binomialis fa

**6. Rendezési algoritmusok.**

**-** egy olyan algoritmus, mely valamilyen szempont alapján sorba állítja elemek egy listáját **-** rendezett adathalmazzal több és hatékonyabb műveletek végezhetőek, mint rendezetlennel **-** megkülönböztetünk belső és külső rendezést **-** belső rendezések: *Bubble sort, Selection sort (kivalaszto), Insertion sort (beszurasos), Merge sort (osszefesulo), Heapsort (kupacrendezes), Quicksort (gyorsrendezes)***-** belső rendezések esetén a rendezés az elemek összehasonlítására alapszik, míg küldő rendezés esetén az algoritmusok nem használják az elemek között fennálló hasonlítást  
- kulso rendezes akkor all fent, amikor az adathalmaz tul nagy ahhoz, hogy beferjen a memoriaba, ezert kissebb darabokra osztjak oket, amelyeket rendeznek, majd a vegen ossze merge-elik

**7. Hasító (hash) táblák és hasító algoritmusok.**

**-** a hash-table egy olyan adatszerkezet, amely egy hash-függvény segtségével rendel minden értékhez egy kulcsot, mellyel későb (ideális esetben) konstans időben azonosítja az értéket, ezáltal implementálva egy asszociatív tömböt **-** egy leképzést végez két halmaz között **-** ideális esetben minden kulcs egyedi, valóságban történhetnek ütközések, így számolni kell a fellépő ütközéssel is, utkozes eseten, egy lancolt listaban taroljak a tobb adatot **-** sok esetben hatékonyabbak mint a keresőfák, ezért gyakran használják asszociatív tömbök implementációjában, adatbázisok indexelésében, illetve cache memória felépítésében **-** a **hasítófüggvény** egy speciális eljárás a hasítótáblák felépítésére **-** nagy méretű adatállományok adatelemeinek gyors, hatékony megkeresésére használják **-** adott K érték esetén kiszámolunk egy f(K) függvényt, amely meghatározza K érték heléyt az adatállományban **-** szükséges továbbá az azonos hasítóértékekkel rendelkező elemek megkülönböztetése

**8. Algoritmusok bonyolultsága.**

Az algoritmusok bonyolultsaga azt meri, hogy egy adott bemeneti adatmennyisegre milyen futasidovel vagy tarhely igennyel rendelkezik egy algoritmus illetve ennek skalazasaval hogyan valtozik ez az ertek.

O(1) – konstans – Konstans szamu lepesbol all az algoritmus, fuggetlenul a bemenet meretetol

O(log(n)) – logaritmikus – altalaban 2-es alapot haszalnak, de nem mindig’

O(n) – linearis – szinte ugyanannyi lepesbol all az algoritmus, ahany bemeneti elem van

O(n^2) – negyzetes – pl bubble sort, selection sort

O(2^n) – exponencialis – pl backtracking algoritmusok

O(n!) – faktorialis – permutaciok

Best Case (pl. elsore megkapjuk a szamot a tombben), Worst Case (ultosonak kapjuk meg), Average case (kb felenel kapjuk meg)

**OOP**

1. **Osztályok és objektumok. Az osztály tagjai (classes objects, members)**

* az osztály egy felhasználó által létrehozott típus
* az osztály tagjai az **attribútumok (adat)** és **metódusok (viselkedés)**
* az objektumok az osztályok példányai, van állapota, viselkedése és futásidőben azonosítható
* az adattagok hozzáférése esetén megkülönböztetünk *privát, publikus* és *protected* jogosultásgot
* a privát csak az osztályon belül érhető el, a publikus bárhonnan, a protected pedig az osztályban és annak leszármazottjaiban

1. **Statikus tagok (static members)**

* statikus tagok az OOP esetén nem objektum szintű adattagok, hanem osztályszintűek
* statikus tag = statikus adat + hozza tartozó statikus metódus
* Az adatok az osztály létrejöttekor lesznek létrehozva, nem szükséges példányosítani
* A példány szintű függvények használhatják a statikus adatokat mert azok???
* Instance methods may use static data. Why? – mivel a statikus adatok osztaly szintuek, ezert a peldanyok tudjak hasznalni azt
* Static methods cannot use instance data. Why? – mert a statikus metodusok csak statikus adatokkal dolgozhatnak
* a függvények is lehetnek statikusak, nem használhatják az objektumok adattagjaihoz
* statikus metódusok nem lehetnek *const* típusúak sem? A statikus metodusnak nincs this -je es a const az a this-re van hatassa. A const fuggvenyek nem valtoztathatnak az adattagokon.
* nem objektumra hívódnak meg hanem az egesz osztalyra 🡪 nem rendelkeznek *this* pointerrel
* az adattagok a futtatás teljes idejében elérhetőek, c++ esetén deklarálni a header-ben kell, inicializálni a .cpp fájlban

1. **Egységbezárás, az információ elrejtése (encapsulation, information hiding)**

* az egységbezárás adatok és a metódusok osztályba való öszezárását jelenti
* az objektum egységbezárja az állapotot a viselkedésmóddal 🡪 az objektum állapotát csak a metódusokok keresztül érhetjük el vagy módosíthatjuk, ez pedig a biztonságot szolgálja
* elrejtjuk az osztaly implementacijat es annak reszleteit
* A felhasznalo egy a szamara elerheto metodusokkal dolgozik, ugymond egy interface-en keresztul
* Sokkal karbantarthatobb lesz igy a kod
* az **információ elrejtése** alatt az objektum elrejti az adatait és bizonyos műveleteit ezzel elkerulve az illetektelen hozzaferest
* nem tudjuk az adatok tárolásának módját, illetve a műveletek implementálását sem
* A kulonbseg az egysegbezaras es az informacio elrejtese kozott hogy az egysegbezaras inkabb a rendszer komplexitasat hivatott elrejteni, mig az informacio elrejtese inkabb a bizotonsagra fokuszal

1. **Származtatás, örökítés (inheritance)**

* a származtatás olyan osztályok között értelmezett viszony, amely segítségével egy általánosabb típusból (ősosztály) egy sajátosabb típust tudunk létrehozni
* uj adat hozzaadasa, uj funkcionalitas hozzaadasa, metodusok felulirasa/modositasa pl print()
* az utódosztály adatokat és műveleteket örököl, kiegészíti ezeket saját adatokkal és műveletekkel, illetve felülírhat bizonyos műveleteket
* Orokiteskor a letrejovesi sorrend: Alap osztaly constructor -> orokitett osztaly constructor. Felszamolaskor forditott sorrend
* megkülönböztethető egyszeres és többszörös örökítés
* publikus, privat, protected orokites  
  public -> az ososztaly publikus es protected adattagjai ugyanazok lesznek  
  protected -> az ososztaly publikus es protected adattagjai protected-ek lesznek  
  private -> az ososztaly publikus es protected adattagjai privat-ok lesznek
* protected es publikus adattagok elerhetoek az orokitett osztalyok szamara, a privat nem

1. **Polimorfizmus. Metódusok túlterhelése és felülírása (polymorphism, overloading, overriding)**

* **polimorfizmus:** többalakúság 🡪 egy ősosztály típusú változó hivatkozhat ugyanazon közös osztályból származó osztályok példányaira
* metodusok tulterhelese – ugyanaz a nev, mas szignatura
* metodusok felulirasa – ugyanaz a nev, ugyanaz a szignatura
* lehet statikus és dinamikus:
  + *statikus*: metódusok túlterhelése, sablonok 🡪 statikus, fordításidejű kötés
  + *dinamikus*: metódusok felülírása 🡪 dinamikus, futásidejű kötés

1. **Interfészek és absztrakt osztályok (interfaces and abstract classes)**

* absztakt osztályoknak nevezünk azon osztályokat, melyeknek van legalább egy absztrakt művelete, amit majd szarmaztatott osztaly implemental vagy felulir
* interface -> felületet határoz meg és nem lehet példányosítani, a műveletek implementálását az utódosztályokra bízza, csak deklaralja a muveleteket, mindent a leszarmazottak implementalnak
* c++ esetén az absztrakt műveltet a *virtual* kulcsszó segítségével kell bekapcsolni, ez engedélyezi a későbbi felülírás lehetőségét is, implementációt az adott osztályban is tartalmaz
* c++ esetén nem léteznek úgynevezet interfészek, csupán tiszta virtuális függvényekből álló osztály, ebben az esetben nem tartozik a deklarációhoz semmiféle implementálás, az utódosztályokra van bízva: ***virtual*** *T <fun>()* ***= 0;***

1. **Kivételek és feltételek (exceptions and assertions)**

* a kivételek egy hibakezelési mechanizmust biztosítanak
* olyan hibák kezelhetőek általuk, melyekre számítani lehet
* a visszatérítési értékkel ellentétben nem biztonságosabb, mivel a hívónak muszáj kezelni, kihagyhat hívásokat a stackről illetve további információkat is tárolhat
* megtalálható néhány előre megírt kivétel, viszont mi is létrehozhatunk újakat, csupán az *exception* osztályból kell származtatni
* a feltétel vagy **assertion** egy régebbi típusú hibakezelési lehetőség
* a paraméterét kielemzi, ha ennek az értéke *false*, egy üzenetet ír ki az *std:cerr*-re, majd *abort*-ot hív, mely megszakítja a program futását
* az assert nem minden esetben fut le, mivel szorosan kötött az NDEBUG makróhoz
* ha a makró nincs definiálva, a fordító az assert üzeneteket kihagyja a fordításból, nem lesz hatása a futásra

**Operacios Rendszerek**

1. **Operációs rendszerek absztrakciós szintjei**

* az absztrakciós szintek segítségével a hardverek fölé olyan rendszert helyezünk, amivel a programozó egyszerűbben tudja kezelni az erőforrásokat
  + **||Felhasználói programok**
  + **|| Fordító | Szövegszerkesztő | Parancsértelmező || Rendszerprogramok**
  + **|| Operációs rendszer || Rendszerprogramok**
  + **|| Gépi nyelv || Hardver**
  + **|| Mikroarchitektúra || Hardver**
  + **|| Fizikai eszköz || Hardver**

1. **A rendszermag(kernel) és a héj(shell) szerepe valamint fontosabb feladataik**

* a kernel felelős a hardver erőforrások kezeléséért, a processzoridő szétosztása
* megkülönböztetünk: - *monolitikus (egy prog. allo rendszermag, bovites vagy modositas csak a teljes mag ujraforditasaval lehetsege, UNIX, Linux)  
   - mikro- (interfeszeken komm. a programok, csaka folyamatok betoltese es azok kozti kommunikacioert felelos, minden mas folyamatot kulonallo rendszerfolyamat vegez)  
   - hibrid (kritikus funkciokat egyetlen egysegbe otvozi, masokat kulonallo modulkent kezel, pl Windows)   
   - exokernelt (felhasznalora van bizva a hardver absztrakcio)*
* a kernel fő feladatai:
  + ki- és bemeneti eszközök kezelése
  + Programok, folyamatok futásának kezelése
    - Indítás, futási feltételek biztosítása, leállítása
    - Memória-hozzáférés biztosítása
    - Processzor idejének elosztása
  + Háttértárolók kezelése
  + Rendszerhívások kezelése
  + Fájlrendszer
* A shell fuggetlen az operacios rendszer magjatol, egy interface-t nyujt a felhasznalo es a kernel szolgaltatasai, funkcioi kozott, vagyis maga a szamitogep hasznalatat teszi lehetove
* Shell feladatai:
  + kapcsolattartás a felhasználóval
  + alkalmazások futásának kezelése

1. **A processzus fogalma és a processzus állapotainak ismertetése**

* a folyamat az operációs rendszerben futó program példánya / egy vegrehajtas alatt levo program
* minden folyamat szamara az Op. Rendszer el kell tarolja a folyamat adatait, eza a PCB(Process Control Block) feladata, ez olyan informaciokat tarol, ami szukseges ahhoz, hogy leallitas utan kesobb ugy indulhasson ujra a folyamat mintha semmi sem tortent volna
* ilyen informaciok pl. a memóriaterületet, állományokat, környezeti változókat
* 3 fő állapotot különböztetünk meg a folyamatok esetén: **Letrejott –** letrejott de meg nem kepes futni **Ready –** Futasra kesz, az utemezo barmikor kivalaszthathja futasra. Ebbe az allapoba kerul akkor is amikor az utemezo megszakitotta, hogy masnak adjon cpu idot.

**Futó:** az aktuális végrehajtás alatt álló folyamat

**Waiting:** erőforrásra, felhasználói beavatkozásra vár

**Terminated –** Befejezodott a futas vagy az oprendszer lallitotta, ki lehet venni a memoriabol

**Zombie –** A folyamat mar nem letezik, viszont az apa folyamathoz meg nem jutott el a kilepesi ertek, tehat nem tudja hogy befejezodott.

