Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika

ŠTÁTNY PEDAGOGICKÝ ÚSTAV
NATIONAL INSTITUTE FOR EDUCATION

Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika

# Počítačové systémy 2

## Predmet: Počítačové systémy

**Línia: Vlastný odborový kontext informatiky a informatickej výchovy**

Operačný program VZDELÁVANIE

EURÓPSKA ÚNIA
Európsky sociálny fond

Európska únia
Európsky sociálny fond

Moderné vzdelávanie pre vedomostnú spoločnosť/Projekt je spolufinancovaný zo zdrojov ES

# Počítačové systémy 2

## Identifikácia modulu

| | |
|---|---|
| **Aktivita projektu:** | 1.2 Vzdelávanie nekvalifikovaných učiteľov informatiky na 2. stupni ZŠ a na SŠ |
| **Línia aktivity:** | Vlastný odborový kontext informatiky a informatickej výchovy |
| **Predmet:** | Počítačové systémy |

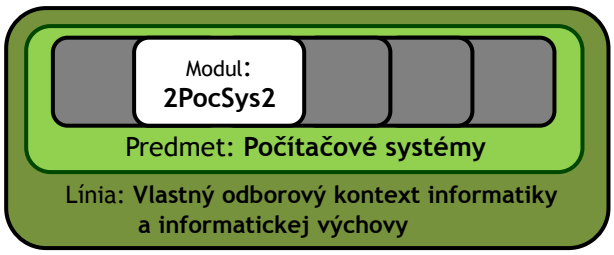
**Garant predmetu:**

RNDr. Peter Gurský, PhD.
ÚINF PF UPJŠ, Košice
peter.gursky@upjs.sk

**Autori:**

RNDr. Jozef Jirásek, PhD.
ÚINF PF UPJŠ, Košice

## Zaradenie modulu




Modul priamo nadväzuje na prvú časť predmetu Počítačové systémy. Účastníci ho absolvujú v treťom semestri vzdelávania. Zvyšné 3 moduly tohto predmetu sú naplánované na štvrtý semester vzdelávania. Tento modul využíva tiež poznatky získané v časti Základy hardvérového a softvérového vybavenia počítača predmetu Digitálna gramotnosť učiteľa.

Modul má e-learningovú podporu

## Abstrakt modulu

Modul priblíži účastníkom význam a činnosť procesorovej jednotky počítača na úrovni logických obvodov a úrovni riadenia strojovými inštrukciami.

Predstaví základný koncept strojového a inštrukčného cyklu a na príkladoch ukáže možnosti začlenenia programovania v strojovom kóde do vyššieho programovacieho jazyka.

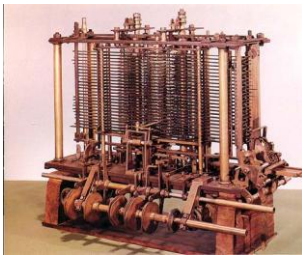
Priblíži proces spracovania strojovej inštrukcie a prístupy, smerujúce k jeho urýchleniu.

Poskytne prehľad o histórii a súčasnom stave používaných architektúr procesorov a načrtne možnosti ich ďalšieho rozvoja.

# Obsah

# Úvod

Informatika sa zaoberá uchovávaním, prenosom a spracovávaním informácií vo forme údajov. V tomto module sa bližšie pozrieme práve na spracovanie údajov, konkrétne na princípy automatického spracovania (**výpočtu**) prostredníctvom špeciálnych zariadení - **procesorov**.



*Torzo mlyna (mill) z analytického stroja (Babbage 1837) - prvého univerzálneho výpočtového systému*

Úloha procesorov je kľúčová hlavne pre aplikačné oblasti informatiky. Niektoré zásady organizácie ich práce a návrhy architektúr pre efektívnu komunikáciu s používateľmi sú známe a zachovávané už niekoľko desiatok rokov. Budeme sa nimi zaoberať aj preto, že tieto zásady by mali byť rešpektované aj pri ich využívaní.

Prvé mechanické zariadenia (pomôcky) na uľahčenie výpočtov boli rôzne mechanické kalkulátory. Zo súčasného pohľadu tvorili vlastne aritmetickú jednotku, ovládanú riadiacim prvkom – človekom. Komplexnejším návrhom organizácie výpočtu sa prvý v svojich prácach zaoberal Charles Babbage. V návrhu svojho analytického stroja (*Analytical Engine*) v polovici 19. storočia vyčlenil sklad (*store*) (s kapacitou 1000 čísel, uchovávaných v dekadickej reprezentácii, každé pozíciou 50 ozubených koliesok) a mlyn (*mill*) – kde sa systémom prevodov a ďalších ozubených koliesok prenášali postupne čísla zo skladu. Po "zomletí" – realizácii aritmetického výpočtu (+,-,*,/) - sa výsledky do skladu vrátili a do mlyna sa priviezli nové čísla. Na riadenie využil Babbage štítky, podobné štítkom na vytváranie vzorov pre tkáčske stroje. Prvýkrát použil aj podmienené skoky, čím sa jeho návrh stal prvým návrhom univerzálneho výpočtového systému (aj keď si to v tej dobe nikto neuvedomoval).

*Alan Turing (1912-1954)*

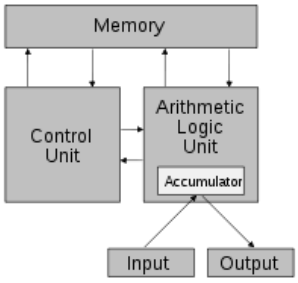
Možnosti realizácie algoritmov skúmali v prvej polovici 20. storočia A. Church, S. C. Kleene, E. L. Post, A. Turing. Vytvorili formálnu definíciu **vypočítateľnosti** a ukázali, že funkcie sú vypočítateľné práve vtedy, keď sú vypočítateľné na **Turingovom stroji**. Turing popísal aj "program" pre univerzálny Turingov stroj – ktorý vie (na základe vhodného očíslovania) zastúpiť prácu akéhokoľvek iného Turingovho stroja, čím dokázal, že je možné v princípe univerzálny počítač skonštruovať. Pokiaľ teda vieme pomocou nášho stroja zrealizovať všetky kroky univerzálneho Turingovho stroja, náš stroj je tiež univerzálny – vie realizovať všetky vypočítateľné funkcie. Súčasne boli nájdené aj problémy, ktoré na Turingovom stroji vypočítateľné nie sú, teda v zmysle predchádzajúcich tvrdení neexistuje algoritmus na ich riešenie.

Niektoré základné neformálne myšlienky a koncepty teórie zložitosti a Turingovych strojov môžete nájsť aj na wikipédii

http://en.wikipedia.org/wiki/Turing_Machine

Tieto fundamentálne výsledky tvoria základy informatiky a nadväzujú na ne viaceré ďalšie úvahy o mierach zložitosti a realizovateľnosti, prípadne optimalizácii riešenia problémov pomocou výpočtových strojov. Aj keď (bohužiaľ) uvedené poznatky nie sú súčasťou tohto vzdelávania, mali by byť v základnej výbave každého informatika (asi tak, ako sa predpokladá, že každý fyzik má aspoň základnú predstavu o teórii relativity).

Babbageova idea zápisu programu ako postupnosti riadiacich štítkov – čísel, idea podmienených skokov, oddelenie výkonnej, riadiacej a ukladacej časti stroja inšpirovala pri konštrukcii prvých funkčných univerzálnych počítačov (aj keď úplne prvý univerzálny elektromechanický (reléový) počítač zostrojil v roku 1941 Konrad Zuse, v tom čase v úplnej izolácii od svetovej vedy). Veľa myšlienok preformuloval a premyslel pri práci nad prvým elektronickým univerzálnym počítačom ENIAC (1946) aj John von Neumann a publikoval zásady pre konštrukciu počítačov, podľa ktorých sú konštruované počítače dodnes.




*Von Neumannova architektúra*

Von Neumannov koncept automatickej realizácie zložitých výpočtov predpokladá rozloženie výpočtu na jednoduché úkony (**inštrukcie**), ktoré sa spracovávajú **sekvenčne** (za sebou). Nasledujúce kroky sú stále závislé na predchádzajúcich a počítač nemôže zasahovať do poradia ich spracovania. Každý krok (inštrukcia) má svoj jednoznačný (binárny) **kód**, ktorý je zapísaný (spolu s údajmi – tiež v binárnej forme) v spoločnej pamäti. **Pamäť** je rozdelená na bunky rovnakej veľkosti a ich poradové čísla sa využívajú na jednoznačnú identifikáciu (adresovanie). Počítač okrem pamäte tvorí ešte **aritmeticko-logická jednotka**, **riadiaca** jednotka a **vstupno-výstupné jednotky**. Počítač je riadený kódom, ktorý je uložený v pamäti

(*stored-program concept*). Inštrukcie sa vykonávajú jednotlivo v poradí, v akom sú zapísané v pamäti. Zmenu poradia vykonávania inštrukcií je možné naprogramovať inštrukciami podmieneného a nepodmieneného skoku.

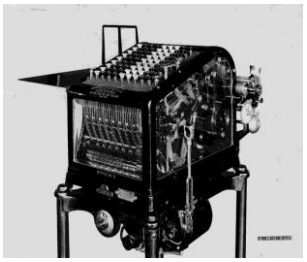
*John von Neumann (1903-1957)*

Princípmi von Neumannovej koncepcie a ich postupným prekonávaním sa ešte budeme časom zaoberať. Mnohé z nich boli (podobne ako Babbageove plány) v svojej dobe nerealizovateľné (napr. požiadavka ukladania programového kódu do pamäte, tvorenej elektrónkami). Vďaka prevratným zmenám v polovodičových technológiách sa ale von Nemannova koncepcia dostala do popredia a slúžila niekoľko desiatok rokov ako základ konštrukcie počítačov a procesorov.

V súčasnosti sa možno blížime k hraniciam jej možností, preto sa hľadajú (okrem paralelizácie a distribucie výpočtov) aj ďalšie alternatívne prístupy a koncepcie – niektoré z nich zmienime na konci modulu.

Realizáciu automatického zariadenia podľa von Neumannovych princípov ukážeme na platforme logických obvodov, ktorú sme si v základoch osvojili v prvom module. Ukážeme, ako je možné spracovávať údaje v binárnom tvare pomocou sekvencií jednoduchých logických funkcií. Budeme sa zaoberať vnútornou organizáciou výpočtu procesora a aj jeho celkovou vonkajšou architektúrou, prístupnou používateľom - programátorom.

Organizáciu práce procesora predstavíme na jednoduchom modeli riadenia údajového toku. V prostredí Logisim namodelujeme jednoduchú aritmeticko-logickú jednotku s možnosťami ukladania medzivýsledkov výpočtu do registrov. Navrhneme mechanizmus riadenia výpočtu pomocou riadiaceho slova. Postupne doplníme návrh o pamäť inštrukcií, pamäť údajov a možnosť realizovať podmienené skoky v mikroprograme. Predstaví sa základný údajový tok strojového cyklu. Účastníci získajú predstavu o časovej postupnosti vykonávania operácii a o taktovacom mechanizme.



*Mechanický kalkulátor z prvej polovice 20. storočia*

Počítače boli skonštruované hlavne na to, aby počítali. Aj slovo Computer sa používalo v 19. a začiatkom 20. storočia na označenie povolania, v ktorom ľudia (väčšinou ženy) počítali za pomocou mechanických kalkulátorov rôzne výpočty.

Ďalšia časť začína viacúrovňovým pohľadom na výpočtový systém a pokračuje charakteristikou strojovej inštrukcie a jej štruktúry – operačný kód, operandy, ich adresovacie režimy. Predstavia sa základné typy inštrukcií, spôsoby zmien riadenia v programe (skoky, podmienené skoky), význam príznakov a príznakového registra, práca so zásobníkom. Konkrétna realizácia bude ukázaná na prípade architektúry Intel x86. Možnostiam zápisu strojových inštrukcií v jazyku Assembler a jeho začleneniu do programu (v prostredí Lazarus) bude venované praktické cvičenie.

Potom bude predstavený vlastný priebeh spracovania strojovej inštrukcie. Ukážeme realizáciu inštrukčného cyklu pomocou viacerých strojových cyklov. Formou prezentácie s diskusiou sa ukážu aktuálne možnosti a problémy urýchlenia spracovania inštrukcií mechanizmom zreťazenia a paralelizácie (predspracovanie dekódovania inštrukcií, problémy so závislými parametrami, predikcia skoku, paralelné predspracovanie viacerých vetiev, využitie viacerých aritmeticko-logických jednotiek a viacerých jadier, možnosti spracovania inštrukcií „mimo poradia“, viacvláknové spracovanie, problémy synchronizácie a kolízie prístupov do spoločnej operačnej pamäte).

Na záver budú predstavené najčastejšie používané architektúry, porovnáme prístupy RISC a CISC návrhu a spôsoby porovnania ich výkonnosti.

Pre praktické cvičenia predpokladáme prístup účastníkov k počítaču s nainštalovaným prostredím Lazarus (samostatne alebo po dvojiciach) a tiež možnosti nainštalovania simulačného programu Logisim.

# Cieľ modulu

Získať konkrétnu predstavu o činnosti centrálnej procesorovej jednotky počítača. Uvedomiť si transformačné procesy medzi reprezentáciou informácie údajmi v registri a signálmi na prepojovacích zberniciach. Porozumieť konceptu údajových a riadiacich tokov, uvedomiť si dôležitosť správneho načasovania činností. Získať predstavu o činnosti procesora v strojovom cykle.

Priblížiť možnosti zápisu postupnosti strojových inštrukcií v jazyku Assembler a jeho integráciu do vývojového prostredia (napr. Lazarus). Porozumieť štruktúre strojovej inštrukcie a spôsobu realizácie inštrukčného cyklu pomocou strojových cyklov. Oboznámiť sa s možnosťami optimalizácie spracovania inštrukcií a vplyvu základných technických parametrov procesora na jeho výkon.

Získať prehľad o histórii a súčasnom stave používaných architektúr procesorov, možnostiach ich ďalšieho rozvoja a technologických hraniciach.

Počítače prebrali ľudskú prácu. Základným cieľom bolo zmechanizovať (neskôr zelektronizovať) manuálne vykonávané opakujúce sa výpočty.