1. **Folyamatok ütemezése időosztásos és valós idejű rendszerekben**

* mikor több folyamat képe futni, akkor az operációs rendszernek el kell döntenie, melyik fusson először, illetve milyen sorrendben fussanak
* az operációs rendszer ezen részét ütemezőnek nevezzük, az erre a célra használt algoritmust pedig ütemező algoritmusnak
* A folyamatok utemezeset egy kulon utemezo vegzi, kulonbozo algoritmusok szerint.
* **időosztásos** rendszerek (processzor kisajatitasa, gyors valasz) esetén több ütemező algoritmus is rendelkezésre áll:
  + Round-Robin
    - minden folyamatnak ki van osztva egy időintervallum, ezalatt engedélyezett a futás, azutan mas folyamat kovetkezik
    - a strategia nehezseget a megfelelo ido szelet kivalasztasa jelenti, mivel ha egy-egy folyamat tul keves idot kap, a rendszer tobb idot fog eltolteni a folyamatok cserejevel, viszont ha ha tul sok idot, akkor a fontosabb folyamatok lehet hogy nem jutnak eleg hamar cpu idohoz
  + Prioritásos ütemezés
    - minden folyamathoz hozzárendelünk egy prioritást és a legmagasabb prioritású folyamatnak engedjük meg hogy fusson
    - annak elkerulese vegett, hogy a magas prioritasu folyamatok kizarjak az alacsony prioritasuakat, az utemezo minden megszakitas utan csokkenti az eppen futo folyamat prioriasat
  + Garantált ütemezés
  + Sorsjáték ütemezés
* **Valos ideju rendszerek:** a folyamatok gyorsan elvegzik feladataikat, csak rovid ideig futnak
* **valós idejű** rendszerek esetén megkülönböztetünk *szigorú* és *lágy* ütemezést
  + **szigorú** 🡪 minden határidőt teljesíteni kell, kevés rendszer rendelkezik ilyennel
  + **lágy 🡪** minimalizálni kell a határidő túllépéseket. Kevés esetén nincs probléma, de ha túl sokat hagyunk ki, prblémák merülhetnek fel

1. **A rendszerhívás fogalma, rendszerhívás végrehajtásának alapelve**

* a legtöbb esetben az operációs rendszer nem engedi meg, hogy az átlagos felhasználó belepiszkáljon a rendszerbe, ehelyett úgynevezett szolgáltatásokat nyújt a programoknak
* minden felhasználói program a processzor felhasználói üzemmódjában fut, ha valami olyan műveletet akar végrehajtani, amely más futó programokat megzavarhat, akkor meg kell kérnie az oprendszert, hogy hajtsa végre a kért műveletet, melynek végrehajtását a jogtól függően az vissza is utasíthatja
* a rendszerhívás folyamata egyszerű lépésekkel leírható:
  + a felhasználói program beállítja a paramétereket
  + mikor megvan, a program jelzi valamiként a kernelnek hogy rendszerhívás történik
  + a jelzés egy ún. **exception**-t dob, mely egy speciális címre ugrik, ahonnét tovább futtatja a kódot (kb mint interrupt)
  + elmentődik a jelenlegi állapot, azonosítódik és végrehajtódik a megfelelő rendszerhívás, az állapot visszaállítódik, majd a kontroll is visszatér a felhasználói program kezébe

1. **Folyamatok ütemezése időosztásos és valósidejű rendszerekben – megvolt mar a 4.kerdesnel**
2. **Atomi műveletek. Kritikus szekció. Kölcsönös kizárás. Szemaforok**

* **Atomi művelet** 🡪 olyan művelet, mely nem szakítható meg semmiként, egy atomi művelet elvégzése során semmilyen más folyamat nem nyúlhat hozza az adathoz amellyel éppen dolgozik.
* **Kritikus szekció 🡪** olyan szekció, melyben biztosítanunk kell, hogy egy időben csupán egyetlen folyamat férhet hozzá az adott erőforrásra, máskülönben versenyhelyzet alakul ki, előbb-utóbb mindegyik folyamat meg kell kapja a szükséges erőforrást
* **Kölcsönös kizárás 🡪** bizonyos erőforrásokhoz a folyamatok csakis kizárólagosan férhetnek hozza, a felhasználónak ezt biztosítania kell, másként nem garantálja semmi a megfelelő futást
* **Szemaforok 🡪** egy olyan absztrakt adattípus, melyet az erőforrásokhoz való hzzáférés szabályozásához használnak, értékének változása atomi művelet. Ha az étéke pozitív, a szemafor nyílt állapotban van, ilyenkor az első folyamat lezárhatja, majd beléphet az adott szekcióba. Ha zárva van, a folyamatnak várnia kell míg a záró folyamat felengedi. Értékétől függően több folyamat is beléphet az adott szakaszba

1. **A holtpont fogalma, kialakulásának feltételei**

* a holtpont olyan állapot, amely akkor következhet be, amikor több folyamat egyidejűleg verseng erőforrásokért, és egymást kölcsönösen blokkolják
* akkor alakul ki, amikor egy F1 folyamat foglal egy eroforrast amire F2 folyamatnak szuksege van es F2 foglal egy olyant amire F1-nek van szuksege
* ha egyik sem tud továbblépni, mert éppen arra az erőforrásra lenne szüksége, amit más foglal: ez holtponthoz vezet
* kialakulásához szükséges feltételek:
  + **kölcsönös kizárás 🡪** legalább egy – többek által igényelt – erőforrás nem megosztható, azaz egyszerre csak egy folyamat használhatja
  + **foglalva várakozás 🡪** valamelyik folyamat lefoglalt egy erőforrást és arra vár, hogy továbbiakat foglaljon le
  + **nem elvehető erőforrások 🡪** a folyamatok addig birtokoljak az eroforrast, amig onszantukbol fel nem szabaditjak azokat
  + **körkörös várakozás 🡪** a körben álló folyamatok mindegyike a következő által foglalt erőforrásra vár

**-** Kezelesi strategiak:

* Strucc alg. – Amikor kicsi a kialakulasi feltetel, es nincs kritikus kovetkezmenye a rendszer ujrainditasanak
* Vedekezunk a kialakulasa ellen, ugy tervezzuk hogy ne alakulhasson ki

1. **Virtuális memória. Virtuális memória megvalósítása lapozással**

* a virtuális memória lényegében a címzési tartományt jelenti. 32 bites rendszereknél 4GB, 64-eseknél sokkal több
* mivel ennyi memória általában nem áll rendelkezésre (64 bit esetén), illetve rossz cím megadása miatt az egész program leállhat, szükséges ennek kezelése
* megoldásként a virtuális címzés nem direkt a címsínre megy, hanem az MMU-ba, mely lényegében átkonvertálja fizikai címmé
* ha ez a cím nincs benne az operatív memóriában, akkor megkeresi, majd kicseréli a legrégebben nem használt adattal, majd frissíti mindkettőnek a fizikai címét
* ezáltal a címzéstartomány teljes egészét megcímezhetjük
* a **map**-elés hátulütője, hogy nagyon sok helyet foglalhat, ezért nem ajánlott a direkt használata
* megoldást jelent a lapozás
* a lapok adott méretű memóriarészek, általában 1-2-4 kB, de lehet több is
* 4 kB esetén, ez a tartomány megcímezhető 12 bittel
* a program nem virtuális címzést használ, hanem egy **lapszámot**, mely lényegében a 4kB-os részek kezdőcíme, majd egy **eltolást**, mely a pontos címet adja meg ezen belül
* így, mikor címezni akarunk, az MMU ellenőrzi, hogy az adott blokk bent van-e a memóriában, ha igen a lap kezdőcímét átalakítja, majd hozzáadja az eltolást, így megkapva a fizikai címet
* ha nem található benne, akkor az adott méretű lapot kicseréli a legrégebben használtal, majd frissíti a lapcímeket
* ezáltal sokkal több memória megcímezhető, és a **mappelés** folyamata sem vesz akkora memóriát igénybe

**Számítógép Architektúra**

1. **A Neumann architektúrájú processzorok minimális regiszterkészletének funkcionális leírása**

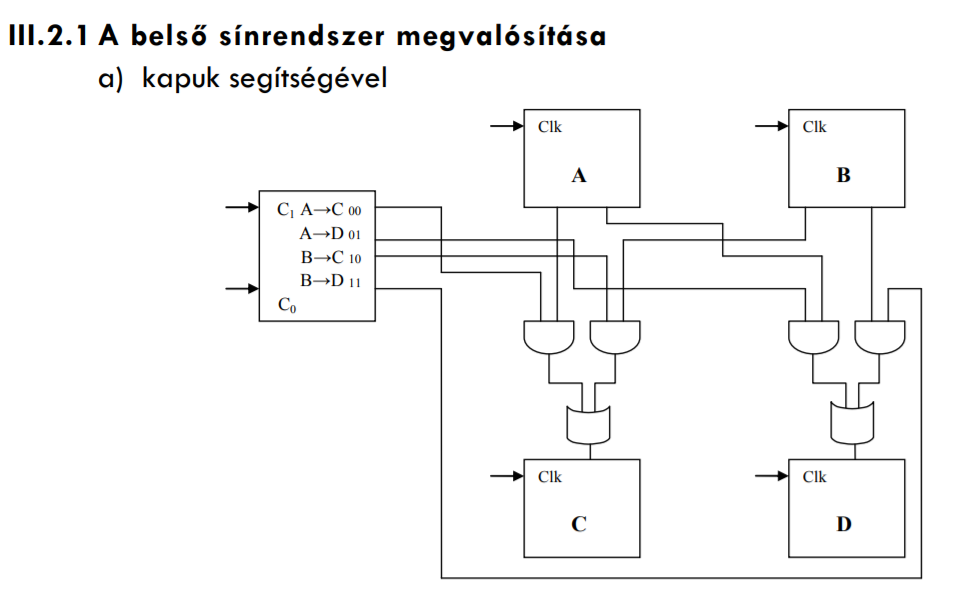
* **AC** (akkumulátor regiszter): általános célú regiszter, műveletekhez szükséges operandusok egyikét, valamit azok eredményét tárolja
* **AR** (adat regiszter): adatátvitelt biztosít a memória és a CPU belső alkotóelemei közt (olvasásnál ide kerül be először a memóriából, írásnál ide kerül bele, majd innen íródik ki)
* **CR** (cím regiszter): a memória fele közvetített (utasítás vagy operandus) címét tárolja (ide íródik be a cím, amelyen a következő utasítás, vagy éppenséggel az adat található, ezzel kerül megcímzésre a tároló)
* **PC** (program számláló): a következő utasítás címét tárolja
* **UR** (utasítás regiszter): dekódolja az utasítást, felosztja utasítás kódra (ez megy a PU fele) illetve operandus címre (ha van, pl. melyik címen van a szükséges adat)
* **SP (verem mutato)**
* **FR (flag regiszter)**

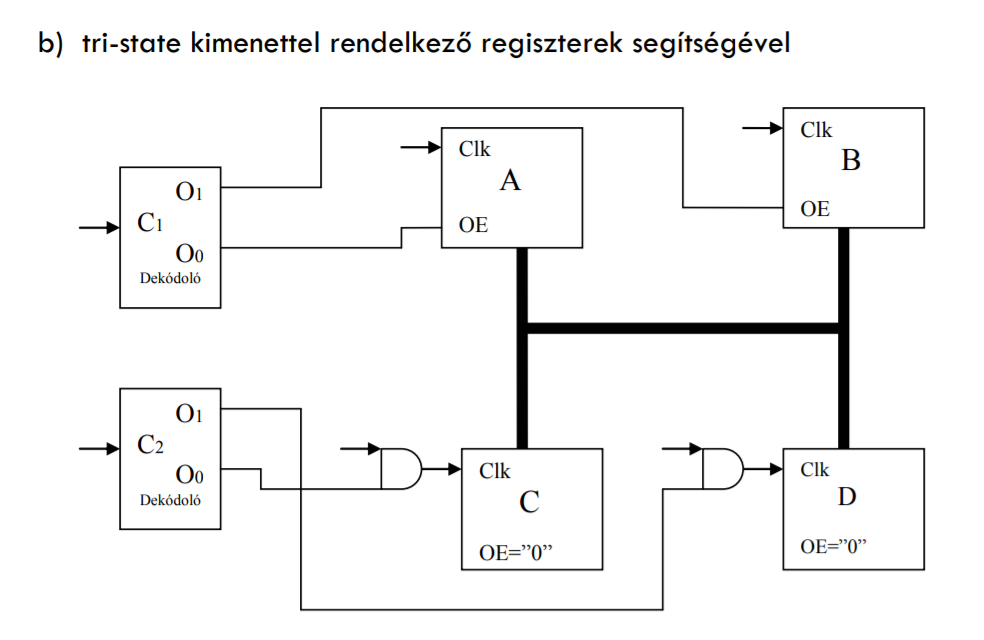
1. **A Neumann architektúrájú processzorok vezérlő egységének funkcionális leírása**

* A Vezérlő egység irányítja, tehát az utasítások végrehajtását, annak közös (kiolvasás és dekódolás – Fetch and Decode) illetve specifikus (műveletvégzés) fázisait.
* a központi egység fő feladata végrehajtani a memóriában tárolt utasításokat
* egy utasítás-végrehajtás ciklus két fázisból tevődik össze:
  + az utasítás előhívása a memóriából és ennek egy belső regiszterbe való betöltése (fetch and decode)
  + az utasítás végrehajtása (execute)
* az utasítás végrehajtása tartalmazza az utasítás dekódolását, az operandusok betöltséét, a múvelet végrehajtását, és ha szükséges, az eredmény visszaírását a memóriába
* a vezérlő egységnek kell biztosítani az utasításokhoz szükséges vezérlőjelek sorozatát
* a vezérlőjelek engedélyezik, vagy lezárják az adat útvonalát, viszont ez több útvonalon is átvezethető
* azt, hogy melyik útvonalon halad az adat, az utasítás határozza meg

1. **A processzorok belső sínjeinek – az adatút – kialakításának lehetőségei**

* A regiszterek az ALU-hoz kapcsolódnak, ez az adatok áramlásának adatútja.
* Az adatút alapfeladata az, hogy kiválassza azt az egy vagy több regisztert, amelyen az ALU műveletet végez (pl. összeadja tartalmukat, az eredményt pedig valamelyik regiszterben eltárolja).
* Egyes gépeken az adatút működését az ún. mikroprogram vezérli, míg más gépeken a vezérlés közvetlenül a hardver feladata.

****

****

1. **Függvényhívási utasítás végrehajtásához szükséges CPU alegységek és azok feladatai**

* függvényhívás esetén, a függvényhívást követő utasítás címét (**PC+1**) a vezérlőegység automatikusan elmenti a verembe (**SP**), majd a programszámlálóba (PC) betöltődik a függvény kezdőcíme
* a visszatérés esetén, a vezérlőegység kiolvassa a verem tetejéről az elmentett utasítás címét, majd betölti a programszámlálóba
* ezen kívül, ha szükséges, olyan regiszterek (pl. **FR**) állapotai is elmentődnek, melyek állapotai szükségesek a korábbi állapot visszaállításához; ezek értékei a függvényből való visszatérés esetén ismét betöltődnek

1. **Az utasítás-formátum hatása a processzorok utasítás-készlet architektúra szintjének tervezésére**

* az **ENIAC**-al szemben, a Neumann architektúrára épülő gépek más utasítás-formátummal rendelkeznek
* az ENIAC formátuma:

Utasítás kód-**x** **|** 1. operandus címe-**y** **|** 2. operandus címe-**y |** eredmény címe-**y |** következő utasítás címe-**y**

* ebben az esetben a memória lokáció mérete **x+4y**, az jelentvén, hogy csupán utasítást tud kezelni, illetve memória lokációt megcímezni
* ezzel szemben a Neumann architektúrához tartozó utasítások másként vannak felépítve:

Utasítás kód – **n |** Operandus cím – **m**

* a memória lokáció mérete **n+m**, a processzor pedig utasítást tud kezelni, memória lokáció megcímezni
* észrevehető, hogy a 2. esetben sokkal több utasítás és nagyobb méretű memória használható ugyanakkora utasításra vetítve

1. **Az operatív tárak szervezése. Lapszervezésű virtuális tár.**

* A számítógépek memóriája, felhasználás szerint 3 típusú lehet:
  + Belső memória (regiszterek): kicsi, gyors, ideiglenes tárolásra szolgál
  + Alap/központi memória: működés közbeni tárolásra szolgál, közvetlen kapcsolaton keresztül fér hozzá a CPU (sín, vezérlő jelek)
  + Háttér memória: nagy, lassú, hosszú távú tárolásra alkalmazzák
* Elérési mód szerint:
  + RAM: minden lokáció elérési ideje ugyanaz
  + Soros elérésű memória: az elérési idő változik a cím függvényében
* Fizikai felépítés alapján:
  + RAM:
    - Dinamikus – kondenzátorokból épül fel, periódikusan frissíteni kell
    - Statikus – D bistabilokból épül fel
  + ROM
  + EPROM – optikai úton törölhető, újrapprogramozható
  + EEPROM – elektromosan törölhető, újraprogramozható
  + FLASH-ROM – gyors, nagyszámú törlés/újraírást támogató ROM memória
* a virtuális memória lényegében a címzéstartományt jelenti, ez nem más mint egy másodlagos memória (háttértár)
* a virtuális memória egyenlő lapokra oszlik
* létezik egy laptáblázat, mely információt tartalmaz minden oldalról: az oldal címét a memóriában, be van-e töltve, és állapotbiteket
* egy virtuális cím egy lapcímből és egy eltolásból áll, a processzor a Memory Management Unit-on keresztül kikeresi a lapszámnak megfelelő információt, ha az alapmemóriában megtalálható, kiolvassa a kezdőcímét és hozzáadva az eltolást, kialakítja a fizikai címet
* ha nincs bent a lap, beolvasódik a háttértárról
* ha nincs elég hely a lap betöltésére, a legrégebben használt lap, kicserélődik a kérttel

1. **Vektorizált megszakításrendszert alkalmazó I/O adatátvitelek végrehajtásának lépései**

* vektorizált megszakításokat a perifériák által küldött adatmennyiség lecsökkentésének érdekében, a címküldés helyettesíthető megszakítás vektor küldésével
* ennek alapján a CPU kiolvassa a megszakítás vektor táblázatból a lekezelő rutin címét
* míg a CPU arra vár, hogy lassú perifériákkal adatcserét végezzen, más perifériák adatokat szolgálhatnak, ennek kivitelezésére egy lehetséges megoldás a megszakításrendszer
* ha az adat elvihető, a perifériás berendezés megszakításkéréssel jelentkezik a CPU felé
* ezek a megszakításkérések úgy tekinthetőek mint függvényhívás
* a CPU meghatározza a megszakításkezelő címét, majd függvényhívásként átugrik arra, lementi a PC+1-et, illetve a FR-t
* mikor az adatok elmentődtek, a függvényből visszatér a CPU, majd folytassa addigi munkáját

1. **Közvetelk memória-hozzáférést (DMA) alkalmazó I/O adatátvitelek végrehajtásának lépései**

* célja a háttértárolón lévő adat memóriába helyezése, minimális CPU időt felhasználva
* ez néhány egyszerű lépésből valósítható meg:
  + a CPU felprogramozza a DMA kontrollerét C darab byte átvitelére X címre
  + a DMA egy kérést intéz a háttértár kontrolleréhez, az adat átmásolására a memóriába
  + a kontroller inicializálja a DMA átvitelt
  + a kontroller elküld minden byte-ot a DMA kontrollerének, a DMA növeli a cél címét(++X), csökkenti a byte-ok számát(--C), ismétli míg C!=0
  + ha nincs több átvitelre váró adat, a DMA egy megszakítást küld a CPU felé, jelezvén, hogy az átvitel megtörtént

1. **Az utasítások végrehajtásának párhuzamosítása csővezeték struktúrák alkalmazásával**

* a lapozás és szegmentálás megnöveli az utasítások végrehajtásának idejét, mivel több mikorlépésre van szükség, valamit több memória hozzáférési ciklusra
* alapesetben is megkülönböztetünk 3 lépést: **decode 🡪 fetch 🡪 execute**
* ha gyorsítani akarjuk ezt a folyamatot, akkor tervezhetünk külön egységeket melyek 1-1 fázis tudnak elvégezni. Ezeket pedig pipeline struktúrába helyezzük, ekkor így alakul:

**fetch 🡪 decode 🡪 exec**

**fetch 🡪 decode 🡪 exec**

**fetch 🡪 decode 🡪 exec**

* ilyen módon elérhetünk egy gyenge párhuzmaosítást, illetve növeljük a sebességet is

**Mikrovezérlős rendszerek**

**1. Generikus mikrovezérlő utasításkészlete. Byte orientált utasítások, bit orientált utasítások, műveletek konstanssal, vezérlő utasítások.**

Egy 8 bites generikus mikrovezérlő (egy olyan mikrovezérlő modell, amely a legtöbb kis és közepes bonyolultságú mikrovezérlő általános felépítési elveit tükrözi és szerkezeti elemeit tartalmazza) utasításkészlete RISC típusú (Reduced Instruction Set Computer) utasításokat tartalmaz. Jellemző ezekre, hogy két részből épül fel: a mnemonic, vagyis az utasítás kódja, ami arra utal, hogy mit kell elvégezni és az operandus, ami általában az adatmemória egy regisztere, ami arra utal, hogy mivel kell elvégezni az utasítás műveletét.

Mivel egy utasítás rendszerint a 16 bit szó hosszúságú programmemória egyetlen regiszterét foglalja el, ezért az utasításban egyetlen regiszter címét lehet befoglalni.

A két regiszteres utasítások esetén a második regiszternek kitüntetett (fix) címe van. Ezt nevezzük akkumulátornak és A-val jelöljük. Az ilyen típusú utasításokat byte orientált utasításnak nevezzük, mert byte-ok közötti műveletek elvégzését vezérlik. A műveletek lehetnek logikai, aritmetikai vagy regiszter eltoló műveletek.

A bit orientált utasítások egy adott adatmemória regiszter adott bitjére vonatkoznak és ez a két információ az utasítás operandusában kell szerepeljen. Két típusa van: a bitbeállító utasítások (set vagy clear), és a feltételes (bitértéktől függő) utasítások.

Ha az utasítás mnemonikja tartalmazza az L (Literal) indexet, akkor az operandus nem egy adott regiszter címe, hanem egy konstans. Az utasítás műveletét ebben az esetben az akkumulátor és a konstans között kell elvégezni.

Vannak olyan utasítások, amelyek nem tartalmaznak regiszter típusú operandusokat, hanem a mikrovezérlő egy adott állapotát koordinálják. Ezek a vezérlő utasítások. Ilyenek az ugró utasítások, a szubrutin hívó és a visszatérő (szubrutinból, vagy megszakításból) utasítások, valamint az állapot szinkronizáló utasítások (pl. watchdog, meg a stanby mód).

**2. Generikus mikrovezérlő memóriaszervezése. Programmemória szerkezete, adatmemória felépítése, speciális funkciójú regiszterek (SFR).**

Egy 8 bites generikus mikrovezérlő (egy olyan mikrovezérlő modell, amely a legtöbb kis és közepes bonyolultságú mikrovezérlő általános felépítési elveit tükrözi és szerkezeti elemeit tartalmazza) memória szervezése Harvard típusú. Ez azt jelenti, hogy a program és adatmemória különálló egységeket képeznek és a szerkezetük sem egyforma (lásd a mellékelt ábrát).

A programmemória EEPROM típusú (elektromosan újraprogramozható, lassan írható, gyorsan olvasható, de a tápfeszültség lekapcsolásakor nem veszti el a tartalmát) és 16 bit szóhosszúságú regiszterekből épül fel. Minden regiszter egy-egy gépi kódú utasítást tartalmazhat. A programot a 0000H címtől kezdődően kell beírni a programmemóriába, mivel a mikrovezérlő alapállapotba tevésekor (Reset) mindig ez címeződik meg. A programmemória címzése a programszámlálóval (PC) történik, amelyik rendszerint 13-16 bit hosszúságú. (lásd a mellékelt rajzot). A megcímzett memóriarekesz tartalma az utasításregiszterbe (IREG) íródik át, hogy a teljes utasítás ciklus alatt az utasításdekóder (IDEC) rendelkezésére tudjon állni.

Az adatmemória RAM típusú (nagy sebességgel írható és olvasható, de a tápfeszültség lekapcsolásakor elveszti a tartalmát) és 8 bit hosszúságú regiszterekből épül fel, ez határozza meg az adat szélességét. Az adatbusza az aritmetikai-logikai egységgel (ALU) kommunikál, A címbuszra pedig az utasításdekóder küldi az utasításokban szereplő operandus címét. Mivel a generikus mikrovezérlő esetén az operandus címe 8 bites, ezért egy adatmemória lap 2exp8=256 byte-ot tartalmaz és lapozással lehet a memória kapacitását növelni. Az adatmemória két féle regisztert tartalmaz: általános célú regisztereket (adat regiszterek) és speciális célú regisztereket (SFR). A SFR regiszterek hardveresen vannak huzalozva a mikrovezérlő szerkezeti elemeihez és periféria moduljaihoz, és az a szerepük, hogy az utasítások segítségével (szoftveresen) lehessen beállítani és követni a hardver elemeket. A SFR regiszterek vezérlő és állapot biteket tartalmaznak.