Prvý elektrónkový univerzálny počítač ENIAC bol postavený na objednávku Laboratória pre balistický výskum americkej armády, ktoré zostavovalo delostrelecké tabuľky. Pred zostavením počítača bolo potrebných 20 hodín práce skúseného „počítača“ (počítačky) na ručný výpočet tabuľky pre 60 sekundovú trajektóriu strely. Po jeho nasadení trval tento výpočet už len 30 sekúnd.

ENIAC pracoval pol minúty nad úlohou pre človeka na 1200 minút. Ušetril teda prácu 2400 ľudí.

ENIAC bol schopný vypočítať 350 násobení za sekundu. Súčasný bežný kancelársky počítač urobí za ten istý čas cca 3,5 miliardy násobení. Je teda 10 miliónkrát rýchlejší. Pokiaľ ho vieme dobre využiť, urobí prácu za 24 miliárd ľudí (ani toľko nás tu niet).

Najrýchlejší superpočítač by to mal dnes ešte miliónkrát rýchlejšie :-)

# Vstupné vedomosti

## Predpokladané vstupné vedomosti, skúsenosti a zručnosti

Modul priamo nadväzuje na prvý modul predmetu Počítačové systémy, predpokladá orientáciu v pojmoch na jeho úrovni. Účastník by mal mať konkrétnu predstavu o funkcii jednoduchých logických obvodov, registra, jednoduchej pamäte a základnú predstavu o spôsobe sekvenčného spracovania inštrukcií v počítači.

Predpokladajú sa tiež určité skúsenosti s prácou v simulačnom prostredí Logisim (možno nájsť v elektronickej podpore kurzu resp. na adrese http://ozark.hendrix.edu/~burch/logisim/) a priemerné zručnosti a skúsenosti v programovaní vo vývojovom prostredí Lazarus.

## Preverenie vstupných vedomostí

Poslucháč by mal vedieť v prostredí Logisim namodelovať a otestovať realizáciu 8-bitového registra pomocou preklápacích sekvenčných obvodov RS.

# 1 Jednoduchý procesor – strojový cyklus

V predchádzajúcom module sme sa naučili, ako zapojiť jednoduché elektronické obvody tak, aby realizovali ľubovoľné logické funkcie. Pokiaľ teda nejaký výpočet vieme zapísať pomocou logických funkcií, môžeme ho tiež zrealizovať elektronickými obvodmi. Pre zložitejšie výpočty potrebujeme samozrejme viac obvodov a komplikovanejšie zapojenia. Súčasné mikroprocesory používajú zapojenia, kde sa počet aktívnych prvkov (tranzistorov) blíži k miliarde.

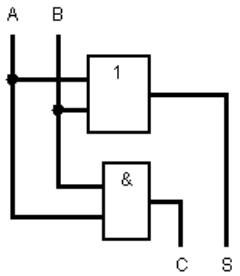
V tejto časti skúsime navrhnúť jednoduché zapojenie, ktoré by bolo schopné vykonávať jednoduché aritmetické a logické operácie na základe dopredu stanoveného postupu. Nepôjde o konkrétne zapojenie vyrábaného procesora, ale budeme na ňom môcť pozorovať niektoré jeho typické vlastnosti a budeme sa snažiť porozumieť princípom jeho činnosti.

## 1.1 Aritmeticko-logická jednotka (ALU)

Aritmeticko-logická jednotka ALU (*Arithmetic Logic Unit*) bude tá časť nášho procesora, kde sa budú uskutočňovať jednoduché aritmetické výpočty a logické operácie. Už v predchádzajúcej časti sme ukázali silu kombinačných obvodov, pomocou ktorých zvládneme realizovať ľubovoľné logické funkcie. Ako to bude s aritmetikou ?

Pri výpočte budeme samozrejme uvažovať dvojkový (binárny) zápis čísla. Úloha sčítania jednociferných binárnych číslel je jednoduchá. Pre možné štyri varianty vstupných hodnôt A a B vyznačíme v tabuľke výsledok (S – sum), prípadne či budeme signalizovať prenos do vyššieho rádu (C – carry).

| A | B | S = A XOR B | C = A AND B |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | **1** | 0 |
| 1 | 0 | **1** | 0 |
| 1 | 1 | 0 | **1** |



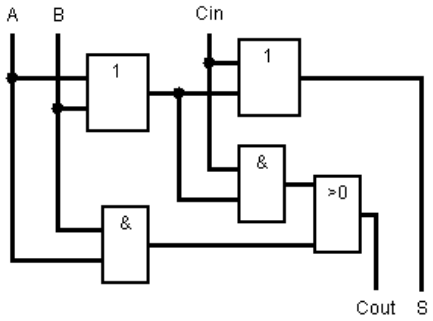

Z tabuľky pravdivostných hodnôt vidíme, že pre výstup S môžeme využiť obvod, ktorý počíta funkciu XOR (čo je vlastne sčítanie modulo 2). Úlohu vypočítať výstup C môžeme zveriť obvodu AND (na výstupe bude log. 1 len vtedy, keď na obidvoch vstupoch budú hodnoty log. 1). Na obrázku je aj schéma zapojenia takejto sčítačky. Nazýva sa **polovičná sčítačka** - HA (*half adder*).

Výpočet súčtu viaciferných čísel v dvojkovom zápise rozložíme na kroky – postupne budeme počítať súčty v stĺpcoch sprava doľava, pričom budeme musieť brať do úvahy tiež prenos, ktorý vznikol z predchádzajúcich rádov (uvedomte si, že pri sčítaní dvoch čísel nemôže byť väčší ako jedna).

Opäť môžeme zostaviť tabuľku pravdivostných hodnôt – tentokrát pre i-ty rád. Vstupné logické hodnoty budú bity i-teho rádu čísel A a B, ktoré chceme spočítať (označíme ich $A_i$ a $B_i$) a prenos $C_{i-1}$ z predchádzajúceho rádu. Výstupom bude súčet $S_i$ v ráde i a prenos $C_i$.

Pravdivostná tabuľka bude trochu zložitejšia. Výsledok $S_i$ je možné realizovať ako sčítanie modulo dva, teda opäť využijeme funkciu (a obvod) XOR tak, že $S_i = A_i$ XOR $B_i$ XOR $C_{i-1}$. Pre realizáciu prenosu $C_i$ musíme tabuľku analyzovať pozornejšie. Prenos je 1, ak platí súčasne A aj B, alebo platí len jedno z nich, a potom je prenos 1 len vtedy, keď ešte platí aj $C_{i-1}$. Čo môžeme zapísať $C_I = (A_i$ AND $B_i)$ OR $((A_i$ XOR $B_i)$ AND $C_{i-1}))$ a realizovať kombinačným obvodom na obrázku - tzv. **plnou sčítačkou** - FA (*full adder*).

| $A_i$ | $B_i$ | $C_{i-1}$ | $S_i$ | $C_i$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |




*Tabuľka pravdivostných hodnôt a zapojenie plnej sčítačky*

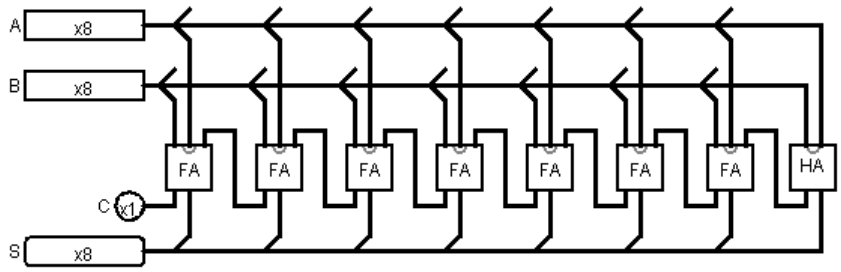
**Zadanie 1**

Zopakujte si ovládanie simulačného programu Logisim a nasimulujte plnú sčítačku pomocou základných logických členov. Otestujte jej funkcie.

V nastaveniach File-Preferences-International zvoľte zobrazenie Rectangular – teda zobrazenie obvodov podľa medzinárodnej (IEC) a európskej (EN 60617-12:1999) normy. Americký štandard ANSI pripúšťa ešte aj „neobdĺžnikové“ tvary (distinctive shape).

Ak chceme sčítať 8-bitové čísla, stačí použiť 8 jednoduchých sčítačiek. Jedna môže byť polovičná (v najnižšom ráde nepotrebujeme započítavať prenos), ostatné plné. Pomocou zapojenia na obrázku sa súčasne spracujú všetky signály binárnej reprezentácie čísla A a B, výstupné signály S sa (po určitom čase) ustália na logických hodnotách, ktoré odbpovedajú binárnej reprezentácii súčtu čísel A a B. Na výstupe $C_{out}$ dostaneme informáciu o prenose do vyššieho rádu, ktoré môžeme použiť v spojitosti s ďalšou sčítačkou alebo na signalizáciu toho, že výpočet asi nebude korektný, pretože výsledok nebolo možné reprezentovať daným množstvom výstupných bitov

Aby sme nemali v zapojení spleť prepojení, použijeme v Logisime splietanie vodičov (Splitter) – jedna čiara potom znázorňuje viac vodičov (nespojených), môžeme ich kedykoľvek rozpliesť.
Simulácia testuje už pri kreslení, či počty vodičov medzi prepojenými komponentami súhlasia. Na chybu je upozornené hlásením "Incompatible widths".

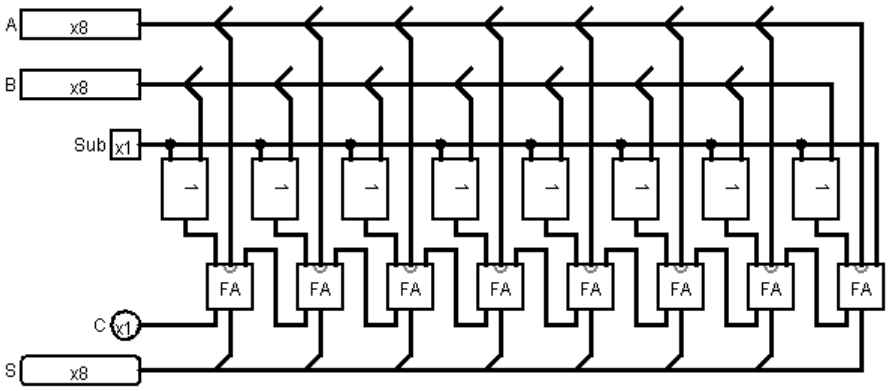



*8 bitová sčítačka*

Ako postupovať pri odčítaní? Pripomeňme si z predchádzajúceho modulu spôsob kódovania záporných čísel doplnkovým kódom. Pokiaľ zvolíme toto kódovanie, na odčítanie A – B = A + (-B) môžeme použiť našu sčítačku, keď predtým urobíme negáciu B a pripočítame jednotku (v doplnkovom kóde platí –B = (NEG B) + 1). Pripočítanie jednotky urobíme jednoducho tak, že polovičnú sčítačku nahradíme úplnou a ako vstupný prenos nastavíme log.1.

Či sa bude spočítavať, alebo odpočítavať, budeme nastavovať špeciálnym signálom Sub, ktorý pri hodnote log.0 nechá pracovať sčítačku, pri hodote log.1 zneguje vstupy B a pripraví hodnotu log.1 na vstupný prenos, čím zariadi, že výsledkom bude rozdiel A-B (v dvojkovom doplnkovom kóde).

Ak chceme realizovať sčítačku väčších čísel, oplatí sa zapojenie celého obvodu optimalizovať. Môžeme to urobiť napr. tým, že použijeme obvody s viacerými vstupmi a upravíme zapojenie (dá sa ukázať, že každý kombinačný obvod vieme zrealizovať len pomocou jednej negácie, konjunkcie a disjunkcie – takže sa to podarí na 3 kroky, neúmerne však vzrastie počet obvodov a počet ich vstupov)




*8 bitová sčítačka a odčítačka*

**Zadanie 2**

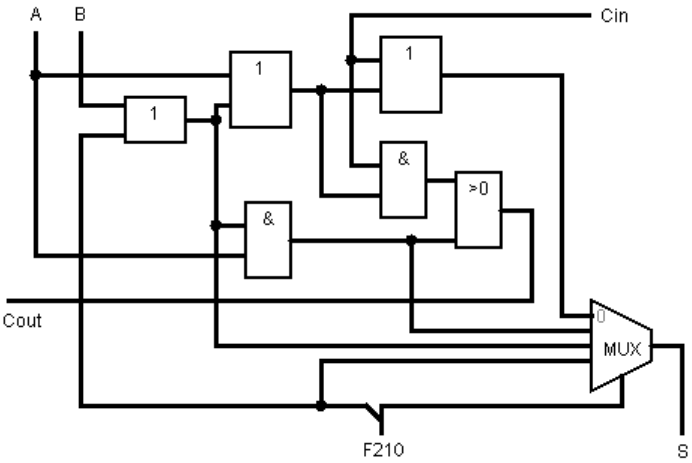
V prostredí Logisim nasimulujte uvedené zapojenie 8-bitovej sčítačky/odčítačky a otestujte jej funkčnosť. Obvody sú pripravené aj v e-learningovom prostredí v súbore PocSys2.circ

Kontrola správnosti výpočtu vzhľadom na aktuálnu interpretáciu premenných je vecou programátora. Procesor realizuje stále rovnaký výpočet – bez ohľadu na to, či programátor interpretuje typ výsledku ako integer alebo word ... Typová kontrola býva často ponechaná prekladaču, ktorý by mal pri preklade na možné problémy upozorňovať.

Všimnime si, že obvod je kombinačný – správny výsledok sa na výstupe objaví až potom, ako signály o prenose $C_i$ postupne aktualizujú výpočet na všetkých FA jednotkách. Ak čas na reakciu jedného FA obvodu je T, správny výsledok dostaneme po čase najviac T · (počet bitov sčítačky). Existujú samozrejme aj riešenia, pracujúce v kratšom čase, ktoré však obsahujú viac logických prvkov.