**3. Generikus mikrovezérlő regisztercímzése. Direkt címzés, indirekt címzés, lapválasztás, táblakezelés, veremkezelés.**

Egy 8 bites generikus mikrovezérlő (egy olyan mikrovezérlő modell, amely a legtöbb kis és közepes bonyolultságú mikrovezérlő általános felépítési elveit tükrözi és szerkezeti elemeit tartalmazza) a gépi kódú utasításokkal éri el az adatregisztereit. Az utasítások operandusa rendszerint egy adatmemória regiszter, ennek a tartalmával történik az utasításban szereplő feladat elvégzése. Az operandusok címzése kétféle képen történhet.

Direkt címzés esetén az utasítás tartalmazza az operandus regiszter abszolút címét. Mivel az utasítás az operandus címét 8 biten kódolja, összesen 2exp8=256 byte-ot lehet megcímezni, tehát egy adott memórialap regisztereit.

Indirekt címzés esetén az utasítás nem tartalmazza az operandus regiszter abszolút címét, hanem csak egy utalást (IND) arra, hogy az utasításdekóder hol keresse azt. A mikrovezérlő hardverstruktúrája úgy van megtervezve, hogy az operandus címét egy speciális funkciójú regiszter, a FSR (File Select Register) regiszter tartalmazza. Ezért mielőtt egy indirekt címzésű utasítást használunk, előbb fel kell töltenünk a FSR regiszter tartalmát az operandus megfelelő címével. Mivel a FSR regiszter is 8 bites, összesen 2exp8=256 byte-ot lehet megcímezni, tehát egy adott memórialap regisztereit. Indirekt címzést akkor használunk, amikor egy programhurokban több regiszter tartalmával akarjuk ugyanazt a műveletet elvégezni, a hurok többszöri átjárásával (pl. iteráció).

A táblakezelés egy módszer, amivel a programmemóriába beírt adatokat (pl. szövegeket, konstansokat, melyeket nem akarunk elveszteni a tápfeszültség lekapcsolásakor) tudjuk áttenni az adatmemóriába. Erre szolgál a „RETLA Lit” utasítás, amelyik a szubrutin visszatérő alapfunkción kívül (RET), az utasításban szereplő 8 bites konstanst (Lit) az akkumulátorba teszi. Több ilyen RETLA utasítást egymásután írva szövegsorokat tudunk memorálni és utána behívni az adatmemóriába (az akkumulátorba), az utasítások iteratív apellálásával.

A veremtár egy speciális adatmemória (tehát RAM), ahova a szubrutin, illetve a megszakítás hívásakor a soron következő utasítás címét elmentjük. Ez egy LIFO (Last In First Out) típusuk memória, mivel a beágyazott hívások esetén mindig a legutoljára betett memóriacím hívódik elő először, a RET típusú utasítás hatására. A generikus mikrovezérlő veremtára 8 byte mélységű és egy számláló (SP – Stack Pointer) címezi meg, amit az utasításdekóder vezérel. Ebből következik, hogy 8 réteg mélységben ágyazhatunk be ismételt meghívásokat.

**4.  Generikus mikrovezérlők megszakításrendszere. Megszakításvektorok, prioritások, vezérlő regiszterek, időbeli viszonyok.**

Egy 8 bites generikus mikrovezérlő (egy olyan mikrovezérlő modell, amely a legtöbb kis és közepes bonyolultságú mikrovezérlő általános felépítési elveit tükrözi és szerkezeti elemeit tartalmazza) megszakításrendszere az utasításdekóder köré van szervezve, hiszen a megszakítás aszinkron folyamat (nincsen az utasítás ciklussal szinkronizálva), ezért a vezérlő hardver kell szabályozza.

Megszakítást kérhet elvileg a mikrovezérlő bármelyik periféria modulja, a saját IRQ (Interrupt ReQuest) jele aktiválásával. A megszakítás azonban csak akkor történik meg, ha az engedélyezve van. Két engedélyező jelre van szükség ahhoz, hogy egy periféria modul megszakítást tudjon kezdeményezi. Egy specifikus (modulra jellemző) megszakítás engedélyező bitre (INTEN) és egy általános (minden modult egyszerre) engedélyező megszakítás bitre (GIE).

Ha egy periféria modul mindkét engedélyező bitje 1-esre van állítva, és egy megszakításkérés érkezik tőle, akkor az éppen végrehajtásban levő főprogrami utasítás még befejeződik, a következő utasítás címe elmentődik a veremtárba és a program hardveres vezérléssel átugrik a 0004H címre. Ez az úgynevezett megszakításvektor, vagyis az a programmemória cím, ahonnan a megszakítás rutin kezdődik. Ugyanakkor, a megszakításvektorra való ugrás pillanatában a GIE engedélyező bit hardveresen törlődik (0-ra áll), azért, hogy egy újabb esetleges megszakításkérés ne legyen kiszolgálva mindaddig, amig az éppen futó megszakításrutin be nem fejeződik.

A megszakításrutin utolsó utasítása egy RETI típusú utasítás kell legyen, amelyik hatására a veremtárból átíródik a programszámlálóba (PC) a főprogram soron következő utasításának a címe, tehát a program visszaugrik a főprogramba és folytatja azt, ahonnan abbahagyta. Ugyanakkor a GIE általános megszakításengedélyező bit hardveresen visszaállítódik 1-esre, tehát egy újabb esetleges megszakításkérést tud fogadni a rendszer.

A megszakításengedélyező biteket egy SFR regiszterben kell beállítani, ami a megszakítás engedélyezést kontrolálja hardveresen huzalozva. A generikus mikrovezérlőnk esetében ennek a neve modulINTCTR regiszter. (ahol a modul a megszakítást kérő periféria neve, pl. TMR1INTCTR). A vezérlőbiteket a legegyszerűbben egy bitbeállító utasítással ( BS vagy BC) lehet beállítani.

A generikus mikrovezérlő egy megszakításvektorral rendelkezik, de vannak olyan mikrovezérlők, amelyekbe két vagy több megszakításvektort is beágyaztak. Ebben az esetben szintén egy SFR regiszterrel lehet beállítani, hogy melyik megszakításvektornak legyen nagyobb prioritása (ha két megszakítás érkezik egyszerre, melyik legyen hamarabb kezelve). A prioritás bitek beállítása hasonló módon történik, mint az engedélyező bitek beállítása.

**5. Generikus mikrovezérlők programozása. Tokok programozása, programozói állapot, konfigurációs bitek, hardverkörnyezet, áramkörben történő soros programozás (ICSP).**

Egy 8 bites generikus mikrovezérlő (egy olyan mikrovezérlő modell, amely a legtöbb kis és közepes bonyolultságú mikrovezérlő általános felépítési elveit tükrözi és szerkezeti elemeit tartalmazza) programozásán azt a folyamatot értjük, amely során az assembler nyelven megírt program átfordítódik gépi, bináris kóddá és ez átíródik a fejlesztő platformról a mikrovezérlő programmemóriájába.

Mivel a programmemória EEPROM típusú memória, vagyis elektromos impulzusokkal programozható (írható) memória, a gépi kódok beírása egy speciális hardver áramkörre van szükség, melynek egy része a mikrovezérlőbe van integrálva. A másik része (a nagyobb feszültségű és áramú rész) a mikrovezérlőn kívül van megvalósítva, ezt nevezzük a Programozónak. A Programozó bemenete rendszerint USB porton keresztül kapcsolódik a számítógéphez, amin a fejlesztő platform fut és ezen keresztül fogadja az érkező bináris kódokat. A mikrovezérlő felé öt szállal kapcsolódik, ezek a tápfeszültség (+ és GND), egy vezérlő bit (MCLR) és a soros szinkron kommunikációs vonal (DAT és CLK).

A Programozó vezérlőbitjét a mikrovezérlő RESET bemenetére kell kötni, ezen keresztül lehet vezérelni a programozási üzemmód beállítását a mikrovezérlőn (ez rendszerint úgy történik, hogy a Programozó a vezérlő pínre a tápfeszültségnél nagyobb feszültséget kapcsol, ennek hatására a mikrovezérlő átkapcsol a programozási üzemmódba).

Programozási üzemmódban a mikrovezérlő az arra kijelölt két port bemenetén (a generikus mikrovezérlő esetén DAT=PB6, CLK=PB7) fogadja az adatokat és írja be a programmemóriába a hardveres programozási protokoll szerint.

A mikrovezérlőbe be lehet írni a programot, vagy akár újraírni akkor is, ha már be van forrasztva a céláramkori lapra. Ebben az esetben az említett öt pínt ki kell hozni a lap szélére egy kompatibilis tűsorra, hogy a Programozót hozzá lehessen csatlakoztatni. Arra is kell ügyelni, hogy a céláramköri lapon a programozásra dedikált pínekre csak olyan alkatrészek kapcsolódjanak, amelyek nem akadályozzák meg a programozó jelek átvitelét. Ezt a környezetet nevezzük ICSP (In Circuit Serial Programming) környezetnek és eljárásnak.

Programozáskor, a végrehajtható (egzekutábilis) utasításokon kívül, az úgynevezett konfigurációs bitek felprogramozása is megtörténik. A Programozó ezeket a biteket, az utasításbitektől eltérően, a programmemória más, speciális címére fogja beírni. Ezek a speciális programmemória regiszterek hardveresen vannak huzalozva a mikrovezérlő egyes áramköreihez (hasonlóan ahhoz, ahogy az adatmemória SFR vannak huzalozva), és meghatározzák, vagy befolyásolják a mikrovezérlő adott áramköreinek a működését. Mivel ezek a vezérlő bitek a programmemóriába vannak beírva, a tápfeszültség kimaradásakor sem veszítik el értéküket (eltérően a SFR regiszterektől).

A legfontosabb konfigurációs bitek az órajel generátor működését beállító bitek, a Reset áramkör működését meghatározó bitek, a Watchdog működését engedélyező bit és a programmemória adatvédelmét korlátozó bitek.

**6. Generikus mikrovezérlők speciális regiszterei (SFR). Hardverkörnyezet, STATUS regiszter, WDT (Watch Dog Timer) regiszter, FSR (File Select Regiszter), PCL/PCLATH regiszterek.**

Egy 8 bites generikus mikrovezérlő (egy olyan mikrovezérlő modell, amely a legtöbb kis és közepes bonyolultságú mikrovezérlő általános felépítési elveit tükrözi és szerkezeti elemeit tartalmazza) speciális funkciójú regiszterein (SFR) értjük azokat az adatmemória regisztereket, amelyek hardveresen huzalozva vannak a mikrovezérlő egyes áramköreihez, rendszerint a periféria modulok vezérlő és állapot bitjeihez.

A SFR regisztereket az általános célú regiszterekhez hasonlóan szoftveresen lehet írni és olvasni a mikrovezérlő gépi kódú utasításai segítségével. Rendszerint a megfelelő vezérlő biteket írjuk (pl. a BS vagy BC utasításokkal) illetve a megfelelő állapotbiteket olvassuk (pl. a BTSS vagy BTSC utasításokkal) és segítségükkel tudjuk beállítani egy-egy periféria modul helyes működését, illetve meg tudjuk állapítani, hogy éppen milyen állapot fázisban vannak. A legfontosabb SFR regiszterek a STAUS, WDT, FSR és PC regiszterek.

A STATUS regiszter az utasításdekóderhez (IDEC) kapcsolódó regiszter, többnyire a jelző (flag) biteket és a lapválasztó biteket tartalmazza. A jelző bitek közül a legfontosabbak a Z és a CY flagek, amelyek akkor jeleznek (1-esre állnak), ha az ALU-ban elvégzett utolsó művelet eredménye zéró (00H), illetve colt átvitel (carry). A lapválasztó bitek (P0, P1, P2) segítségével az adatmemória aktuális lapját lehet beállítani. Minden adatmemória lap 256byte kapacitású, hogy 8 bittel lehessen megcímezni egy-egy regisztert. Erre azért van szükség, mert az utasítások operandus címe, illetve a FSR indirekt címző regiszterek 8 bitet tartalmaznak.

A WDT regiszter a watchdog áramkör időzítését végzi. Ebbe a regiszterbe beírt konstans határozza meg, hogy a watchdog áramkör inicializálása után (a CLRWDT utasítással) mennyi idő kell eltelnie amig az áramkör nem generál egy Reset jelet, amellyel a mikrovezérlőt alapállapotba hozhatja. (t=Tosc \* WDT, vagyis az órajel generátor periódusa szorozva a WDT regiszterbe beírt konstanssal). Ahhoz, hogy ez ne történjen meg, a programba megfelelő idöközönként egy-egy CLRWDT utasítást kell elhelyezni, ami újraindítja a watchdog számlálóját.

A FSR regiszter az adatmemória indirekt címzést szolgálja. Indirekt címzés esetén az utasítás nem tartalmazza az operandus regiszter abszolút címét, hanem csak egy utalást (IND) arra, hogy az utasításdekóder hol keresse azt. A mikrovezérlő hardverstruktúrája úgy van megtervezve, hogy az operandus címét egy speciális funkciójú regiszter, a FSR (File Select Register) regiszter tartalmazza. Ezért, mielőtt egy indirekt címzésű utasítást használunk, előbb fel kell töltenünk a FSR regiszter tartalmát az operandus megfelelő címével.

A PC (Program Counter) regisztereket a programmemória címzésére használjuk. Mivel a programmemória címbusza rendszerint 13-16 bites (13 bit a generikus mikrovezérlőnk esetén), ezért két 8 bites SFR regiszterre van szükségünk, hogy a tartalmát írni és olvasni tudjuk. Ezek a PCL és a PCLATH nevű regiszterek, melyek a programszámláló alsó, illetve felső 8 bitjét tartalmazzák. A PCLATH regiszter abban különbözik a PCL regisztertől, hogy tamponként működik (latch), vagyis tartalma nem íródik be rögtön a programszámláló felső felébe (PCH), hanem csak akkor, amikor a PCL regisztert is írjuk. Erre azért van szükség, hogy mind a 16 bit egyszerre íródjon be a programszámlálóba.

**7. Az assembler nyelv szintaktikája. Címkemező, utasításmező, operandusmező, megjegyzésmező, direktívák. A gépi kódra fordítás lépései.**

Mikrovezérlők esetén, a gépi kódú programokat assembler nyelven írjuk meg, a fejlesztői platform szövegszerkesztőjében. Egy assembler nyelven írt utasítás négy mezőben helyezkedik el, szóhatárra igazítva (oszloponként) a szövegszerkesztő egy sorában. Ezek a címke, mnemonik, operandus és komment mezők.

Jellemzően, egy utasítás két fő részből épül fel: a mnemonic, vagyis az utasítás kódja, ami arra utal, hogy mit kell elvégezni és az operandus, ami általában az adatmemória egy regisztere, ami arra utal, hogy mivel kell elvégezni az utasítás műveletét.

Ugró utasítások esetén, vagy a szubrutinok, megszakításrutinok meghívásakor tudnunk kell azt a címet, ahova ugorni akarunk, vagy ahol a meghívott szubprogram kezdődik. Ezért az ugrási pontokat és a szubprogramok kezdetét címkével jelöljük, az ugró utasításokban vagy a szubprogram hívásokban ezekre a címkékre hivatkozunk. A megfeleltetést az assembler fogja elvégezni, átalakítva a címkék helyzetét abszolút címekké.

A megjegyzés mező fontos eleme az assembler nyelven írt gépi kódú programoknak. Mindig egy pontosvesszővel kezdődik a negyedik mezőben és az éppen elvégzendő utasítás folyamatát kommentálja. Azért fontos, mert később (napok, hetek, hónapok múlva) könnyebben vissza tudunk emlékezni arra, hogy mit is akartunk gépi kódban programozni, tekintettel arra, hogy ezen a gépközeli (alacsony szintű) programnyelven a programozás folyamatát relatív nehéz nyomon követni. A kommenteken kívül a nyomon követést elősegíti, ha a programmoduloknak folyamatábrákat készítünk.

Egy assemblerben megírt programnak nem csak végrehajtható (exekutábilis) utasításai vannak. Az assemblert utasítani és egyben informálni kell, hogy hogyan fordítsa le az utasításokat bináris kódba és hogyan küldje el ezeket a mikrovezérlő programmemóriájába. Erre szolgálnak a nem végrehajtható utasítások, vagyis a direktívák. A fontosabb (leggyakrabban használt) direktívák az ORG, amely megmutatja, hogy az utána következő végrehajtható utasításokat milyen programmemória címre kell beírni, az EQU, amely hozzárendel egy címkéhez egy abszolút címet, az END, amelyik a végrehajtható program végét jelöli, és a CONFIG\_, amivel a konfigurációs bitek értékét lehet meghatározni.

Az assemblerben írt gépi kódú program bináris kódra való fordítása (kompilálása) két lépésben történik. Az első átfutáskor az assembler megoldja a címkék átfordítását abszolút címekké, valamint ellenőrzi a szintaktikai (alaki) és szemantikai (értelmezésbeli) hibákat, mint pl.: téves mnemonik jelölés, operandus hiánya, operandus helytelen használata, akkumulátor helytelen használata, címke hiánya vagy téves jelölése. Ezeket hibaüzenetek, vagy figyelmeztetések formájában meg is jeleníti. Ha nem talál hibát, akkor a második átfutáskor elkészíti a fordítási listát (az utasítások kód és cím megfeleltetését, valamint a címkék táblázatát a hozzárendelt abszolút értékkel együtt) és a forrásprogramot (src), valamint megjelenít egy statisztikát a lefordított programról.

**8. Generikus mikrovezérlő időzítő perifériái. Hardverkörnyezet, számláló üzemmód, időzítő üzemmód, kiolvasás/összehasonlítás (Capture/Compare) üzemmód, PWM moduláció.**

Egy 8 bites generikus mikrovezérlő (egy olyan mikrovezérlő modell, amely a legtöbb kis és közepes bonyolultságú mikrovezérlő általános felépítési elveit tükrözi és szerkezeti elemeit tartalmazza) egyik leggyakrabban használt periféria modulja az időzítő (Timer) modul. Használata elősegíti, hogy a legtöbb időzítő, számláló, impulzus generáló funkciókat hardveresen oldjuk meg, vagyis az inicializáló rutinok segítségével beállítjuk a modul működését és utána az önállóan fogja megoldani a problémát, anélkül, hogy a mikrovezérlő részéről értékes utasítás időt és programrészeket kéne erre elhasználni. Rendszerint egy egyszerű mikrovezérlő is egynél több időzítő perifériát tartalmaz.

Egy Timer modul három fő egységre oszlik (lásd a mellékelt ábrát): bementi kontrol áramkör, soros bináris számlálók, kimeneti (megszakítást kontroláló) áramkör. A generikus mikrovezérlő számláló része három sorbakapcsolt 8 bites számlálót tartalmaz, vagyis egy 24 bites bináris számlálót alkot, amelyik 000000H-tól FFFFFFH-ig tud számolni, felfele, vagy lefele. A számlálók hardveresen huzalozva vannak három SFR regiszterhez (TMRxL, TMRxM és TMRxH), így lehetséges a számlálók írása és olvasása szoftveresen gépi kódú utasítások segítségével.

A bemeneti áramkört két fő vezérlő bit koordinálja: a számlálást engedélyező bit (TMRxEN), valamint a bejövő impulzusok típusát meghatározó bit (TMRxMD). Ez utóbbi értéke alapján (0 vagy 1), a 24 bites számlálóra két irányból érkezhetnek impulzusok: vagy a mikrovezérlő óragenerátorától érkeznek az impulzusok (ebben az esetben időszámlálásról, mérésről beszélünk), vagy a mikrovezérlő egyik I/O portlábán keresztül a külvilágból (ebben az esetben eseményszámlálásról beszélünk).

Az említett két vezérlő biten kívűl még két vezérlő bittel lehet meghatározni a modul működését: a számlálás irányát (le vagy fel) meghatározó bit (TMRxU/D), valamint a Timer modul üzemmódját meghatározó két bit (CCPxM1 és CCPxM2). Ezek segítségével négy üzemmódot tudunk beállítani a modulon: az előbb ismertetett számláló üzemmódot, a kiolvasó, az összehasonlító és végül az impulzus szélesség moduláló üzemmódot.

A kiolvasó üzemmód esetén, egy vezérlő bit aktiválásakor (CCPxEN) a számlálók tartalma átíródik három párhuzamos regiszterbe, amelyek szintén hardveresen vannak huzalozva három SFR regiszterhez (CCPx1, CCPx2 és CCPx3). Ezáltal nyugodtan lehet hosszabb időn keresztül feldolgozni a számláló értékét egy adott pillanatban, mivel ezt az értéket lementhetjük a CCP tampon regiszterekbe.

Összehasonlító üzemmód esetén a CCP regiszterekbe van lehetőség beírni egy 24 bites értéket és a modul ezzel a számmal fogja összehasonlítani a számlálóba érkező impulzusok számát. Megegyező érték esetén a modul megszakítást kérhet. Ilyen módon meg lehet határozni, hogy a számláló meddig számoljon (időzítsen), mielőtt elindítanánk egy új műveletet.

Impulzus szélesség modulációs üzemmód (PWM) esetén egy második rend három byte-ból álló párhuzamos regiszter is aktiválódik. Ezek szintén hardveresen vannak huzalozva három SFR regiszterhez (PRxL, PRxM és PRxH). Ide kell szoftveresen beírni a generált PWM impulzus periódusának megfelelő 24 bites számértéket. A kitöltési tényezőnek (hogy mennyit áll 1-esben az impulzus egy periódus alatt) megfelelő 24 bites számértéket pedig a CCP regiszterekbe kell beírni. A TMR számlálók bemenetére a mikrovezérlő óragenerátorának az impulzusai kerülnek. A PWM rendszer kimenete a mikrovezérlő valamelyik I/O portjának egyik pínjére vezetődik ki, ezáltal vezérelni lehet egy külső áramkört PWM jellel.

A Timer modulok kimeneti megszakítást kontroláló áramköre biztosítja a megszakításkérő jelet (TMRxINTFL) és be lehet állítani rajta a megszakítást engedélyező biteket (TMRxINTEN és GIE). A Timer modul összes vezérlő és állapot bitjeit két SFR regiszter tartalmazza (TMRxCTR és TMRxINTCTR), ezeken keresztül lehet őket hardveresen beállítani vagy leolvasni.

**9. Generikus mikrovezérlő kommunikációs perifériái. Hardverkörnyezet, I/O portok, párhuzamos portok, szinkron soros portok, aszinkron soros portok. Az üzenetek szerkezete.**

Egy 8 bites generikus mikrovezérlő (egy olyan mikrovezérlő modell, amely a legtöbb kis és közepes bonyolultságú mikrovezérlő általános felépítési elveit tükrözi és szerkezeti elemeit tartalmazza) rendelkezik számos kommunikációs perifériával, amelyeken keresztül tartja a kapcsolatot a külvilággal. Ezek két féle típusúak: párhuzamos vagy soros adatátviteli egységek, aszerint, hogy az adatok átvitele egyszerre történik 8 biten, vagy ütemesen, bitenként viszik át egyetlen adatvonalon. Minden mikrovezérlő rendelkezik legalább egy hardveres párhuzamos és egy hardveres soros kommunikációs megoldással.

A mikrovezérlők alapvető kommunikációs eszköze a párhuzamos I/O port. Ez 8 bit hosszúságú és egy mikrovezérlő egyszerre több porttal is rendelkezhet (PortA, PortB, PortC stb.). Ezeken keresztül tartja a kapcsolatot a külvilággal. Az I/O portok 8 bitenként vannak szervezve, mivel minden port hardveresen kapcsolódik két, speciális funkciójú (SFR) regiszterhez. Az egyik SFR regiszter (DIRx) meghatározza a port bitjeinek az irányát (IN vagy OUT), míg a másik SFR regiszter (PORTx) meghatározza a port bitjeinek az értékét. Ha pl. a DIRA regiszter valamelyik bitjét 0-ra állítjuk, akkor a PortA megfelelő bitje kimenetnek (OUT) konfigurálódik. Ha viszont a DIRA adott bitjét 1-re állítjuk, akkor az A port megfelelő bitje bemenetnek (IN) konfigurálódik. Ha pl. az A port valamelyik bitje kimenetre van állítva, akkor az értékét (0 vagy 1) úgy állítjuk be, hogy a PORTA regiszter megfelelő bitjébe írjuk be ezt az értéket. Ha viszont az A port adott bitje bemenetre van állítva, akkor a PORTA regiszter megfelelő bitjét beolvasva megkapjuk az adott bemeneten levő logikai állapotot. Az I/O portok kezelését nagymértékben megkönnyíti az a lehetőség, hogy a portok paramétereit szoftveresen, gépi kódú utasításokkal tudjuk kezelni.