Pokiaľ vieme sčítať, s ostatnými aritmetickými operáciami si už poradíme (ukázali sme v predchádzajúcom module). Potrebujeme ešte nejaké logické operácie – budú sa vykonávať po jednotlivých bitoch spracovávaných slov. Postačí negácia NOT (pomocou nej a sčítania viem urobiť aj odčítanie) a jedna z operácii AND alebo OR (druhú tiež viem urobiť pomocou prvej a negácie a naopak).

Logické operácie potrebujeme hlavne pre prácu s informáciami na úrovni jednotlivých bitov. K tým nemáme priamy prístup - ALU vie pracovať len s celým binárnym slovom. Pre potreby premiestňovania údajov medzi ich dvoma umiestneniami sa pridáva operácia kopírovania vstupu na výstup.




*Jednobitový segment jednoduchej ALU*

Na obrázku vidíme, ako pridať ďalšie funkcie do jednobitového segmentu našej ALU. Vychádzame zo zapojenia segmentu plnej sčítačky (ktorú vidíme v pravej hornej časti). Jednotlivé funkcie budeme nastavovať pomocou signálov F0, F1, F2. Logické úrovne signálov F0 a F1 určia pomocou multiplexora, ktorý z jeho vstupov sa prenesie na výstup. Vstup F2 určí cez obvod XOR, či do sčítačky pôjde logická hodnota signálu B, alebo jej negácia.

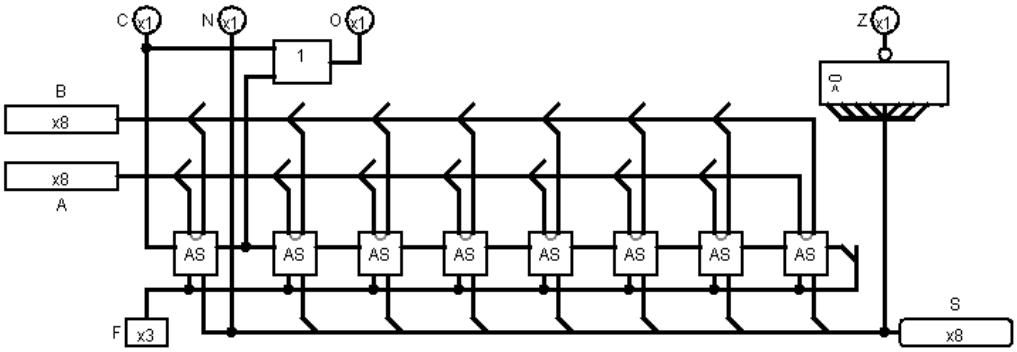
| F2 | F1 | F0 | S |
|---|---|---|---|
| 0 | 0 | 0 | A + B |
| 1 | 0 | 0 | A + (NOT B) = A – B - 1 |
| 0 | 0 | 1 | A AND B |
| 1 | 0 | 1 | A AND (NOT B) |
| 0 | 1 | 0 | B |
| 1 | 1 | 0 | NOT B |
| 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 |

Pokiaľ sú na vstupoch F0 a F1 úrovne log.0, pracuje obvod ako FA-plná sčítačka. Vstup F2 prípadnou modifikáciou vstupu B určí, či ALU bude realizovať sčítanie alebo odčítanie. V obidvoch prípadoch sa na základe vstupného signálu $C_{in}$ aktualizuje aj hodnota výstupného prenosu $C_{out}$.

V prípade, že F0=1 a F1=0, na výstup S sa dostane druhý zo vstupov multiplexora MUX. Ten bude mať v závislosti od stavu vstupu F2 hodnotu A AND B alebo A AND (NOT B). Prípad F0=0 a F1=1 (tretí vstup multiplexora) bude znamenať, že na výstupe S bude buď samotné vstupné B, alebo jeho negácia (vzhľadom k hodnote F2 – obvod XOR v tomto prípade funguje ako invertor signálu, riadený vstupom F2).

Posledná možnosť F1=1, F0=1, je využitá na rozšírenie výstupu podľa vstupu F2. Ak F2 je 0, dostanem na výstupe S=0, ak F2 je 1, výstupom budú (pre každý segment) jednotky.

K základným operáciám sa zvyknú pridávať operácie posunu (vyžadujú ďalšie prepojenia so susednými segmentami) – môžete ich vyskúšať implementovať namiesto posledných dvoch operácií 011 a 111. Zapojenie 8-bitovej ALU s navrhnutými segmentami bude vyzerať asi takto:




Aby sme dosiahli korektné odčítanie, musíme v spoločnom zapojení priviesť signál zo vstupu F2 tiež na vstup $C_{in}$ segmentu pre rád 0 (vpravo).

Dokážte, že k prekročeniu rozsahu pri sčítaní dochádza práve vtedy, keď sú výstupy C najvyššieho a druhého najvyššieho segmentu sčítačky navzájom rôzne.

Signál o tom, že pri výpočte sme dosiahli prenos aj na úrovni najvyššieho rádu (ktorý sme už nemohli nikde zaznamenať) prezentujeme výstupom C (carry). Pridali sme ešte ďalšie výstupné signály – **príznaky** (*flags*) výsledku operácie – v tomto pripade príznak N (*negative*) – že výsledkom operácie bolo záporné číslo (resp. najvyšší bit výsledku bol 1), príznak O (*overflow*) – že ak by sme interpretovali čísla v doplnkovom kódovaní, došlo k prekročeniu rozsahu (súčtom záporných čísel sme dostali kladné alebo súčtom kladných čísel záporné číslo), príznak Z (*zero*) – že výsledok bol 0.

Príznaky nastavuje ALU nezávisle od toho, ako v danom okamihu programátor hodnoty interpretuje. Programátor si môže vybrať, ktoré príznaky využije v ďalšom výpočte podľa svojich zvolenych typov reprezentácie.

**Zadanie 3**

Testujte jednoduché výpočty v simulovanej ALU v prostredí Logisim. Pozorujte, ako sa zmeny na vstupe okamžite propagujú na výstup obvodu.

## 1.2 Blok registrov, ukladanie priebežných výsledkov

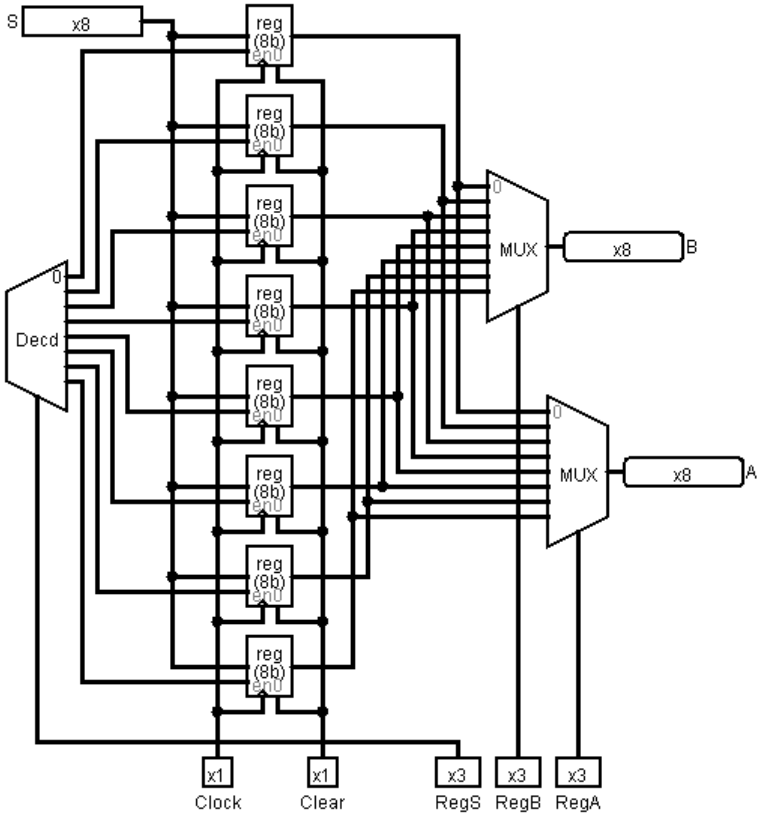
ALU je kombinačný obvod, ktorý mení výstup okamžite so zmenou vstupu. Ak chceme výsledky výpočtov využiť aj neskôr, musíme mať možnosť ich niekde uložiť. Tiež aj podľa potreby ich z ukladacieho priestoru vybrať.

Pri simuláciách môžete použiť 8-bitové registre, ktoré ste si pripravili z predchádzajúceho modulu.

Dočasné ukladanie vyriešime registrami. Pripomeňme si, že register je sekvenčný obvod, ktorý je schopný si zapamätať úroveň vstupných signálov v čase nábehu riadiaceho signálu Load (resp. Clock pre synchrónnu verziu) a tieto prezentovať potom na výstupe (až pokiaľ nepríde nový signál Load). Dá sa jednoducho realizovať pomocou obvodov RS.

Pokiaľ na vstup registra privedieme signály, odpovedajúce logickými úrovňami binárnemu zápisu čísla, register poslúži na jeho dočasné uchovanie, reprezentované stavmi príslušných RS obvodov. V registri si môžeme pamätať len jedno číslo. Uložením nového čísla sa starý stav obvodov RS stratí, už sa k nemu nemožno vrátiť.

V našom zapojení budeme využívať 8 osembitových synchrónnych registrov. Výber vhodného registra, ktorého obsah budeme chcieť spracovať v aritmeticko-logickej jednotke ako vstup A nastavíme signálnymi vstupmi RegA0, RegA1 a RegA2, pre výber registrov na vstup B použijeme signály RegB0, RegB1 a RegB2. Pre realizáciu výberu môžeme s výhodou použiť multiplexor.

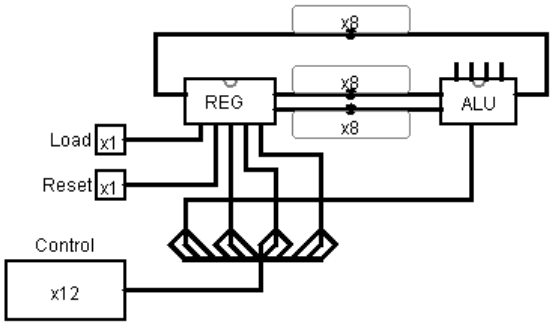



***Blok registrov** (register file) s možnosťami zápisu a čítania*

Dekodérom zabezpečíme, aby sa signál Enable (ktorý povoľuje zápis do registra) dostal len do jediného registra podľa stavu signálov RegS0, RegS1, RegS2. 8 bitový vstup S potom privedieme spolu so signálmi Clock (ktorý nábehovou hranou zapíše do registrov aktuálny stav vstupu S) a Clear (nastavenie všetkých registrov do stavu 0) ku každému registru v bloku.

Výstupy z bloku registrov napojíme na vstupy ALU a výstupy ALU zase na vstup do bloku registrov. Riadiace signály pre obe zariadenia (F pre ALU a RegA, RegB, RegS pre blok registrov) sústredíme na jednom mieste – odkiaľ bude prebiehať riadenie

celého **údajového toku** (*datapath*). Nastavením riadiacich signálov určíme, odkiaľ prídu údaje, čo sa s nimi má v ALU urobiť a kam ich potom uložiť.




*Zapojenie jednoduchého údajového toku medzi registrami a ALU*

Vnáraním do schémy – (pravým kliknutím na podobjekt) môžeme v prostredí Logisim sledovať stav úrovní signálov vo vnútri všetkých obvodov, ktoré sme vytvorili. Tu napr. môžeme vysledovať obsahy jednotlivých registrov pri výpočte

Aktuálne úrovne signálov možno sledovať počas simulácie objektom Probe (z knižnice Base).

Aby nedošlo k riziku, že vypočítaný údaj nahradí obsah v registri, ktorý je práve vybraný ako vstup do ALU (nový údaj by sa tak dostal do ALU, kde by opäť zmenil výsledok atď.), do registra zapisujeme až potom, keď sa výsledok ustáli. Zápis urobíme nábehom signálu Load. Nové hodnoty sú síce hneď spočítané, ale zápis je možný až novou nábehovou hranou signálu Load. Dovtedy môžeme modifikovať riadiace signály a zmeniť výpočet podľa našich predstáv.

**Zadanie 4**

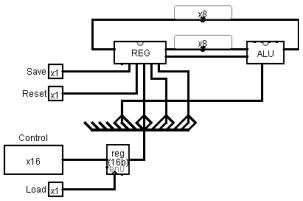
Vytvorte podľa obrázku (resp. vyberte z knižnice) zapojenie jednoduchého údajového toku (Datapath) a resetujte ho.

Nastavte operáciu (signály F) na 111 (jej výstupom sú samé jednotky) a výsledok zapíšte do registra R0 – kód 111000000000 (na výbere vstupných registrov nezáleží, vždy bude výsledok 11111111). Nábehom signálu Load zapíšte výsledok do registra. Jeho zmenený obsah sa ihneď prejaví aj na výstupoch A a B (testujte stav pomocou objektov Probe).

Vypočítajte rozdiel R1-R0 a uložte ho do registra R1 – kód 100001000001. Pretože 0-(-1)=1 dostaneme do R1 (po opätovnej voľbe signálu Load) jednotku.

Teraz môžem jednotku v R1 pripočítavať postupne k obsahu registra R2 (kód 000010010001). Keď už nebudeme meniť nastavenie riadiacich signálov, hodnota v R2 sa bude pri každej nábehovej hrane signálu Load zväčšovať o 1.

## 1.3 Riadenie údajového toku, riadiace slovo, pamäť inštrukcií




*Zapojenie údajového toku riadené obsahom registra*

Riadiace signály bude najvýhodnejšie uchovávať tiež v špeciálnom registri, niekedy sa označuje ako **riadiaci register** a jeho obsah je **riadiace slovo** (*control word*). Keď do registra zapíšeme číslo (riadiace slovo), dostaneme na výstupe signály presne v logických úrovniach, odpovedajúcich binárnym hodnotám zápisu príslušného čísla.

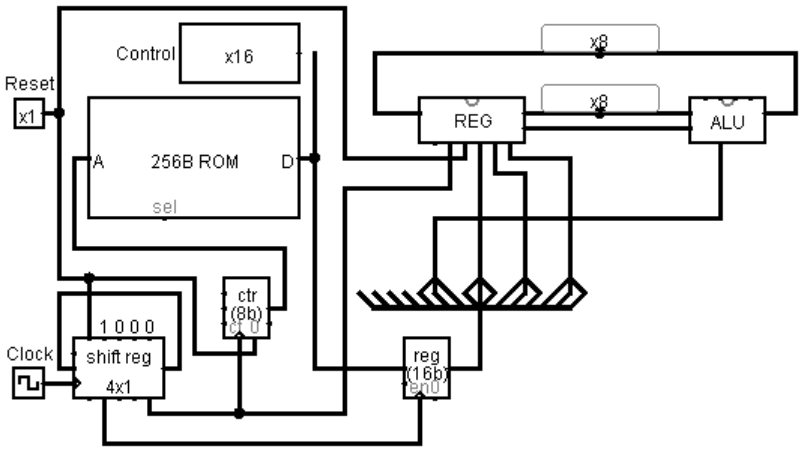
Aby sme nemuseli neustále zadávať riadiace slová ručne, môžeme ich mať pripravené v pamäti. Riadiace slová tam môžeme uchovávať ako obyčajné čísla v binárnom zápise. Túto pamäť budeme nazývať **pamäť inštrukcií** (*instruction memory*).

ROM pamäť je možné jednoducho obvodovo realizovať multiplexovaním riadkov matice invertorov – iné možnosti popíšeme v ďalšom module

Pre uchovanie riadiacich slov postačí pamäť typu ROM – Read Only Memory (teda len na čítanie). Aby sa celý výpočet zautomatizoval, použijeme na postupný výber z pamäte inštrukcií **počítadlo** (*counter*), ktorého hodnota sa bude postupne zvyšovať

Zapojenie počítadla pomocou RS sekvenčných obvodov sme spomínali v predchádzajúcom module. Môžete ho nájsť aj medzi štandardnými pamäťovými modulmi prostredia Logisim.

a privádzať na adresový vstup pamäte inštrukcií postupne prirodzené čísla, zväčšované o 1. Do riadiaceho registra sa takto budú postupne dostávať riadiace slová, zapísané na nasledujúcich miestach v pamäti inštrukcií.




*Programovateľný radič údajového toku*

ROM pamäť je možné nastavovať v simulačnom režime prostredia Logisim.

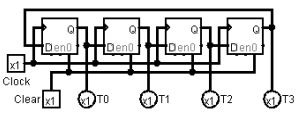
Na synchronizáciu výpočtu (aby všetko prebiehalo postupne za sebou a riadené obvody sa stíhali prepínať) použijeme generátor časových impulzov (z Base menu) a posuvný register.

**Časovač** (*clock*) je dôležitá súčasť každého sekvenčného stroja. Časovač strieda pravidelne úrovne log.0 a log.1. Pre súčasné mikroprocesory sa udáva frekvencia týchto zmien (*clock rate*) v rádoch GHz – čo znamená, že za jednu sekundu sa vygeneruje niekoľko miliárd časových impulzov (jednotka Hz udáva počet opakovaní cyklických zmien signálu za sekundu – rozmer $s^{-1}$). Zmeny v logickej úrovni signálu sa dajú využiť na postupné odštartovanie jednotlivých činností. Časovač teda posúva výpočet v čase a zabezpečuje dodržiavanie poradia jednotlivých krokov výpočtu. Najmenšie časové úseky, generované časovačom procesora, sa nazývajú **takty** *(clock cycle)*.

Pri frekvenciách nad GHz je trvanie jedného taktu menšie ako nanosekunda. (za jednu nanosekundu prejde svetlo vo vákuu vzdialenosť 30 cm, elektrický signál podľa typu vodiča cca 20 cm).

V každom takte sa väčšinou odštartuje vykonanie nejakej kombinačnej logickej funkcie. Nábehová hrana časovača slúži ako štartovací impulz taktu, ktorý najskôr urobí zmenu v stave systému (najčastejšie zápisom logických úrovní signálov do registrov), potom sa pokračuje vo výpočte kombinačnými obvodmi. Systém musí byť navrhnutý tak, aby sa každá elementárna činnosť ukončila v čase trvania jedného taktu (pre nesynchronizované kombinačné funkcie sa čas ich realizácie dá len odhadovať). Pre efektívne využívanie času je potrebné navrhovať elementárne činnosti (vykonávané v jednom takte) s približne rovnakým trvaním.

Pre pravidelne sa opakujúcu postupnosť jednoduchých krokov výpočtu môžeme výhodne využiť **posuvný register** (shift register). Posuvný register je register, v ktorom sa v každom kroku presúvajú stavy jeho RS obvodov jedným smerom. Využíva sa často na prevod paralelne uložených údajov (binárneho čísla v registri) na sériový signál (sekvenciu – postupnosť bitov zápisu tohto čísla) na výstupe, prípadne naopak, na prevod sériového signálu na paralelný údaj.




Jednoduché zapojenie kruhového registra

Pre naše potreby využijeme počiatočné nastavenie posuvného registra s jednou jednotkou (zvyšné hodnoty budú 0). Prepojíme sériový výstup registra s jeho vstupom, čím zabezpečíme posúvanie do kruhu – rotáciu. V jednoduchom zapojení s časovačom sa budú v časových cykloch (taktoch) striedať obsahy 4-bitového registra v poradí 1000, 0100, 0010, 0001. Využijeme ich na to, aby sme postupne spúšťali jednotlivé funkcie riadeného údajového toku.

Zatiaľ rozdelíme riadenie len na dva takty. V prvom zapíšeme aktuálny stav signálov D z pamäte inštrukcií do riadiaceho registra, čo aktualizuje riadiace signály údajového toku. Na základe aktuálnych riadiacich signálov sa vypočíta v ALU

príslušný výsledok a objaví sa na úrovniach výstupných signálov aritmeticko-logickej jednotky. Po ustálení výpočtu – v druhom takte – zapíšeme výsledok signálom Load do bloku registrov (do registra podľa nastavenia v riadiacom slove). Súčasne zvýšime hodnotu počítadla - čo spôsobí, že sa na výstupe pamäte inštrukcií nastavia logické úrovne signálu D podľa nového riadiaceho slova. Posuvný register v rotačnom zapojení v ďalšom takte znova signálom Load zapíše do riadiaceho registra nové úrovne z pamäte a celá činnosť sa bude opakovať.

Takto zapojený obvod bude automaticky vykonávať činnosti, ktoré sme robili v predchádzajúcom zadaní manuálne. Pokiaľ v pamäti inštrukcií nastavíme správnu postupnosť riadiacich slov (a dokážeme stroj vo vhodnej chvíli zastaviť), máme zariadenie, ktoré vie vypočítať ľubovoľný sekvenčný výpočet.

Pre testovacie účely použijeme samozrejme od začiatku manuálne ovládanie časovača. Keď budeme mať postup odladený, frekvencia časovača sa dá príslušnými parametrami nastaviť (Tick Frequency v záložke Simulate).

Cyklické opakovanie jednoduchých činností – taktov -(ktoré sme v našom prípade realizovali posuvným registrom) je typické pre jednoduchšie zapojenia procesorov. Postupnosť taktov jedného cyklu sa zvykne označovať ako **strojový cyklus** (*machine cycle*). Každé riadiace slovo má teda na svoje spracovanie k dispozícii práve jeden strojový cyklus.

**Zadanie 5**

Testujte jednoduché výpočty v simulovanej ALU v prostredí Logisim. Uložte do ROM pamäte postupnosť riadiacich slov z predchádzajúceho zadania a sledujte jeho vykonávanie.

## 1.4 Komunikácia s pamäťou údajov

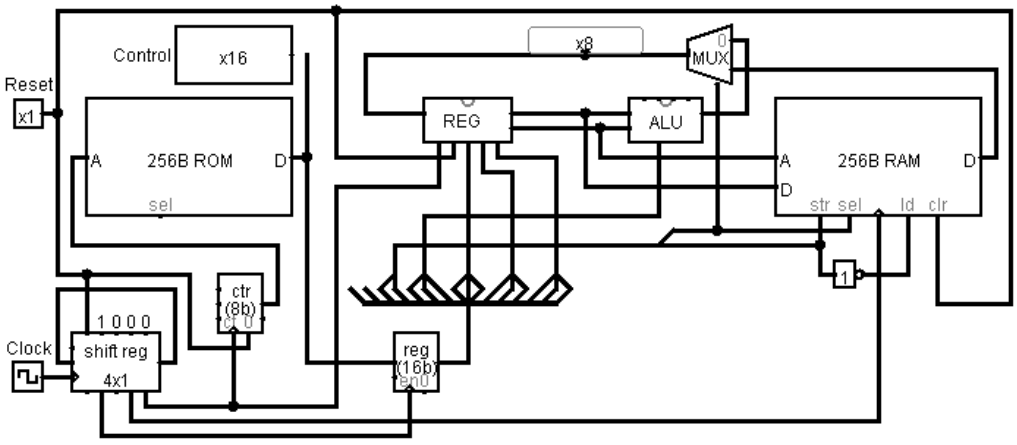
Využívanie registrov na spracovanie údajov väčšieho rozsahu by bolo neefektívne. Ak máme vstupných či výstupných (alebo aj pomocných) údajov viac, pomôžeme si pamäťou s možnosťou zápisu – RAM pamäťou.

Označenie RAM (Random Access Memory – pamäť s priamym prístupom) pochádza historicky z časov, kedy nie všetky typy pamäte umožňovali prístup ku všetkým údajom v rovnakom čase (napr. doba prístupu k záznamu na CD resp. magnetickom disku závisí od jeho umiestnenia).

Dnes sa označením RAM myslí skôr pamäť s možnosťou zápisu (RWM) a so závislosťou na napájaní (volatilná pamäť). Po vypnutí elektrického prúdu sa všetky údaje RAM pamäte stratia.

(v klasickom zmysle by RAM pamäťami boli aj flash a mnohé ROM pamäte)

V prostredí Logisim si môžeme vybrať RAM pamäte s obojsmernou údajovou zbernicou (ktorá sa často používa pre veľkú – hlavnú – operačnú pamäť) alebo pamäť s oddeleným zapisovacím a čítacím tokom. Oddelené toky sa jednoduchšie ovládajú, no potrebujú dvojnásobne viac prepojení. V prípade zapojenia procesora si to môžeme dovoliť, použijeme teda na ukladanie údajov pamäť RAM s oddeleným zápisom a čítaním (tak ako sme ju navrhli na záver predchádzajúceho modulu).




*Programovateľný radič údajového toku s prístupom do **údajovej pamäte***

Najjednoduchším riešením je pridanie prístupu do pamäte priamo do riadeného údajového toku. Signály, privedené na vstup A ALU, pripojíme tiež na adresové vstupy údajovej pamäte RAM, signály zo vstupu B na údajové vstupy pamäťovej matice. Výstupné údajové signály z pamäte musíme multiplexovať s výsledkami ALU jednotky. Budeme sa teda musieť rozhodnúť, či v danom strojovom cykle budeme počítať v ALU, alebo pristupovať do pamäte.

Potrebujeme aj dva ďalšie riadiace signály do riadiaceho slova. Jeden bude signalizovať, že aktuálny strojový cyklus sa použije na prístup do pamäte - ten súčasne pripojíme na vstup Select pamäte, ktorý ju uvádza do prístupného stavu (v našom minulom zapojení vstup Enable). Druhý riadiaci signál určí, či budeme do pamäte zapisovať, alebo z nej čítať. Ten pripojíme na vstupy str (Store) a ld (Load) pamäte (v našej sme už mali jednobitový signál R/W).

Prístup do pamäte vyžaduje väčšinou dosť času (v starších typoch počítačov sa pri prístupe k pomalším pamätiam používali tzv. čakacie takty). Aby sme zabránili aj iným hazardným stavom (v čase zápisu riadiaceho slova ešte nie je známa hodnota, ktorá sa má zapísať do pamäte), potrebujeme na prístup do pamäte zvláštny takt, ktorý zaradíme medzi zápis riadiaceho slova a zápis výsledku.

Strojový cyklus bude teda pracovať v troch taktoch – v prvom sa uloží riadiace slovo do riadiaceho registra a prebehne výpočet v ALU, v druhom (ak je to požadované) sa zapíše do údajovej pamäte alebo pripraví údaj z miesta signalizovaného na jej adresovom vstupe, v treťom sa podľa typu operácie zapíše do výsledného registra výsledok ALU operácie alebo prečítaný údaj z pamäte.

**Zadanie 6**

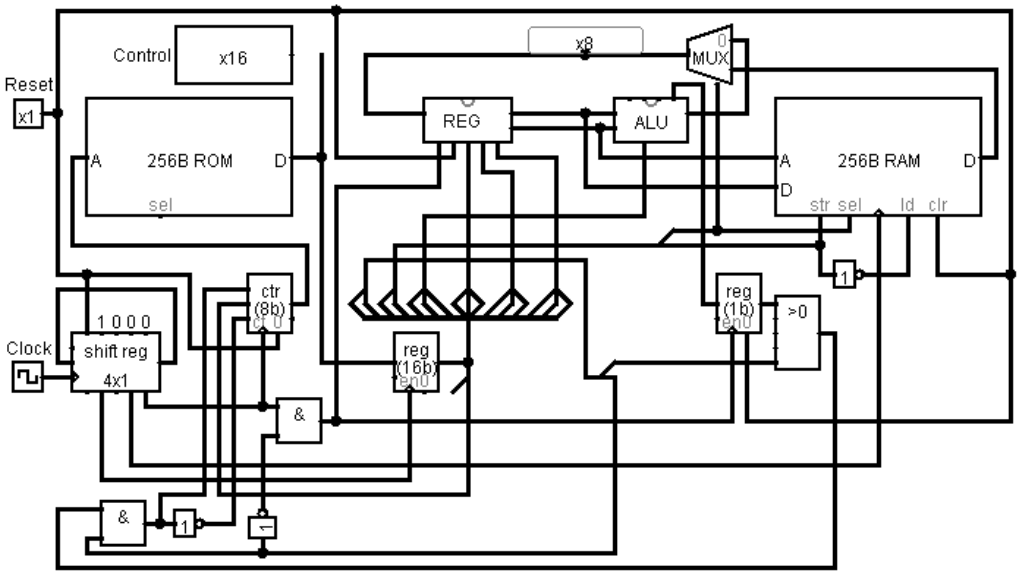
Pripojte k programovateľnému radiču údajovú pamäť podľa obrázku (alebo vyberte z knižnice obvod Datapath+im+dm). Navrhnite a uložte postupnosť riadiacich slov, ktoré vypočítajú súčet čísel, uložených v údajovej pamäti na miestach 0 a 1 a výsledok uloží na miesto 2.