Az I/O portok pínjei (bitjei) multiplexelve vannak több belső periféria modulhoz, azért, hogy ezek a modulok is tudjanak a külvilággal kommunikálni. Azt, hogy melyik pín milyen modulhoz van hozzárendelve az adott modul SFR regisztereiben található vezérló bitekkel tudjuk kiválasztani. Ilyen módon lehet hozzárendelni I/O píneket a belső, soros kommunikációs modulokhoz is. Ezek rendszerint két félék lehetnek: aszinkron soros kommunikációs modulok, amelyek csak egy fizikai szálon kommunikálnak (ez az adatszál, a DAT) és szinkron soros kommunikációs modulok, amelyek két fizikai szálon kommunikálnak (az adatszál a DAT, és az órajel a CLK). Van olyan szinkron modul is, amely két adatvezetéket használ, egyet a bejövő adatoknak (MISO) és egyet a kimenő adatoknak (MOSI). Ezen kívül még egy engedélyező vezetéket (CS) is használ. Ilyen a struktúrája a leggyakrabban használt SPI nevű szinkron soros modulnak is. Egy másik gyakran használt szinkron soros modul az I2C. A gyakrabban implementált aszinkron soros modulok közül kiemelkedik az USART modul, amelyet mindkét üzemmódra lehet konfigurálni. Aszinkron üzemmódban egy relatív biztonságos adatátviteli protokollt használ, amely START és STOP bitek használatára épül.

Az aszinkron soros protokollok általában legalább egy START bitet használnak a kommunikáció indítására és időzítést használnak az adatbitek küldésére. Minden bit egy periódusnyi időre marad az adatbuszon. A vevő modul a START bittel szinkronizálja magát, és ugyanazt a periódusú időzítést generálja saját magának, mint az adó periódusa. A vevő az időzítés fázisát eltolja fél periódussal, hogy az adatok beolvasását a bitek vonalidejének (az az idő ameddig egy adatbitet az adatvonalon tartanak) a közepén kezdeményezze. Így nagyobb adatátviteli sebességet lehet elérni.

A szinkron soros protokollok az órajel frontjait használják az adatbitek szinkronizálására. Általában az adó modul az órajel felfutó élein teszi ki a következő adatbitet az adatvonalra. A vevő az adatbitek beolvasását az órajel lemenő élein fogja elvégezni. Mivel az órajel közös (mindig az adó generálja és a CLK vonalon elküldi a vevőnek), valamint a kitöltési tényező 50% (egy periódus alatt az órajel ugyanannyi ideig van 1-ben, mint amennyit 0-ban), biztosítva van az olvasás félperiódusos eltolása, vagyis az olvasás mindig az adatbit vonalidejének a közepén történik.

**10. Mikrokontrolleres fejlesztés lépései és eszközei. Integrált fejlesztő környezetek (IDE), a programfejlesztés lépései, programozási alapelvek: címkiosztás, változók használata, paraméterátadás, feladat (task) kezelés, watchdog használata.**

A mikrovezérlős alkalmazások fejlesztőse rendszerint egy kompatibilis integrált fejlesztői platformon történik (IDE – Integrated Development Environment). Azért nevezik integrált platformnak, mert egy platformon gyűjti össze az összes fejlesztéshez szükséges modult. Ezek a következők: a Szövegszerkesztő (Editor), ahol a programot és direktíváit írjuk meg és szerkesztjük szöveges formában; a Fordító (Compilator), amelyik az assembler nyelven (vagy más magasabb szintű nyelven) megírt programot átkonvertálja bináris kódokká; a Szimulátor, amelyik a lefordított programot off-line futtatja akár lépésenként és kijelzi a különböző regiszterek pillanatnyi értékeit; opcionálisan az Emulátor, amivel a programot on-line lehet futtatni akár az alkalmazás áramköri környezetében; ; és a Programátor, amivel a lefordított gépi kódú programot fel lehet tölteni a mikrovezérlő programmemóriájába. A legismertebb IDE platformok a Microchip cég MPLAB fejlesztői környezete, vagy az ARDUINO nevű, nyílt forráskódú fejlesztői platform.

A programfejlesztés lépései:

1 – A feladat elemzése

2 – Az alkalmazás megtervezése

3 – Folyamatábra (Organigram) elkészítése

4 – Kódolás, felhasználva a gépi programozás elveit (megszakításkezelés, perifériakezelés stb.)

5 – Fordítás, ellenőrzés, szimulálás (debugolás)

6 – Betöltés, felprogramozás

7 – Futtatás

8 – Korrekciók

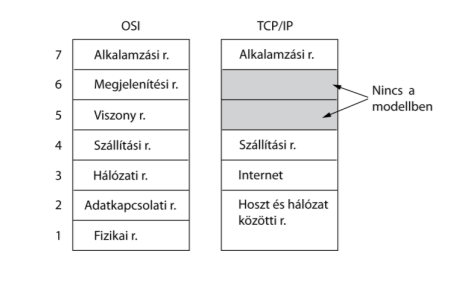
9 – Dokumentálás

A kódolás műveletekor (4. lépés) figyelembe kell venni a gépi kódú programozás alapelveit. A címkiosztás az EQU direktívával történik és minden olyan változónak meg konstansnak adunk az elnevezésére jellemző nevű címkét, amelynek abszolút címét vagy értékét a programban használjuk. A változók címének kijelölésekor figyelembe kell venni a SFR regiszterek, valamint az általános célú regiszterek címkiosztásait (nem keveredhetnek egymással). A főprogram és a szubrutinok, valamint a megszakításrutinok közös változóit paraméterátadásra használhatjuk (pl. a megszakításrutin kiszámítja az értékeit és a főprogram felhasználja vezérlésre, vagy fordítva). A főprogram változóit feladat (task) kezelésre is használhatjuk (pl. a főprogramban változtatjuk a változó értékét, a megszakításrutin vagy valamelyik szubrutin pedig az adott értéknek megfelelő feladatot fogja elvégezni).

Fordításkor (5. lépésnél): Az assemblerben írt gépi kódú program bináris kódra való fordítása (kompilálása) két lépésben történik. Az első átfutáskor az assembler megoldja a címkék átfordítását abszolút címekké, valamint ellenőrzi a szintaktikai (alaki) és szemantikai (értelmezésbeli) hibákat, mint pl.: téves mnemonik jelölés, operandus hiánya, operandus helytelen használata, akkumulátor helytelen használata, címke hiánya vagy téves jelölése. Ezeket hibaüzenetek, vagy figyelmeztetések formájában meg is jeleníti. Ha nem talál hibát, akkor a második átfutáskor elkészíti a fordítási listát (az utasítások kód és cím megfeleltetését, valamint a címkék táblázatát a hozzárendelt abszolút értékkel együtt) és a forrásprogramot (src), valamint megjelenít egy statisztikát a lefordított programról.

**Hálózatok**

1. **Sorolja fel az OSI és a TCP/IP referenciamodell rétegeit és határozza meg röviden a rétegek feladatait**



* **OSI:**
  + **Fizikai réteg:** 
    - Feladata a bitek továbbítása a kommunikációs csatornán
    - Feszültésgszintek hozzárendelése a logikai értékekhez
    - A fizikai réteg kódolja a kereteket és létrehozza azokat az elektromos, optikai vagy rádióhullám jeleket, amelyek a keret bitjeinek felelnek meg
    - Ezután a jelek egyesével elküldésre kerülnek az átviteli közegen.
    - A célállomás fizikai rétege fogadja ezeket a jeleket a közegen, bitekké alakítja őket, majd a biteket keretként továbbítja az adatkapcsolati rétegnek.
    - Az interfész, mechanikai és elektronikai kérdésekre összpontosít
  + **Adatkapcsolati réteg:**
    - Az átvitel a fizikai rétegnél nem tökéletes, ezért megpróbálja kijavítani
    - Hibaérzékelő és hibajavíró kódokat tartalmaz
    - Keretezés, nyugtázás a csomag vesztések kezelésére
    - Adatforgalom szabalyozasa, hogy a lassu vevot ne arasszak el
    - Mindezen celokra ugynevezett keretekbe agyazza az adatokat
  + **Hálózati réteg:**
    - Feladata a csomag eljutatása a célig (az adatkapcsolási réteg csak egyik hossztól a másikig viszi a csomagot)

Ismernie kell a kommunikációs alhálózatot:

útkeresés

torlódások elkerülése

különböző hálózatok közötti átmenet

Alhálózat működését irányítja

* + - Útkeresés:
      * Statikus: táblázatokban
      * Dinamikus: minden csomag új útvonalat keres
    - Torlódás kezelése
    - Szolgáltatás minőségének biztosítása
    - Különböző hálózatok közti átmenet megoldása (Ipv4 és Ipv6)
  + **Szállítási réteg:**
    - Célja a hatékony, megbízható és gazdaságos adatszállítás biztosítása a forrásgéptől a célgépig függetlenül az alatta elhelyezkedő rétegek típusától.
    - Adatokat fogad a viszony rétegből, feldarabolja és továbbítja a hálózati rétegnek
    - Feladata, hogy az adat hibátlanul megérkezzen a túlsó oldalra, vagy jelezze a hibákat
    - Itt dől el a szolgáltatás típusa: kapcsolat vagy üzenetorientált
  + **Viszonyréteg:**
    - Két hoszt közötti viszony felépítése
    - Párbeszéd kialakítása (handshake), szinkronizáció
  + **Megjelenítési réteg:**
    - Adattípusok közti konverziót hajtja végre
    - encrypt, decrypt, tömörítés
  + **Alkalmazási réteg:**
    - Gyakran használt protokoll sokasága: HTTP, HTTPS, ftp, mail
* **TCP/IP** [**https://www.tutorialspoint.com/The-TCP-IP-Reference-Model**](https://www.tutorialspoint.com/The-TCP-IP-Reference-Model)**:** 
  + **Alkalmazási réteg:**
    - A felhasználó által indított program és a szállítási réteg között teremt kapcsolatot a leggyakrabban használt protokollokat tartalmazza (HTTP, FTP, SMTP)
  + **Szállítási réteg:**
    - Itt dől el, hogy milyen protokollokkal küldi tovább az adatokat (TCP vagy UDP)
  + **Internetréteg – halozati retegnek felel meg:**
    - Az egész architektúrát összefoglalja
    - Bármilyen hálózatba csomagokat tudjon küldeni, és meg is érkezzen a köztes hálózatok típusától és a csomagok sorrendjétől
  + **Host es Halozat réteg:** nem specifikalt, a LAN-tol fug

**-** ez felel meg a fizikai retegnek, a bitek tovabbitasaert felelos

1. **Sorolja fel a TCP protokoll szolgáltatásait, tulajdonságait**

* egy összeköttetés alapú protokoll, nem képes adatszórásra és többesküldésre (multicast)
* *bitfolyamot biztosít* két végpont között, tulajdonságaik: rendezett, hibamentesek, nincs adat kettőzés
* Socketeken kommunikalniak. Socket tartalmazza az Ip cimet es a portot ahol elerik egymast
* *nem örzi meg az üzenet határokat*: küldheti azonnal vagy pufferelheti
* *push függvény*: ha az üzenet azonnali küldi az adatokat, nem puffereli őket, hanem amilyen gyorsan csak lehetséges, továbbítsa
* *sürgős adat*: a protokoll megszakítja az előbbi adatok feldolgozását, ahogy a fogadó is, majd a csomag kezelése után visszatér eredeti csomagok kezeléséhez
* Képes alkalmazkodni az összekapcsolt hálózatok változó paramétereihez.
* A TCP protokoll megvalósítása lehet felhasználói folyamat, könyvtári eljárás vagy az operációs rendszer része.
* A TCP a helyi folyamatoktól kap adatokat, amelyeket szegmensekre bont és IP csomagok segítségével továbbit a hálózatban

---------------------------------------------

A TCP nyújtotta szolgáltatás lényegi jellemzői A TCP lényegi működési jellemzői öt pontban foglalhatók össze:

**1. Adatfolyam orientált (Stream oriented)**

Az alkalmazás által továbbítani kívánt adatokat bitek – helyesen oktetek – sorozatának tekintjük. Az adatfolyam-szolgáltatás pontosan ugyanazt a bitsorozatot adja át a célgépen futó alkalmazásnak, amelyet az adatokat küldő gép adott át az átviteli szolgáltatásnak.