Simulujte výpočet, pomaly prechádzajte jednotlivými fázami a sledujte priebeh spracovávaných údajov.

## 1.5 Skoky v programe, univerzálny výpočet.

Minimálna množina inštrukcií na realizáciu univerzálneho výpočtu je pre registrové počítače napr. pripočítanie a odpočítanie jednotky a skok v prípade nulového výsledku;
resp pripočítanie jednotky, mazanie registra a skok v prípade rovnosti obsahov registrov ...

Aby sme mohli realizovať akýkoľvek algoritmický výpočet (v zmysle definície algoritmu podľa Turinga a Churcha), chýba nám realizácia skokov (špeciálne podmienených skokov). Využijeme signály o príznakoch výsledku výpočtu ALU - pre jednoduchosť vezmeme len jeden príznak – Z (zero), ktorý uložíme vo fáze ukladania v špeciálnom **príznakovom registri** (*flag register*).




*Programovateľný radič s možnosťou realizácie podmienených skokov*

K riadiacim signálom pridáme jeden, ktorý bude signalizovať, že strojový cyklus realizuje skok, druhým signálom rozhodneme, či skok sa má uskutočniť za

akýchkoľvek okolností (bezpodmienečný skok), alebo skok bude závisieť od stavu príznaku Z, uloženého v príznakovom registri.

V prípade, že sú splnené podmienky pre skok, do počítadla sa nastaví číslo, ktoré tvorí 8 dolných bitov riadiaceho slova (príslušná aritmetická funkcia sa vykoná tiež, ale ďalším obvodom zabezpečíme, aby v poslednom takte (ukladaní výsledku) sa neaktivoval ani blok registrov, ani zápis do príznakového registra.

**Zadanie 7**

Pripojte k programovateľnému radiču s pamäťou rozšírenie o riadenie toku údajov podmieneným skokom (alebo vyberte z knižnice obvod Datapath+im+dm+jmp).

Navrhnite a uložte postupnosť riadiacich slov, ktoré vypočítajú súčin dvoch čísel, uložených v údajovej pamäti (môžete použiť riešenie, ktoré nájdete cez elektronickú podporu kurzu – nahrajte ho do inštrukčnej ROM pamäte).

Ukončenie programu môžete zrealizovať mikroinštrukciou skoku na miesto, kde sa táto nachádza – dostaneme nekonečnú slučku, v ktorej bude procesor čakať.

Simulujte výpočet a sledujte jeho priebeh a riadenie jednotlivých krokov.

Dopracovali sme sa k návrhu jednoduchého zariadenia, ktoré je schopné automaticky realizovať akékoľvek algoritmické postupy. Komplexnejšie návrhy pridávajú k tejto schéme ďalšie výkonné prvky – jednotku pre prácu s číslami v pohyblivej rádovej čiarke, grafické jednotky pre rýchly výpočet obrazu, jednotku riadenia vstupno-výstupných zariadení a pod. O niektorých z nich budeme hovoriť v ďalších častiach. Zariadenie tohto druhu nazývame všeobecne **radič** (*controller*). V prípade, že riadenie sa robí pomocou riadiacich slov, uložených v riadiacej pamäti, nazývame toto zariadenie **programovateľný radič** (na rozdiel od obvodového radiča – radiča bez pamäte). Procesor je teda špeciálnym typom programovateľného radiča a jeho riadiace slová sa nazývajú aj **mikroinštrukcie** a tvoria **mikroprogramy**. Mikroprogramovanie je špecifická činnosť, ktorá sa realizuje len pri návrhu procesora. Mikroprogram je uložený v procesore a väčšinou ho nie je možné zmeniť.

## Čo sme sa naučili

Navrhli sme jednoduchú aritmeticko-logickú jednotku s niekoľkými aritmetickými a logickými operáciami a ukázali sme, ako je možné tento výpočet riadiť pomocou riadiaceho slova.

Princípom riadenia výpočtu je uloženie riadiaceho slova do registra, kde sa jeho jednotlivé bity (v binárnom zápise) prejavia ako riadiace signály príslušných logických úrovní, ktoré nastavia výkonné jednotky tak, aby v jednom strojovom cykle realizovali krok výpočtu.

Zaoberali sme sa možnosťami rozdelenia zložitejšieho výpočtu na jednoduchšie kroky, o možnosť uchovania riadiacich slov (mikroinšrukcií) pre tieto kroky v pamäti a ich postupného automatického vykonávania.

Ukázali sme aj jednu z možnosti realizácie skokov a podmienených skokov v mikroprograme, ktoré umožňujú využiť mikroprogramovateľný radič ako procesor pre vykonanie ľubovoľnej algoritmicky riešiteľnej úlohy.

# 2 Strojové inštrukcie

Výpočtový proces, prebiehajúci v procesore, je komplexná činnosť. Aby sme ho vedeli presnejšie pozorovať, popísať, riadiť, prípadne navrhnúť alternatívne riešenia, je výhodné sa na neho dívať z rôznych úrovní.

V prvej časti sme sa venovali organizácii výpočtu vnútri procesora. Získali sme základnú predstavu o možnostiach stavby procesora pomocou logických obvodov. Skúsme sa teraz pozrieť na jeho vonkajšiu architektúru – aké služby poskytuje používateľovi (programátorovi), aké sú pravidlá jeho využívania, spôsoby komunikácie, jeho ohraničenia.

V úvode tejto kapitoly si predstavme viacúrovňový model výpočtového systému (*multilevel machine* podľa [9]), ktorého súčasťou je aj pohľad na architektúru systému. Tento model umožňuje oddeliť problémy jednotlivých vrstiev a riešiť ich v ohraničenom kontexte

## 2.1 Úrovňový model architektúry výpočtového systému

Úrovňový model vychádza z popisu výpočtového systému na rôznych stupňoch abstrakcie. V každej úrovni je možné vytvoriť vlastný model výpočtu - virtuálny výpočtový systém – v ktorom môžeme riešiť problémy nezávisle na realizácii ostatných vrstiev úrovňového modelu.

Začali sme v predchádzajúcom module tou najnižšou úrovňou architektúry – fyzickou realizáciou **spínacích prvkov**. Zaoberali sme sa historickými prístupmi, využívajúcimi elektromagnetické relé, elektrónku, tranzistor, integrovaný obvod. Táto úroveň pohľadu zaujíma najmä technológov, ktorí riešia problémy používaných materiálov, miniaturizácie základných aktívnych prvkov, možnosti ich vodivého prepojenia na minimálnej ploche. Pokiaľ by bol objavený iný materiál, či princíp výroby spínačov (napr. na úrovni kvantových fyzikálnych efektov, či biologických spínačov na úrovni DNA reťazcov), celá pyramída by mohla ostať zachovaná – vymenila by sa len jej spodná vrstva.

Pokračovali sme návrhmi jednoduchých aj zložitejších logických obvodov na realizáciu logických funkcií. Problematika efektívneho návrhu **logických obvodov** v podstate nezáleží na tom, z čoho budú vyrábané (ako vyzerá najnižšia vrstva). Aj v našich úvahách sme riešili problémy len na úrovni "poprepájaných obdĺžnikov". Úlohou tejto vrstvy je zapojenia optimalizovať vzhľadom k zadaným kritériám – minimalizovať počet jednoduchých prvkov v návrhu, minimalizovať čas, potrebný na propagáciu zmien, minimalizovať počet prekrížení prepojovacích vodičov, optimalizovať cenu riešenia.

| |
|---|
| Vývojové prostredia, middleware |
| Programovacie jazyky |
| Volania služieb operačného systému |
| Strojové inštrukcie, ISA model |
| Riadiace slová, mikroarchitektúra |
| Logické obvody |
| Technológia spínacích prvkov |

*Úrovňový model architektúry výpočtového systému*

Vo vrstve **mikroarchitektúry** systému sme sa pohybovali v prvej časti tohoto modulu. Priblížili sme si základné problémy realizácie údajových tokov a ich riadenia, možnosti mikroprogramovania na úrovni riadiacich slov. Úlohou tejto vrstvy je efektívne spracovať postupnosť strojových inštrukcií (dodanej vyššou vrstvou) – jednoduchých úloh - za pomoci aktuálnej konfigurácie logických prvkov. V súčasných typoch procesorov nie je možné vykonať každú inštrukciu pomocou jedného riadiaceho slova. Riešia sa problémy súčasnej práce na viacerých inštrukciách, rozdeľovanie čiastkových úloh pre viacero výkonných jednotiek rôzneho typu,

problémy predpovedania výpočtu, prípravy viacerých variantov postupu, spolupráca silne spriahnutých procesorov. V záverečných kapitolách tohoto modulu sa ešte k týto problémom vrátime.

Vyššia vrstva poskytuje pohľad na architektúru výpočtového systému na úrovni **strojových inštrukcií** (ISA – *Instruction Set Architecture*) Tento stupeň abstrakcie je vhodný pre programátorov v strojovom jazyku. Poskytuje informáciu o funkčnosti systému z používateľského pohľadu, o spôsobe uchovávania údajov, obmedzeniach, ktoré má systém a pravidlách, ktoré by mal používateľ (programátor) dodržiavať. Táto úroveň je na rozhraní medzi hardvérovou realizáciou a softvérovými možnosťami využitia výpočtového systému.

Nad vrstvou strojových inštrukcií sú jednoduché programy **operačného systému**, ktoré sprístupňujú ovládače vstupno-výstupných zariadení, riadia využívanie spoločnej pamäte a procesora. Používateľ má možnosť programovať tieto ovládače sám, no býva to väčšinou náročné a boli by určené len pre konkrétne hardvérové konfigurácie. Používaním funkcií (služieb) operačného systému sa dosiahne kompatibilita programového vybavenia medzi počítačmi s tým istým operačným systémom. Používateľským rozhraním k tejto vrstve je sústava volaní služieb operačného systému, ktorá je dostupná používateľom na vyšších vrstvách. Jednou z týchto služieb je aj preklad (či interpretácia) z programovacieho jazyka do jazyka strojových inštrukcií, ktorým táto vrstva prenáša úlohy vyššej úrovne k ich realizácii na nižších úrovniach.

Väčšina programátorov pristupuje k výpočtovému systému z úrovne **programovacieho jazyka**. Programovací jazyk obmedzuje programátora na využívanie konkrétnej množiny príkazov a údajových štruktúr. Sprístupňuje mu (niektoré) funkcie operačného systému a chráni pred ostatnými používateľmi.

Prekladač preloží (prípadne len interpretuje) príkazy vo vyššom jazyku do jazyka strojových inštrukcií (strojového kódu), prípadne volaní systémových služieb

Nad programátorským pohľadom býva niekedy ďalšia vrstva, ktorá môže zjednocovať pohľad na spoločné požívané zdroje, distribuuje výpočty (typicky medzi viacerými výpočtovými systémami), stará sa o komunikáciu a zdieľanie zdrojov.

Aplikačné programy potom tvoria priame rozhranie medzi používateľom a výpočtovým systémom a prispôsobujú interakciu a využívanie funkcií na rôznych úrovniach modelu bez znalosti konkrétnych jeho architektúr.

Špeciálnym prípadom jazyka je aj Assembler

Rozdelenie do popísaných úrovní je v poslednom čase veľmi aktuálne hlavne v súvislosti s konfigurovaním virtuálnych strojov, virtuálnych operačných systémov, virtuálnych používateľských prostredí, virtuálnych ISA architektúr. Typickým príkladom je definovanie vlastnej ISA úrovne JVM (Java Virtual Machine) pre programy v jazyku Java. Adaptáciou JVM na konkrétnu fyzickú architektúru získame možnosť využívať všetky aplikačné programy, napísané v Jave bez nutnosti ich opätovného prekladu. Ďalším príkladom sú procesory firmy AMD, ktoré udržujú ISA úroveň kompatibilnú s architektúrou Intel nad vlastnou mikroarchitektúrou s vlastnou organizáciu výpočtu.

## 2.2 Model výpočtu na úrovni strojových inštrukcií

Prvotné počítače boli schopné zvládať realizáciu svojich inštrukcií priamo pomocou riadiacich slov (ako sme ukazovali v prvej časti modulu).Zavádzaním nových inštrukcií ale stúpali nároky na realizáciu a boli už z hľadiska komplikovanosti riadiacich slov ťažko realizovateľné.

Úroveň strojových inštrukcií sa teda oddelila od úrovne mikroinštrukcií (resp. riadiacich slov), ktorej ostala vnútorná realizácia výpočtu strojových inštrukcií, resp. v súčasnosti mnohé iné prvky, skryté pred očami programátora. Pre programátora ostáva pohľad na výpočtový systém ako realizáciu sekvenčnej postupnosti inštrukcií, pričom na nižšej úrovni sa výpočet riadi inými postupmi (a ako budeme spomínať, v niektorých prípadoch môže dokonca prehadzovať poradie vykonávania inštrukcií).

V minulosti sa údajová štruktúra, s ktorou bolo možné pracovať v ALU, nazývala **slovo** (word).

Dnes word označuje často dva bajty (bez ohľadu na architektúru).

**ISA** (*Instruction Set Architecture*) **architektúra** zahŕňa formát a reprezentáciu údajov, registrový model, inštrukčný súbor, spôsoby adresácie, pamäťový model, adresové konvencie, riadenie behu, stavy počítača, vstupy a výstupy.

**Formát a reprezentácia údajov** – v prevažnej väčšine súčasných počítačov sa používajú údaje v binárnej forme. Celočíselné údaje sú interpretované len v ohraničenom rozsahu – je obmedzený šírkou spracovania v aritmeticko-logickej jednotke (počtom bitov ALU) resp. veľkosťou pracovných registrov. Podľa šírky spracovania aritmetiky rozlišujeme 8-bitové, 16-bitové, 32-bitové resp. 64-bitové procesory.

**Bit** – skratka od Binary digit – binárna číslica.

**Bajt** (*byte*) je usporiadaná postupnosť ôsmich bitov. Označenie bolo navrhnuté a všeobecne rozšírené v IBM System/360 architektúre (1956). (snáď ako rovnako znejúce slovo k *bite*)

Pre štvoricu bitov je štandardné označenie *Nibble*.

Pre záporné čísla sa štandardne používa dvojková doplnková reprezentácia, ktorá (ako sme ukázali) nevyžaduje špeciálne modifikácie ALU. ALU pri výpočte postupuje stále ako s nezápornými číslami, ale pripraví informácie o prípadných chybách v rozsahu pre bezznamienkovú i znamienkovú reprezentáciu. Programátor si vyberie, ktorý príznak bude akceptovať. Čísla s pohyblivou rádovou čiarkou sa kódujú väčšinou podľa normy IEEE-754, výpočty sa vykonávajú v zvláštnej aritmetickej jednotke so zvýšenou presnosťou, medzivýsledky ostávajú v špeciálnych FP (*floating point*) registroch.

Údaje sú uložené predovšetkým v **operačnej pamäti** (*main memory*) s priamym prístupom (RAM), ktorú na tejto úrovni chápeme ako usporiadané pole bajtov, očíslované adresami obyčajne od 0 do $2^n$-1 (kde n je veľkosť adresovej zbernice – v súčasných typoch procesorov 32 bitov a viac). Bajt je najmenšia adresovateľná časť pamäte. Pokiaľ chcem meniť iba jednotlivé bity, musím prečítať celý bajt, bit zmeniť a opäť zapísať celý zmenený bajt.

Označenie Little a Big endian vzniklo ako asociácia na knihu J. Swifta Gulliverove cesty, konkrétne na vojnu medzi Lilliputom a Blefuskom o to, či sa majú rozbíjať vajíčka z užšieho alebo zo širšieho konca.

Viacbajtové čísla (resp. iné viacbajtové údajové štruktúry) môžeme do pamäte ukladať principiálne dvoma spôsobmi: od najnižšieho bajtu (**Little endian**) – používaný v architektúrach Intel alebo od najvyššieho bajtu (**Big endian**) - používal sa v klasických sálových počítačoch, v súčasnosti napr. v procesoroch SUN Sparc, IBM PowerPC, Motorola. Napríklad číslo v hexadecimálnom tvare 0x12345678h sa zapíše v Big endian tvare v postupnosti bajtov 12 34 56 78 a v Little endian tvare postupnosťou 78 56 34 12. Najnovšie konštrukcie procesorov umožňujú nastavovanie endianov softvérovo.

*Vajíčko v Little-endian pozícii*

Okrem údajov býva v pamäti uložený tiež program - teda jeho strojový kód (v súlade s von Neumannovou koncepciou). Alternatívna voľba – oddelenie pamäte inštrukcií od pamäte údajov (tzv. Harvardská architektúra) má výhody v rozložení komunikácie na dva nezávislé kanály, lepšie je tiež ochrana kódu pred prepísaním (inštrukčná pamäť je chránená na zápis). Nevýhodou je nemožnosti zdieľať efektívne nevyužitý pamäťový priestor. V súčasnosti sa prikláňa k tzv. **modifikovanej Harvardskej architektúre**, kde sa vytvára virtuálne oddelenie pamäťových priestorov pre kód a pre údaje pomocou segmentácie a stránkovania (v skutočnosti na úrovni mikroinštrukcií dnes Harvardská architektúra prevláda – aj my sme ju v návrhu v prvej kapitole použili).

Často používané údaje a priebežné medzivýsledky výpočtov ukladáme do pracovných **registrov**. Registre, prístupné používateľovi, sú jednoznačne identifikovateľné a charakterizované šírkou v bitoch. Rozoznávame **univerzálne registre**, ktoré je možné použiť na ľubovoľné účely a **špeciálne registre,** ktoré využíva procesor na špecifické činnosti.

Riadenie vykonávania programu sa organizuje pomocou špeciálneho registra, označovaného všeobecne **PC register** (*Program Counter* - programové počítadlo resp. ukazovateľ miesta programu, kde sa práve nachádzame). Obsahuje v každom okamihu adresu miesta v operačnej pamäti (resp. inštrukčnej pamäti), kde sa nachádza kód inštrukcie, ktorú práve vykonávame alebo chceme vykonať.

Po prečítaní kódu inštrukcie sa automaticky zvýši obsah PC registra, teda potom obsahuje adresu miesta v operačnej pamäti za už prečítaným kódom. To umožní

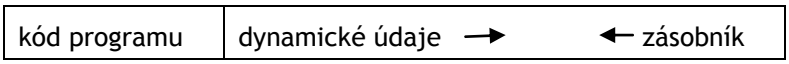
bezprostredne po ukončení vykonávania jednej inštrukcie pracovať s nasledujúcou (jej adresu už máme v PC registri).

Pokiaľ chceme zmeniť postupnosť vykonávania inštrukcií, zrealizovať **skok** v programe, stačí v priebehu spracovania inštrukcie zmeniť obsah registra PC. Kód nasledujúcej inštrukcie sa bude čítať z adresy podľa nového PC.

**Podmienené skoky** sú zmeny riadenia, ktoré sa uskutočnia len vtedy, ak sú v okamihu vykonávania inštrukcie splnené určené podmienky. Najčastejšie sa vyhodnocuje stav procesora (v stavovom slove), prípadne **register príznakov** (*flag register*). Príznaky sú obyčajne jednobitové údaje v príznakovom registri, ktoré sa nastavujú pri určených udalostiach, resp. ako stav výsledku niektorých operácií. Typickými príznakmi bývajú prenos (carry) mimo rozsah aritmetického registra, nulový výsledok (zero), chyba prekročenia celočíselného rozsahu (overflow) a iné. Ak požadované podmienky pre zmenu riadenia nie sú splnené, pokračuje sa vykonávaním nasledujúcej inštrukcie.

Inštrukcie podmienených skokov môžu testovať aj kombinácie príznakov.

Pre ukladanie medzivýsledkov, ktorých počet prekročí možnosti univerzálnych registrov, slúži obyčajne aj **zásobník** (*stack*). Zásobníková pamäť je vnorená (podobne ako aj údaje a program) v operačnej pamäti. Na udržanie tejto údajovej štruktúry stačí jeden špeciálny **register SP** (*Stack pointer* – ukazovateľ na **vrchol zásobníka** – obsahuje v každej chvíli adresu v operačnej pamäti, kde sa nachádza vrchol zásobníka).




*Štandardné rozmiestnenie programu a údajov v pamäti*

Zásobník je obyčajne realizovaný od vyšších adries k nižším. Program je totiž uložený od nižších adries k vyšším a v klasickom pamäťovom modeli za programom a statickými údajmi nasledovali dynamické údajové štruktúry. Pokiaľ dostal program vyhradený priestor v operačnej pamäti, bolo výhodné využiť ako zásobník práve koniec tohto priestoru. Pamäť sa takto efektívne rozdelí medzi tieto dve štruktúry. Problémom býva známe "pretečenie zásobníka", kedy zásobník zasiahne do údajov programu (čo sa dá ustrážiť len neustálou kontrolou za cenu zníženia rýchlosti programu).

Zásobník je údajová štruktúra typu LIFO a ukladanie sa rieši v dvoch krokoch - najskôr treba znížiť hodnotu registra SP, aby ukazoval o toľko bajtov menej, koľko chceme vložiť do zásobníka a v druhom kroku treba na vytvorené miesto požadované údaje uložiť. Opačným spôsobom riešime výber zo zásobníka – najskôr prečítame, čo sa nachádza v pamäti na mieste, kde ukazuje register SP (teda na vrchole zásobníka), potom musím obsah tohto registra zvýšiť (staré hodnoty v pamäti ostávajú, pokiaľ zásobník znova narastie, budú prepísané).

Štruktúra zásobníkovej pamäte umožňuje jednoduché vnáranie procedúr (*nested procedure call*).

Okrem zmeny riadenia pomocou skokov väčšinou architektúra podporuje aj **volanie procedúr** (*Procedure Call*). Tie sa realizujú podobne ako skoky zmenou PC registra, ale ešte pred zmenou sa jeho hodnota uloží do zásobníka. Dôvod je ten, aby sa procesor vedel neskôr vrátiť na miesto, z ktorého toto volanie uskutočnil (na rozdiel od jednoduchého skoku, kde po zmene PC registra už nebude vedieť vrátiť jeho pôvodnú hodnotu späť). Pre návrat z procedúry je vyčlenená špeciálna inštrukcia, ktorá obnoví obsah PC registra výberom zo zásobníka. Programátor si len musí dať pozor, aby pri skončení procedúry vybral zo zásobníka všetko to, čo si doňho v priebehu procedúry vložil.

Pomocou zásobníka môžeme tiež odovzdávať parametre volania procedúry. Vo volanej procedúre ich budeme mať prístupné na adresách s konkrétnym posuvom (offsetom) od hodnoty v PS.

Popis architektúry obsahuje tiež spôsoby komunikácie procesora so vstupno-výstupnými zariadeniami. Tieto budeme rozoberať v ďalšom module, tak nebudeme zatiaľ zaťažovať novými pojmami. Potom popíšeme aj spôsoby prerušovania výpočtu (dôležitý mechanizmus, ktorý nám dovoľuje s výpočtovým systémom interaktívne pracovať).

## 2.3 Operačný kód, operandy, adresovacie režimy

Nemýľte si strojový kód s riadiacim slovom mikroinštrukcie. Aj keď je tu určitá podobnosť, strojový kód potrebuje na svoju realizáciu niekoľko strojových cyklov – mikroinštrukcií.

Elementárnym krokom výpočtu z pohľadu programátora na tejto úrovni je **strojová inštrukcia** (alebo jednoducho inštrukcia). Jazyk strojových inštrukcií je tým nástrojom, ktorým môže programátor riadiť výpočet. Pre každú inštrukciu existuje jej **strojový kód** (*machine code*) – jednoznačne priradené binárne číslo, ktoré je možné uložiť do pamäte. **Program** je potom postupnosť strojových kódov inštrukcií (binárnych čísel), uložených v pamäti za sebou. Vykonávanie programu znamená postupné (sekvenčné) realizovanie strojových inštrukcií podľa ich strojových kódov.

Strojová inštrukcia musí obsahovať všetky údaje na to, aby mohla byť zrealizovaná jedna jednoduchá operácia – je to najmä **typ operácie** a parametre (**argumenty**), s ktorými sa má operácia vykonať. Binárny kód inštrukcie (strojový kód) najčastejšie začína kódom operácie, ktorá sa má realizovať - **operačným kódom**. Zvyšok bitov zápisu inštrukcie tvorí informácia o mieste, kde sa nachádzajú parametre požadovanej operácie – **operandy**.

| Operačný kód | Operand | Operand |
|---|---|---|

*Strojový kód inštrukcie*

Niektoré operácie (unárne) mávajú len jeden operand (napr. negácia), niektoré nemajú žiadne operandy (napr. návrat z procedúry).

Počet operandov súvisí s typom operácie – obyčajne sa používajú binárne operácie, ktoré potrebujú dva vstupné operandy, doplnené o tretí operand, ktorý určuje, kam sa má uložiť výsledok. Aby sa zjednodušil zápis inštrukcie, často sa výsledok ukladá na miesto jedného zo vstupných operandov (výsledkom sa prepíše jeho pôvodná hodnota).

Operand určuje umiestnenie argumentu (vstupného resp. výstupného) inštrukcie. Na určenie miesta existuje v každej architektúre niekoľko spôsobov -**adresovacích režimov** (*addressing modes*). Patrí medzi ne aj

Príklady adresácie:

```
MOV   R1,R2
MOV   R1, [R2]
MOV   R1, 100
MOV   R1, [100]
MOV   R1, @[100]
MOV   R1, [R2+100]
MOV  R1, [R2+R3+100]
```

Niekedy sa pridávajú aj autoinkrementálne a autodekrementálne registrové nepriame režimy (po výbere argumentu sa automaticky zväčší resp. zmenší hodnota indexového registra).

- **registrový priamy** režim – operand identifikuje register, ktorého obsah je argumentom inštrukcie; bez prístupu do operačnej pamäte
- **registrový nepriamy** – operand identifikuje register, ktorého obsahom je adresa miesta v operačnej pamäti, kde sa nachádza argument
- **bezprostredný (konštantný)** režim – operand je argumentom inštrukcie; argument sa v priebehu spracovania nemôže meniť (nemôže byť výstupný), jeho veľkosť je ohraničená, nemusí sa ale pristupovať do operačnej pamäte
- **priamy** režim – operand obsahuje adresu miesta v operačnej pamäti, kde sa nachádza argument; musí sa do operačnej pamäte, adresa sa nemení
- **nepriamy** režim – operand obsahuje adresu miesta v pamäti, na ktorej je adresa miesta v pamäti, kde sa nachádza argument; do pamäte sa musí dvakrát!, adresu uloženia argumentu možno meniť – využiť ako ukazovateľ (pointer)
- **indexový** režim – operand špecifikuje posunutie v pamäti vzhľadom na obsah indexového registra, kde sa nachádza argument inštrukcie; posunutie je konštantné - používa sa na prístup k prvkom poľa, zmenou indexového registra sa dostanem k ostatným prvkom poľa
- **bázovo-indexový** režim – polohu argumentu v pamäti určujú bázový a indexový register s posunutím podľa operandu; možno meniť dynamicky umiestnenie poľa i index

## 2.4 Typy inštrukcií

**Jazyk strojových inštrukcií** (machine instruction language) definuje zoznam inštrukcií, ktoré môže programátor použiť.

Inštrukcie pre **operácie prenosu** sú najčastejšie používaným typom inštrukcií. Obsahujú dva operandy, ktoré určujú miesto, z ktorého sa majú údaje preniesť (*source*) a miesto, kam sa údaje majú uložiť (*destination*). Prenosy bývajú navzájom medzi registrami, medzi registrom a operačnou pamäťou a medzi dvoma

pamäťovými miestami. Údaje v mieste zdroja ostávajú, na cieľové miesto sa prenesie ich kópia.