**2. Virtuális áramköri kapcsolat (Virtual Circuit Connection)**

Telefon kapcsolathoz hasonlítható. Az alkalmazások az operációs rendszerhez fordulnak, kérik az átviteli szolgáltatást. Az operációs rendszer kérésére a végponti protokoll-gépek kapcsolatba lépnek egymással, ellenőrzik, mindkét fél kész-e a kapcsolat létrehozására, majd megállapodnak a részletekben. Ezután a protokoll-gépek értesítik a végpontokon futó két alkalmazást, hogy a kapcsolat létrejött, megkezdhetik a kétirányú adatátvitelt. A virtuális áramkör megléte alatt a két protokoll-gép együttműködése biztosítja, hogy az átvitt adatok hibátlanok legyenek. Csak a helyrehozhatatlan hibákat jelentik az alkalmazásoknak (ekkor azonban a virtuális áramkör megszakad). Azért nevezzük a kapcsolatot virtuális áramkörnek, mert az alkalmazások úgy látják, mintha az egy dedikált, külső körülmények által meg nem zavart kapcsolat lenne. A megbízhatóságot az adatfolyam átviteli protokoll biztosítja.

**3. Pufferelt átvitel (Buffered Transfer)**

Az alkalmazások (közel) tetszőleges mennyiségű adatot adhatnak át átvitelre a szállítási szolgáltatásnak. Az adatokat a protokoll-gép pufferben gyűjti, majd a hatékonyságot szem előtt tartva kisebb-nagyobb csomagokban továbbítja azokat. Szükség esetén akár egy-egy oktet is átvitelre kerül (pl. egy billentyűleütés). Az erre szolgáló úgynevezett push mechanizmus kényszeríti a protokollt, hogy a puffer megtelte előtt vigye át az adatot. Ilyenkor a vételi oldalon működő protokoll-gép késleltetés nélkül átadja az adatot az alkalmazásnak.

**4. Strukturálatlan adatfolyam (Unstructured Stream)**

A TCP által kézbesített adat nem strukturált. A szállítási szolgáltatás semmit sem tud az átviendő adatok tartalmáról, azok struktúrájáról. Az alkalmazásoknak kell megegyezniük az adatok szerkezetében, és megérteniük az adatfolyamot, felismerni az abban lévő esetleges adathatárokat.

**5. Egyidejű kétirányú kapcsolat (Full Duplex Connection)**

A TCP kapcsolat egyidejű adatfolyam-átvitelt biztosít mindkét irányban. Az alkalmazások lezárhatják az egyik irányú adatfolyamot, ha kívánják (az ettől kezdve fél-duplex lesz). A kétirányú kapcsolat azért is előnyös, mert az ellenkező irányban haladó adatfolyam vezérlő – szolgálati – üzenetet is továbbíthat. Ez – a piggy-backing-nek nevezett – megoldás csökkenti a hálózati forgalmat

1. **Sorolja fel az UDP protokoll szolgáltatásait, tulajdonságait**

* összeköttetés nélüli protokol
* nem garantálja a csomag célba érkezését
* nem rendelkezik semmilyen visszajelzéssel a felhaszánló fele
* alacsony késleltetéssel rendelkezik
* (8 bájtos fejrész + adat) szegmensbál áll
* Szolgáltatásai
  + - Interfész az IP protokollhoz (bind kapcsolódik egy portra)
  + - Multiplexeli a csomagokat
  + - UDP nem végez forgalomirányítást, Hibakezelést, Újraküldést rossz szegmens esetén
  + - Alkalmazás – olyan alkalmazások, amelyeknek fontos a pontos időkezelés, csomagforgalom, és hibakezelés
  + - Nem kell kiépíteni a kapcsolatot

- Kapcsolatmentes. A feladó és a címzett nem beszélgetnek: a feladó elküldi a csomagot, a címzett vagy megkapja, vagy nem. De semmilyen formában sem jelez vissza.   
- Megbízhatatlan. A csomagokban nincs semmi sorszámozás, nincs semmi elveszés elleni védelem. UDP esetében minden ilyesmit arra az alkalmazásra bíznak, amelyik küldi/fogadja a csomagokat. (CRC ellenőrzés ugyan van, de az csak szótlan csomageldobáshoz vezet.)   
• Nem tárol. Ahogy beérkezik egy UDP csomag, az rögtön megy is tovább az alkalmazás réteg felé. Ha az nem veszi át, akkor úgy járt.   
• Nem tördel. Tördeljen helyette más. (Ezt mondjuk megteszi az IP réteg.)   
• Nincs benne folyamatszabályozás. Azaz nem tud se gyorsítani, se lassítani.   
• Hagyományos. A megszokott dobozolós technológiát követi.

A teljesség kedvéért említsünk meg néhány példát a használatára.

• A DNS nagyon jó példa, ugyanis az vegyesen használ UDP-t, illetve TCP-t. Ha elfér az üzenet egy UDP csomagban - tipikusan névfeloldási kérés, illetve válasz - akkor az UDP lesz a befutó. Ha több csomagra lesz szükség - tipikusan zónatranszfer - akkor viszont a TCP.  
 • Van, amikor maga az alkalmazás szintű protokoll foglalkozik a csomagok szállításának megbízhatóságával. Ilyen pl. a TFTP.   
• Van, amikor a feladó periodikusan küldi el az információkat. Ilyenkor nem gond, ha az egyik küldemény nem érkezik meg, hiszen hamarosan megy a másik. Ezt tudja a korábban már említett RIP.   
• Végül a multicast. A TCP csak egy-egy tipusú kapcsolatot tud kezelni.

**Szoftver Tervezes**

**1.) Szoftver projekt fejlesztés lépései.**

Ezen lépések azért lettek kitalálva és feljegyezve, hogy megkönnyítsék a programozók, fejlesztők dolgát, olyan fontos fázisokat foglal magába, amelyek elengedhetetlenek számukra. Az idő elteltével több ilyen modell jött létre, elég csak a spirál, szökőkút, vagy a vízesés modellre gondolni, mindközül az utóbbi a legelterjettebb és legfontosabb, ez abból áll, hogy az egyik fázis végkimenetele a másik kezdete.

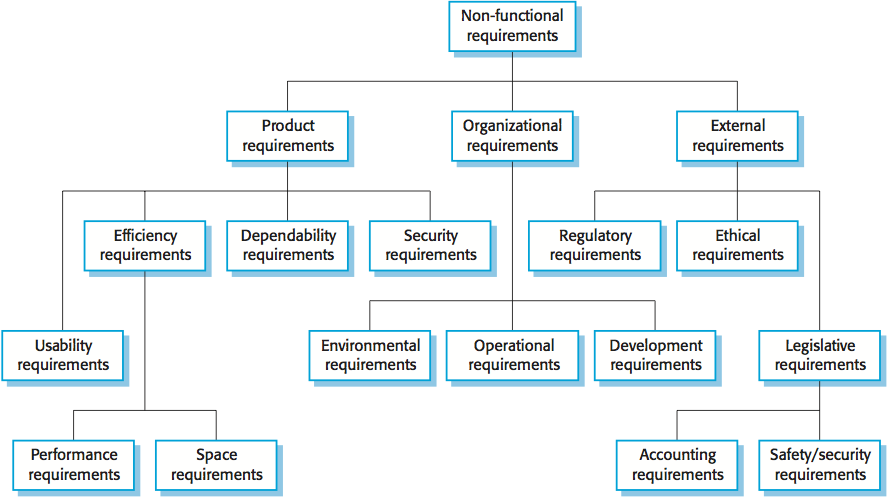
**Fázisok:**

1. **Előzetes tervezés:** Felfedezni a szervezet céljait, problémáit. Ezt követően számos megoldás jöhet létre a problémák orvoslására. Költségek kielemzése és kiírása, amelyek a céget súlythatják a fejlesztés során.
2. **Rendszerezés, követelmények meghatározása:** Itt konkrét tervekkel áll elő a programozói cég, ajánlást tesz a problémák megoldására. Tényket gyűjt (*felhasználói követelmények beszerzése* dokumentációval, ügyfélinterjúk stb.), a meglévő rendszert vizsgálja (előnyök hátrányok), javasolt rendszert elemez (megoldást talál a hátrányokra, hiányosságokra).
3. **Tervezés:** Ebben a lépésben a kívánt funkciókat és műveleteket részletesen ismertetjük, beleértve a frontend és a backend részt. (UML, pénzügyi tervek stb) Kik vannak benne? A projektvezető. A fejlesztő nincs benne általában.
4. **Implementáció**: Itt történik a kódolás. Követelmény specifikáció alapján kódolni. Tesztelés (egyesek külön fázisként teszik). Unit tesztekre gondolunk, nem adjuk ki a kezünkből a szoftvert és akkor teszteljük. Még lehet javítani rajta.
5. **Karbantartás:** Karbantartjuk a meglévő rendszert, hogy az ne tudjon elavulni a konkurencia mellett. Tesztelés itt is lehet, a kiadott programot teszteljük és javítjuk a bugokat.

2.)  **Követelmény specifikáció**

Ennek tudatában tervezzük meg, majd építjük fel a rendszert. A szoftverek tervezése során nehéz megszabni azt, hogy pontosan hogyan kellene működnie a rendszernek, meg hogyan kellene kinéznie a UI-nak, ehhez jönnek segítségül a követelmények. A követelmények tervezése az a folyamat, amikor meghatározzuk és elemezzük a riválisokat, hasonló rendszereket, valamint a megszorításokat, azt követően ellenőrizzük és ledokumentáljuk azokat. A követelmény, mint fogalom nagyon tág jelentéssel bír az IT-ban, a tárgyalását általában két részre bontják: felhasználói, illetve rendszer követelményekre.

1. **Felhasználói követelmények:** A felhasználói követelmények technikai értelemben magas szintű dokumentálás diagrammokkal kiegészítve, amely elsősorban azoknak a személyeknek készül, akik nincsenek tisztában a rendszer kivitelezésével, fekete dobozként tekintenek a rendszerre és nem érdekli őket a beljeseje csak a működése, nem rendelkeznek részletes háttérinformációval, technikai ismeretekkel a rendszerről. Következtetésképpen elsősorban a felhasználóknak, az ügyfeleknek, illetve ezeket képviselő menedzsereknek készül.
2. **Rendszer követelmények:** A rendszerkövetelmények tartalmazzák a szoftver funkciók és megszorítások leírását és azt, hogy mit kell leprogramozni a rendszerbe. A rendszerkövetelményeknek két típusát különböztetjük meg: funkcionális és nem-funkcionális követelmények.
   1. **Funkcionális követelmények:** A funkcionális követelmények tartalmazzák a leírást arról, hogy a rendszernek milyen funkciókkal kell rendelkeznie és ezek a funkciók hogyan kellene működjenek. Milyen inputra milyen output érkezik, technikai oldalról vizsgálódva.
   2. **Nem funkcionális követelmények:** A nem-funkcionális követelmények tartalmazzák a rendszer megszorításokat és a működési feltételeket. Ezen követelményekben a fejlesztésre vonatkozó leírásokat is megtaláljuk. Ez a rész drasztikusabb az előzőnél, ugyanis egy rosszul, vagy egyáltalán nem működő rendszert kapunk abban az esetben, amennyiben a feltételek nem teljesülnek. Ezen rész három főbb komponensre osztható: termék, szervezetei és külső követelményekre.
      1. **Termékhez kötött követelmények:** olyan dolgok tartoznak ide, mint például a teljesítmény, security, használhatóság stb.
      2. **Szervezethez kötött:** Környezeti, szervezeti, harcképességi stb.
      3. **Külső körülmények:** Etikai, biztonsági stb.



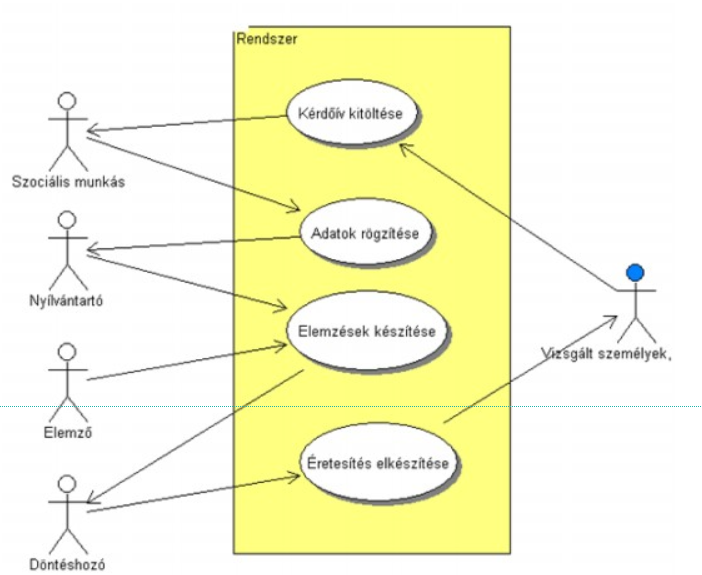
3.) **UML diagramok. Használati eset diagram (dinamikus kép). Osztály diagram (statikus kép).**

Az UML diagramok szoftver modellezéshez vannak kötve, hogy azok könnyebben implementálhatóak legyenek. Két nagy csoportra osztjuk ezeket: strukturális, illetve viselkedés diagrammokra. Az előző csoportba tartozik például az osztály diagram, a komponens diagram, az objektum diagram, a másodikba a Use-Case diagram, az aktivitási diagram, a szekvencia diagram, az idődiagram.