Môžeme prenášať jednotlivé bajty, slová alebo celé sekvencie bajtov (*string*). Pri prenose bajtových sekvencií (v operačnej pamäti) musíme dbať na to, aby sa prekrývané sekvencie kopírovali v správnom poradí (od začiatku, resp. od konca).

**Aritmetické operácie** tvoria jadro výpočtovej časti programov. Štandardne sú implementované operácie sčítania, odčítania, násobenia, delenia (celočíselného), operácie výpočtu zvyšku (*mod*) po delení. Existujú aj špeciálne operácie pre pridanie (*increment*) jednotky a odobratie (*decrement*) jednotky.

Výpočet výsledku operácie prebieha podľa zvoleného rozsahu – bajt, slovo (2 bajty), rozšírené slovo (4 bajty), pričom sa neprihliada na znamienko (počíta sa stále s prirodzenými číslami s nulou modulo veľkosť rozsahu). Interpretácia výsledku ostáva na programátorovi (typovej kontrole prekladača). Pre celé čísla je teda možné použiť reprezentáciu v tvare dvojkového doplnkového kódu.

Okrem uloženia samotného výsledku (často sa zvykne ukladať na miesto jedného zo vstupných operandov) sa zaznamená aj príznak výsledku do príznakového (*flag*) registra (resp. stavového slova programu PSW). Príznaky sú jednobitové informácie napr. o tom, či výsledkom operácie bola nula (Zero), výsledok mal v najvyššom ráde jednotku (Negative), pri výsledku došlo k prenosu z najvyššieho rádu (Carry), či došlo k prekročeniu rozsahu v zmysle dvojkovej doplnkovej interpretácie (Overflow).

Špecifickými operáciami sú aritmetické operácie s číslami, zapísanými vo formáte s pohyblivou rádovou čiarkou (obyčajne podľa normy IEEE 754). Pre tieto operácie bývajú vyhradené špeciálne registre a samostatné aritmetické jednotky, kam sa pošlú vstupné údaje a očakáva sa výsledok.

V predchádzajúcom module sme spomínali aj **logické operácie**. Tie sa obyčajne vykonávajú po jednotlivých bitoch operandov. Typicky sú k dispozícii binárne operácie AND, OR, XOR, unárna operácia doplnku NOT. Logické operácie umožňujú prístup k jednotlivým bitom bajtu, čo nám štandardná adresácia neumožňuje. Môžeme napr. nastavovať bity (operáciou OR s maskou nastavovaných bitov), mazať (operáciou AND), prípadne negovať bity (operáciou XOR).

**Operácie posunu** sa často používajú pre urýchlenie výpočtov – môžeme pomocou nich rýchle násobiť a deliť mocninami dvojky. Procesory väčšinou rozlišujú medzi logickými a aritmetickými posunmi (vľavo aj vpravo). Pri logických posunoch dochádza k posunutiu bez ohľadu na znamienko, na uvoľnené miesto sa vkladá 0 alebo hodnota príznaku Carry. V prípade rotačných logických posunov prechádza na voľné miesto bit, ktorý sa uvoľni na opačnej strane registra. Aritmetické posuny zachovávajú znamienko operandu (pri posune vľavo sa ignoruje najvyšší bit a pri posune vpravo si uvoľnené najvyššie miesto uchová pôvodnú hodnotu.

K štandardným typom inštrukcií patria ešte inštrukcie pre vstupno-výstupné operácie, inštrukcie sw prerušenia a riadenia stavu procesora. K týmto sa vrátime v ďalšom module, kde sa budeme zaoberať komunikáciou procesora s okolím.

**Práca so zásobníkom** zahŕňa typicky vloženie a výber zo zásobníka. Používa sa často vtedy, keď potrebujeme univerzálne registre na medzivýsledky výpočtu. Netreba zabudnúť, že zo zásobníka vyberáme údaje v presne opačnom poradí, v ktorom sme údaje do zásobníka vložili.

**Operácie prenosu riadenia - skoky, podmienené skoky,** sme už spomínali. Realizujú sa zmenou PC registra, v prípade podmienených registrov na základe stavu príznakov (nastavených v poslednej operácii)

Spomenieme tiež reakcie procesora na hw prerušenia (maskovateľné, nemaskovateľné), synchrónne hw prerušenia – mimoriadne udalosti, výnimky (exceptions), pasce (traps), výpadky (faults), prekročenia rozsahu pridelenej pamäte, neznámy kód inštrukcie, delenie nulou atď.

**Volanie podprogramov** zabezpečujú inštrukcie, ktoré dovoľujú realizovať vnárané procedúry kombináciou skoku a uloženia adresy nasledujúcej inštrukcie na vrchol zásobníka. Na konci podprogramu sa vráti riadenie na miesto, uložené vo vrchole zásobníka.

## Čo sme sa naučili

Získali sme predstavu o rôznych úrovniach pohľadu na výpočtový systém, špeciálne viac o úrovni strojových inštrukcií a ISA architektúre. Ukázali sme charakteristické črty a funkcie, ktoré môžeme na tejto úrovni od výpočtového systému - procesora - očakávať.

# 3 Architektúra Intel x86, asembler

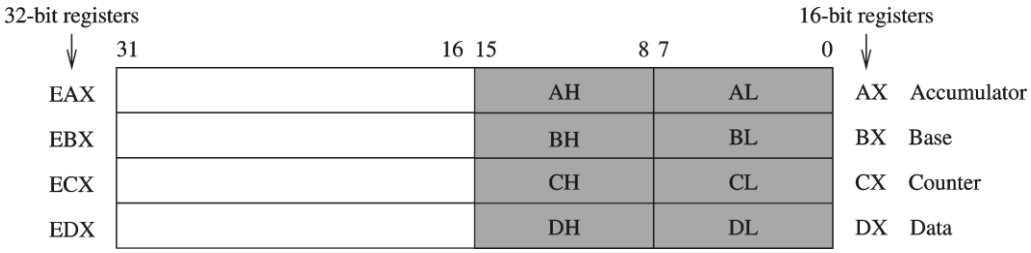
Konkretizujme v tejto časti architektúru procesorov Intel x86. Táto architektúra sa používa už od 80-tych rokov minulého storočia a obsahuje ju väčšina procesorov bežných osobných počítačov. Konkrétne modely procesorov sa na úrovni strojových inštrukcií z dôvodov kompatibility a prenositeľnosti programov príliš nelíšia. Občas bývajú pridávané špeciálne inštrukcie, ktoré je ale možné softvérovo ošetriť.

Ukážeme si aj možnosti využiť strojový kód vo vyššom programovacom jazyku (v prostredí Lazarus).

## 3.1 Štruktúra registrov procesorov typu Intel x86

Architektúru Intel x86-64 si netreba mýliť s celkom odlišnou 64-bitovou architektúrou Intel IA-64 Itanium.

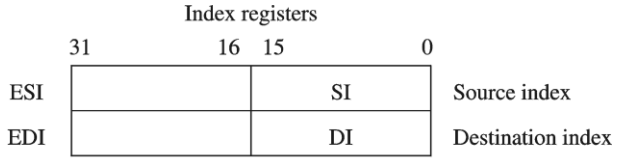
Pre jednoduchosť uvedieme 32-bitovú verziu x86. Existuje aj rozšírený 64-bitový variant x86-64 (niekedy označovaný len x64).

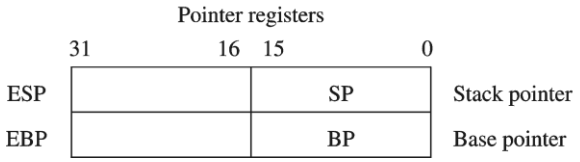



V x86-64 je možné použiť 64-bitové registre, ktoré sú ďalším rozšírením uvedených. Označujú sa RAX, RBX, RCX, RDX ...

Základnú skupinu registrov tvoria univerzálne registre EAX, EBX, ECX, EDX (označenia sledujú určité doporučenia pre ich využitie – register A – aritmetický register pre medzivýsledky výpočtov, B – bázový register pre prístup k prvkom poľa, C – počítadlo (counter), D – údaje (data)). Ku každému registru je (z dôvodov kompatibility s predchádzajúcimi modelmi s menším počtom bitov) aj 16-bitový a 8-bitový prístup. Registre AX, BX, CX a DX sú dolnými 16-bitovými časťami registrov EAX, EBX, ECX a EDX. K ich obsahu môžeme pristupovať po ôsmich bitoch – k horným registrami AH, BH, CH a DH a k dolným registrami AL, BL, CL, DH. Ak teda máme v EAX hexadecimálne hodnotu 0x12345678h, bude v AX hodnota 0x5678h, v AH 0x56h a v AL 0x78h.

Architektúry Intel používajú zápis do operačnej pamäte s prístupom Little endian. Uvedený obsah registra EAX sa do pamäte uloží v poradí bajtov 78 56 34 12.

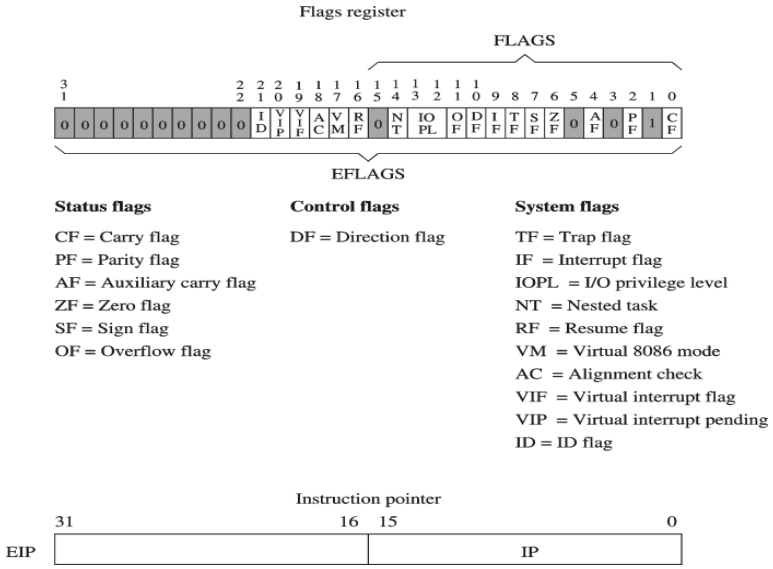


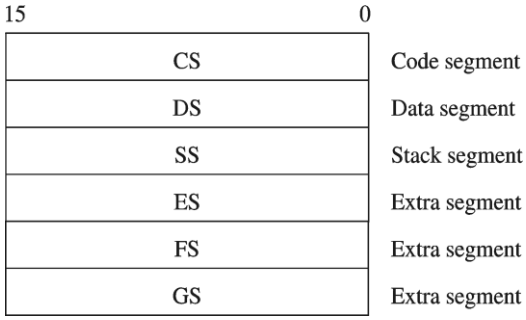

Registre ESI a EDI (resp. ich 16-bitové dolné časti SI a DI) slúžia ako zdrojový a cieľový index pre prenášanie reťazcov v operačnej pamäti. Dajú sa používať tiež ako univerzálne registre. Registre ESP a EBP využívame na prácu so zásobníkom. ESP (resp SP) udáva adresu vrcholu zásobníka v operačnej pamäti, EBP slúži na prístup k parametrom procedúry, ukladaným cez zásobník.

Dôležitým registrom pre riadenie výpočtu je príznakový register (Flags). Obsahuje jednobitové informácie (príznaky), ktoré sa nastavujú po ukončení väčšiny inštrukcií a signalizujú, ako inštrukcia dopadla. Najčastejšie sa používajú príznaky CF (prenos z najvyššieho rádu), ZF (výsledok operácie bol nula), SF (výsledok operácia bol záporný), OF (došlo ku prekročeniu rozsahu v celočíselnej reprezentácii). DF (slúži na riadenie smeru kopírovania blokov údajov -pre kopírovanie od začiatku (forward)

je DF=0 a pre kopírovanie od konca (backward) je DF=1). IF hovorí o zákaze prerušenia (ak IF=0).




Možno najdôležitejším je register IP, ktorý obsahuje adresu nasledujúcej inštrukcie programu, ktorá sa má vykonať.




Architektúra x86 využíva spoločný adresovací priestor operačnej pamäte pre inštrukcie aj pre programy. Rozlíšenie a vzájomná ochrana sa uskutočňuje pomocou **segmentácie**. Kód programu je v kódovom segmente, ktorého umiestnenie v operačnej pamäti je popísané na mieste, určenom v CS (code segment) registri. Kód strojovej inštrukcie, ktorú má procesor vykonávať, sa teda nachádza v operačnej pamäti na mieste, určenom CS s posunom IP od začiatku segmentu (zvykne sa označovať CS:IP).

V popisovači segmentu je uvedené, kde segment začína, jeho veľkosť a práva prístupu do segmentu.

Adresa uloženia údajov sa viaže na údajový segment, ktorého popis je možné vyhľadať obyčajne podľa registra DS (data segment) a príležitostne aj ES, FS alebo GS. Zásobníková pamäť je tiež v špeciálnom segmente operačnej pamäti, určenom SS (stack segment) registrom. Aktuálna pozícia SS:SP v zásobníku sa vypočíta pomocou SS a SP registra. Údaje sa do zásobníka ukladajú od vyšších adries k nižším.

Okrem týchto registrov procesory Intel x86 disponujú špeciálnymi registrami pre prácu s pohyblivou rádovou čiarkou a pre riadenie špeciálnych funkcií procesora.

## 3.2 Strojové inštrukcie architektúry Intel x86, asembler.

Procesor je schopný spracovať sekvencie jednoduchých príkazov – strojových inštrukcií – vo forme binárnych čísel, uložených do operačnej pamäte. Pokiaľ by musel programátor pri programovaní pracovať priamo s týmito číslami, mal by to veľmi ťažké.

Prekladač, vytvárajúci celý obraz úlohy zo symbolického jazyka sa zvykne väčšinou tiež volať **Assembler**. Jeho úlohou je popripájať k výsledku aj ďalšie funkčné knižnice, sprostredkovať komunikáciu s operačným systémom, prípadne s ďalšími procesmi, využívať systémové zdroje a vstupno-výstupné zariadenia.