Ezek közül az osztály, illetve a Use-Case diagram a leggyakoribb.

**A Használati eset (Use-Case) diagram:** Ezen megközelítés az egyik leggyakoribb és leghasználtabb az UML diagram típusok közül, bemutatja a rendszert használó alanyokat, felhasználókat és azok szemszögéből mutatja be a rendszert. A különféle státusszal rendelkező szereplők számára elérhető funkcionalitásokat, illetve a kölcsönhatásokat is megmutatja. Remek kiindulópont lehet minden projekt esetében, ugyanis könnyen be lehet azonosítani általa a fontosabb szereplőket és a rendszer fő folyamatait. Alany állítmány jellege van.

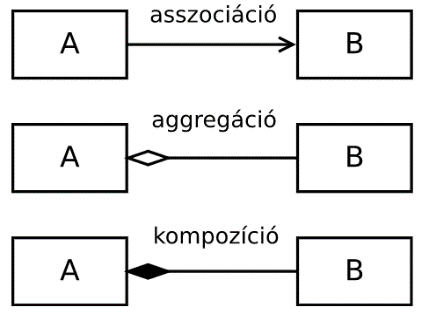
Use case dokumentálása is legalább annyira fontos, mint maga a diagram, két féle képpen lehet dokumentálni: természetes írott módon, mondatokkal, illetve táblázat formátumban, sablonokat használva.

Pl. 

**Osztály diagram:** Az objektumorientált programok tervezése jó esetben UML osztálydiagram rajzolásával, tehát papírral, ceruzával és radírral kezdődik, a legritkább esetben sikerül elsőre a tervezés, sokszor kell javítani, módosítani. Statikus, ki kivel van kapcsolatban.

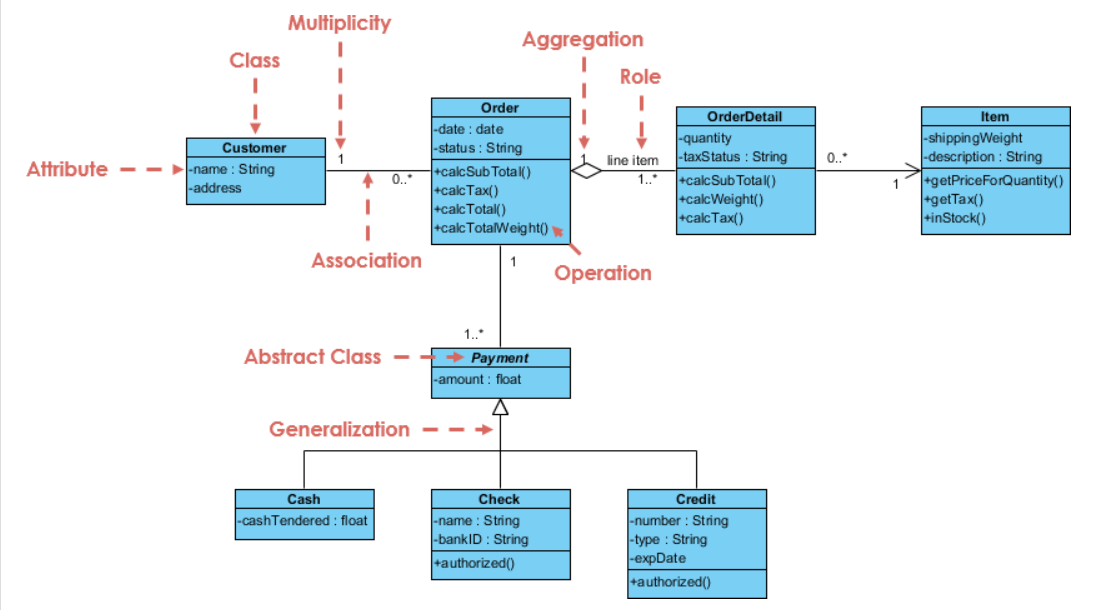
Deffiniciók, mik az osztály objektum stb itt is elmondani (OOP részből tudniillik)

**Jelölések:**

****

* Asszociáció: Gyenge ismeretség, azaz az objektum ismeri a másik létezését.
* Aggregáció: Bármi, ami a kettő között van, nem biztos, hogy annyira fontos, mint a kompozíció. Rész-egész. A rész túléli az egészet.
* Kompozíció: rész-egész viszonyt fejez ki, A objektumok tartalmaznak B-t. A rész nem éli túl az egészet.

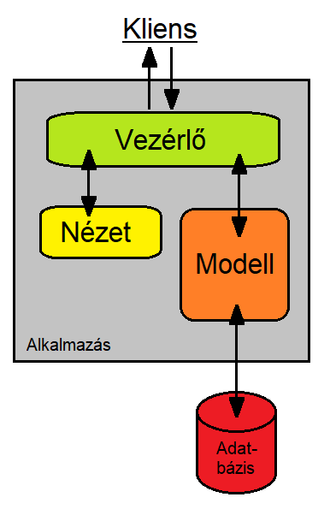
Az osztálydiagramm több osztályból áll, amelyek háromba osztott téglalapként vannak definiálva. A téglalap legfelső részében található az osztály neve, középütt az attribútumokkal, amelyek a private(-), public(+) és protected(#) kategóriába sorolhatók, úgyanúgy, mint a legalsó sorban található osztály metódusok. A függvények és attribútumok neve után kettősponttal van jelezve azok típusa.

Pl. 

**4.) Architekturális minták. Model-View-Controller architektúra. Előnyök és hátrányok.**

Az **architekturális**  minták nagyban hasonlítanak a design patternekhez, annyi különbséggel, hogy a rendszert ha tározzák, míg a másik egy bizonyos problémára ad megoldást. Ez magasabb szintű a design patterneknél, jól bevált architekturális minták léteznek, amelyek megoldást nyújtanak a problémákra, az egyik ilyen a MVC.

A **Model-View-Controller** az egy programtervezési minta. A lényege az, hogy az összetett, sok adatot a felhasználó elé táró alkalmazásban az adathoz (model) és a felhasnzálói felülethez (view) tartozó dolgokat szétválassza, hogy a felhasználói felület ne befolyásolja az adatkezelést, a UI felbolygatása nélkül történjen mindez. Az MVC ezt úgy éri el, hogy elkülöníti az adatok elérését és a logikát az adatok megjelenítésétől és a felhasználói interakciótól egy közbülső összetevő, a vezérlő bevezetésével.



**Model:** Az információk. Sok alkalmazás használ állandó tároló eljárást (mint pl adatbázis), azonban az MVC nem említi külön az adatelérést, ezt beleérti a modelbe.

**View:** A felhasználói felület.

**Controller:** Az eseményeket, jellemzően felhasználói műveleteket dolgozza fel és válaszol rájuk, illetve a modellben történő változásokat is kiválthat

1. A felhasználó valamilyen hatást gyakorol a felhasználói felületre (pl. megnyom egy gombot).
2. A vezérlő átveszi a bejövő eseményt a felhasználói felülettől
3. A vezérlő kapcsolatot teremt a modellel, esetleg frissíti azt a felhasználó tevékenységének megfelelő módon (pl. a vezérlő frissíti a felhasználó kosarát).
4. A nézet (közvetve) a modell alapján megfelelő felhasználói felületet hoz létre (pl. a nézet hozza létre a kosár tartalmát felsoroló képernyőt). A nézet a modellből nyeri az adatait. A modellnek nincs közvetlen tudomása a nézetről.
5. A felhasználói felület újabb eseményre vár, mely az elejéről kezdi a kört.

**Előnyei:**

* ***Egyidejű fejlesztés*** – Több fejlesztő tud egyszerre külön a modellen, vezérlőn és a nézeteken dolgozni.
* ***Magas szintű összetartás*** – MVC segítségével az összetartozó funkciók egy vezérlőben csoportosíthatóak. Egy bizonyos modell nézetei is csoportosíthatóak.
* ***Függetlenség*** – MVC mintában az elemek alapvetően nagy részben függetlenek egymástól
* ***Könnyen változtatható*** – Mivel a felelősségek szét vannak választva a jövőbeli fejlesztések könnyebbek lesznek
* ***Több nézet egy modellhez*** – Modelleknek több nézetük is lehet
* ***Tesztelhetőség*** - mivel a felelősségek tisztán szét vannak választva, a külön elemek könnyebben tesztelhetőek egymástól függetlenül

**Hátrányai:**

A MNV hátrányait általában a szükséges extra kódból adódnak.

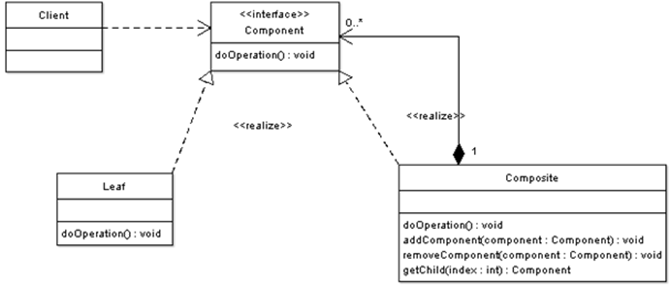
* ***Kód olvashatósága*** – A keretrendszer új rétegeket add a kódhoz ami megnöveli a bonyolultságát
* ***Sok boilerplate kód*** – Mivel a programkód 3 részre bomlik a ebből az egyik fogja a legtöbb munkát végezni a másik kettő pedig az MVC minta kielégítése miatt létezik.
* ***Nehezebben tanulható*** – A fejlesztőnek több különböző technológiát is ismernie kell az MVC használatához.

5.) **Tervezési minták. Összetétel (Composite), Egyke (Singleton), Megfigyelő (Observer) minták.**

Az informatikában a programtervezési mintának nevezik a gyakran előforduló programozási feladatokra adható általános, újrafelhasználható megoldásokat. Céljuk az, hogy leírást vagy sablont nyújtsanak. Segítik formalizálni a megoldást.

**Összetétel (Composite):**

Az **összetétel** minta az írja le, hogy az objektumok egy csoportját ugyanúgy kell kezelni, mint egy adott objektum példányait külön-külön. Az összetétel itt arra utal, hogy fa struktúrába szervezzünk objektumokat így reprezentálva a rész-egész hierarchiákat. Az összetétel minta lehetővé teszi, hogy a kliensek az önálló objektumokat és összetételeket egységes módon kezeljék.



A *Component* egy olyan interfész, amelyet mind az összetett osztálynak, mind a levélelemnek implementálnia kell. A *Leaf* osztály reprezentálja a tovább már nem osztható elemeket, míg a *Composite* osztály objektumai további komponensekből épülnek fel.

**Egyke (Singleton):**

Az **egyke** egy olyan programtervezési minta, amely egy objektumra korlátozza egy osztály létrehozható példányainak számát. Abban az esetben érdemes használni, amennyiben a rendszer működése rendellenes lenne, vagy össze is omolna abban az esetben, ha több objektumot is lehetne példányosítani. **Nem lehet publikus konstruktora**, csak akkor hasznos, ha egyetlen objektumra van szükségünk.

**Megfigyelő (Observer):**

A **megfigyelő** mintában egy objektum, melyet alanynak hívunk, listát vezet alárendeltjeiről, akiket megfigyelőknek hívunk és automatikusan értesíti őket bármilyen állapotváltozásról, többnyire valamely metódusuk meghívásán keresztül. Többnyire elosztott eseménykezelő rendszerek kialakításakor használjuk. A Megfigyelő minta kulcsfontosságú része az ismert Model-View-Controller (MVC, Modell-Nézet-Vezérlő) architekturális modellnek. A Megfigyelő mintát számos programozási könyvtár és rendszer alkalmazza, többek között szinte minden GUI toolkit.

A **megfigyelő minta akkor hasznos**, amikor egy objektum egy másik objektum állapotában érdekelt, és tudatában kell lennie (értesítést kell kapnia) az ebben bekövetkező változásokról. A megfigyelő mintában megfigyelőnek (*Observer*) hívják azt az objektumot, amely ellenőrzi egy másik objektum, az alany (*Subject*) állapotváltozásait.