Aby sme odlíšili tento spôsob (ktorý už vlastne je v hierarchii o dve úrovne abstrakcie vyššie), budeme slovo asembler vo význame jednoduchého zápisu strojovej inštrukcie symbolickým jazykom používať s malým začiatočným písmenom.

Informácie o rôznych prekladačoch strojového jazyka možno nájsť na http://en.wikipedia.org/wiki/List_of_assemblers

| | | | |
|---|---|---|---|
| Ulož 5 do registra AX | ax := 5; | MOV AX,5 | 66B80500 |
| Ulož 10 do registra CX | cx := 10; | MOV CX,10 | 66B90A00 |
| Pripočítaj obsah registra CX k registru AX | ax := ax + cx; | ADD AX,CX | 6601C8 |

*Príklad zápisu postupu slovne, v Pascale, v asembleri, v strojovom kóde*

Na zjednodušenie programovania na úrovni strojových inštrukcií slúži asembler – symbolický zápis strojových inštrukcií pomocou čitateľných textových reťazcov. Tento je možné vnárať do textu programu vo vyšších programovacích jazykoch, alebo použiť prekladač, ktorý je schopný previesť zápis inštrukcií v symbolickom programovacom jazyku (Assembly language) do vykonateľnej postupnosti kódov strojových inštrukcií.

My budeme používať symbolický zápis strojových inštrukcií, vložený do programu vo vývojovom prostredí Lazarus (*inline assembly*). Pri preklade programu sa jednoducho symbolický zápis prepíše do strojového kódu počítača v mieste, zadanom jeho polohou v zdrojovom texte programu. Pri tejto metóde musíme byť opatrní, aby sme zmenou obsahu registrov resp. pamäťových miest nespôsobili v programe chyby.

Inštrukcie strojového kódu v asembleri zapíšeme do zvláštneho bloku, ktorý začne rezervovaným slovom **`asm`** a končí slovom **`end`**. Pred prvým výskytom asemblerového bloku je potrebné zadať prekladaču typ procesora, ktorého symbolický zápis budeme používať. Direktíva **`{$ASMMODE intel}`** napríklad upozorní prekladač, že budeme používať asembler pre procesor architektúry Intel.

Každú strojovú inštrukciu v asembleri píšeme väčšinou do zvláštneho riadka (resp. viac inštrukcií musíme oddeliť bodkočiarkou). Zápis inštrukcie začína skratkou pre operačný kód, ktorý je nasledovaný operandami, oddelenými čiarkou. Komentár musí zohľadňovať pravidlá Pascalu a nesmie byť vnútri zápisu strojovej inštrukcie.

Režimy adresácie - len jeden operand sa môže nachádzať v operačnej pamäti !

```
MOV     AX, 100
MOV     AX, BX
MOV     AX, [BX]
```

Asembler procesorov Intel používa tzv. reverzný zápis – teda inštrukcia MOV AX, BX hovorí, že do registra AX treba skopírovať obsah registra BX. Pre prístup k argumentom inštrukcie môžeme použiť viacero adresovacích režimov. V bezprostrednom režime použijeme číselné konštanty (nie mená premenných konštantného typu) – implicitne v desiatkovej sústave, hexadecimálny zápis uvedieme znakom $. Registrový režim zvolíme jednoducho použitím mena príslušného registra, nepriamy registrový režim uvedením mena registra do hranatých zátvoriek. Zmenu segmentu v našich príkladoch nebudeme využívať.

Názvy registrov a ich častí sú vnútri bloku **`asm`** rezervovanými slovami. V zápise inštrukcií inline asemblera v prostredí Lazarus môžeme využívať aj označenie premenných vyššieho jazyka – pokiaľ by malo dôjsť ku kolízii s rezervovanými slovami assemblera, možno použiť prefix & pred označením premennej. Používanie premenných vyššieho jazyka prekladá prostredie Lazarus do príslušných zložitejších adresovacích režimov.

Podrobnejšie si môžete prečítať o vnorovaní strojového kódu do Object Pascalu v http://docs.embarcadero.com/products/rad_studio/cbuilder6/EN/CB6_ObjPascalLangGuide_EN.pdf kapitola 13 Inline assembly code, Object Pascal Language Guide, Borland Software Comp. 2002.

Predstavíme teraz niektoré typy operácií inštrukčnej sady procesorov Intel x86 spolu s príkladmi zápisu v symbolickom jazyku.

## Operácie prenosu

Syntax operácie prenosu je MOV cieľ, zdroj. Inštrukcia hovorí procesoru, že má obsah zdrojového miesta skopírovať do cieľovej pozície. Jeden z operandov musí byť registrového typu, ktorý tiež určuje veľkosť údajov, ktorý sa má preniesť (bajt, slovo resp. dvojslovo).

```
MOV    dst, src
```

Vyskúšajte v prostredí Lazarus nasledujúcu procedúru, v ktorej si všimnite prístup k jednotlivým častiam registra EAX a dodržiavanie kontroly typov argumentov.

```
var
  a, b: word;
  x, y: byte;
begin
    {$ASMMODE intel}
  asm
    MOV     EAX,$12345678
    MOV     a,AX
    MOV     x,AL
    MOV     y,AH
  end;
  Memo1.Append(inttohex(a, 8));
  Memo1.Append(inttohex(integer(x), 8));
  Memo1.Append(inttohex(integer(y), 8));
end;
```

## Aritmetické operácie

Pre štandardné aritmetické operácie sčítania a odčítania používame kódy ADD a SUB. Sú to binárne operácie, preto potrebujeme dva vstupné operandy. Architektúra x86 používa jeden z nich (v poradí prvý) tiež aj ako cieľový operand. Ten samozrejme nemôže byť adresovaný v bezprostrednom režime. Často sa používajú aj inštrukcie INC a DEC, ktoré hodnotu jedného operandu zväčšia resp. zmenšia o 1.

```
ADD    dst, src
SUB    dst, src
```

```
INC    dst
DEC    dst
```

**Zadanie 8**

Nastavte v prostredí Lazarus zobrazovanie okna Assembler a Registers (v menu Zobraziť-Okná ladenia). Urobte ladiacu značku na začiatku kódu v asembleri v predchádzajúcej ukážke a spustite beh programu. Keď sa beh pozastaví na značke, uvidíte v sledovaných oknách obsah registrov v okamihu prerušenia činnosti programu a výpis strojového kódu v mieste prerušenia v symbolickom jazyku.

Pozrite kompilovaný program a porovnávajte so zdrojovým kódom v Pascale. Vo vykonávaní pokračujte jednoduchými krokmi a všímajte si zmeny stavu v registroch.

Inštrukcia NEG vyrobí dvojkový doplnok k operandu (číslo s opačným znamienkom). Na celočíselné násobenie, delenie a zvyšok sa môžu použiť inštrukcie MUL a DIV.

Po ukončení každej aritmetickej operácie sa uložia do príznakového registra aj príznaky výsledku. Inštrukcia CMP nastaví príznaky tak, ako SUB, ale výsledok sa do cieľového operandu neuloží.

## Logické operácie

Logické operácie vykonáva procesor po jednotlivých bitoch operandov. Opäť je ich niekoľko druhov – AND, OR, XOR a aj NOT (unárny operand negácie). Podľa výsledku sa aj tu nastavia príznaky v príznakovom registri. Ak chceme testovať jednotlivý bit

v operande, môžeme použiť inštrukciu TEST , ktorá pracuje ako AND, ale neukladá výsledok.

```
SHR   dst, src
SHL   dst, src
SAR   dst, src
SAL   dst, src
ROR   dst, src
ROL   dst, src
RCR   dst, src
RCL   dst, src
```

## Operácie posunu a rotácie

Posuny v registroch používame pre urýchlenie niektorých činnosti (násobenia, delenia mocninami dvojky) ale aj na prácu s údajmi na úrovni bitov. Operácia prebehne v mieste cieľového operandu. Druhý operand určí, koľkokrát sa má posun uskutočniť (mala by to byť konštanta alebo obsah registra CL).

```
JMP   dst
JZ    dst
JC    dst
JNZ   dst
```

## Operácie prenosu riadenia – skoky, podmienené skoky

Sekvenčné spracovanie inštrukcií je možné zmeniť inštrukciami nepodmieneného skoku - JMP alebo podmienených skokov - Jx - kde x je druh podmienky, teda zoznam príznakov, ktoré musia byť nastavené na 1, aby sa skok uskutočnil (napr. JZ, JC, JO, ale aj JNZ, JNC - skok ak nie je príznak nastavený na 1). V prípade nesplnenia podmienky sa pokračuje vykonávaním nasledujúcej inštrukcie. O zmene riadenia tak rozhoduje príznak výsledku naposledy vykonávanej aritmetickej resp. logickej inštrukcie.

Cieľovým operandom inštrukcií skoku je adresa inštrukcie, ktorou sa má vykonávanie programu pokračovať. V architektúre x86 rozoznávame krátke skoky (short), kde možno skočiť len v rozsahu -128 až 127 bajtov od aktuálnej adresy (skok sa vykoná pripočítaním tejto hodnoty k aktuálnemu obsahu IP registra), blízky (near), argumentom sa prepíše celý aktuálny obsah IP registra a vzdialený skok (far jump) - pri ktorom sa mení aj obsah segmentového regtistra.

```
procedure TForm1.Button1Click(Sender:TObject);
var
  i, a, b, c: integer;
begin
  for i := 1 to 10000000 do
  begin a := 30000; b := 30000; c := 0;
    repeat
    if odd(b) then c := c + a;
    b := b div 2; a := a * 2;
    until b = 0;
  end;
Memo1.Append(inttostr(c));
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  i, a, b, c: integer;
begin
  for i := 1 to 10000000 do
  begin a := 30000; b := 30000; c := 0;
    {$ASMMODE intel}
      asm
        MOV EAX, a
        MOV EBX, b
        MOV ECX, 0
@2:     SHR EBX, 1
        JNC @1
        ADD ECX, EAX
@1:     SHL EAX, 1
        CMP EBX, 0
        JNZ @2
      end;
  Memo2.Append(inttostr(c));
end;
```

*Program na súčin dvoch čísel pomocou sčítania v Pascale a v Asembleri*

Aby sme nemuseli odpočítavať pozície jednotlivých kódov vo výslednom programe, používame v symbolickom zápise návestia. **Návestie** je identifikátor, umiestnený v symbolickom zápise inštrukcií, ktorému je pri prepise do strojového kódu pridelená hodnota pozície v programe podľa jeho výskytu v symbolickom zápise. Na krátke vzdialenosti v rozsahu jedného **`asm`** bloku môžeme používať návestia, začínajúce znakom @ bez deklarácie.

Špeciálna inštrukcia LOOP odčíta jednotku v registri CX a ak sa ešte nedosiahla 0, vykoná sa zmena riadenia – skok na adresu, zadanú operandom. Inštrukcia LOOP pomôže efektívne realizovať cykly.

## Práca so zásobníkom

Pre prácu so zásobníkom používa asembler v architektúre x86 inštrukcie PUSH (vlož do zásobníka) a POP (vyber zo zásobníka). Využijeme ho pre dočasné uskladnenie obsahov registrov, keď ich potrebujeme na chvíľu na iné účely.

```
PUSH    src
POP     dst,
```

## Volanie podprogramov

Špeciálnym spôsobom zmeny v programe je inštrukcia volania procedúry – CALL. Špeciálna tým, že okrem skoku na miesto, zadané operandom, sa do zásobníka uloží adresa nasledujúcej inštrukcie (za inštrukciou CALL). Pokiaľ chcem potom vrátiť výsledok z procedúry a pokračovať inštrukciou, kde som prestal, stačí vrátiť zásobník do počiatočného stavu – vtedy bude na vrchole zásobníka tá adresa inštrukcie, kde by sme chceli, aby riadenie prešlo. Pri návrate z procedúry (inštrukciou RET bez operandov) zmení procesor obsah IP registra podľa vrchola zásobníka, čím sa vráti výpočet na miesto, kde sme ho predtým opustili.

```
CALL    dst
RET
NOP
```

Inštrukcia NOP – no operation neurobí nič. Používa sa na výplň v pamäti na odstránenie prípadných hazardov.

## Vstupno-výstupné operácie

Budú podrobnejšie spomínané v nasledujúcom module.

```
IN      dst,port
OUT     port,src
```

Začlenením strojového kódu do vyššieho programovacieho jazyka môžem dosiahnuť rýchlejší výpočet a efektívnejšie využívania zdrojov. Špeciálne môžem optimalizovať prístup do pamäte uložením často využívaných parametrov do registrov, optimalizovať aritmetické a kryptografické výpočty.

**Zadanie 9**

Prepíšte vyššie uvedený program na násobenie pomocou sčítania do prostredia Lazarus a odlaďte ho. Pozorujte jeho výpočet a porovnajte dosiahnuté časy.

Skúste jednoduchým testom porovnať rýchlosť vykonania inštrukcií INC AX a ADD AX,1 a podobne inštrukcií XOR AX,AX a MOV AX,0.

## Čo sme sa naučili

Poznali sme základné prvky architektúry Intel x86 a vyskúšali ovládanie počítača na úrovni strojových inštrukcií. Poznali sme spôsob, ako začleňovať program v asembleri do vyššieho programovacieho jazyka.

# 4 Inštrukčný cyklus

V tejto časti sa vrátime na úroveň riadenia výpočtu programovateľným radičom pomocou riadiacich slov. Ďalšie konkrétne rozširovanie nášho zapojenia do stavu, kedy by sa radič stal plnohodnotným procesorom by už bolo komplikované. Ukážeme si len niektoré postupy, ktoré sa pri tomto návrhu môžu použiť.

Návrh riadiacich sekvencií (niekedy sa označuje ako mikroprogramovanie) je súčasťou návrhu procesora (mikroarchitektúry), robí sa pred vlastnou výrobou a ostáva pre používateľa procesora neprístupná a skrytá.

Skúsme postupovať teraz opačne. Vieme už, aké typy inštrukcií chceme zrealizovať, tak sa budeme snažiť realizáciu popísať pomocou sekvencie riadiacich slov - krokov (mikroprogramu).

## 4.1 Časové členenie spracovania inštrukcie

Úlohou procesora je postupne vykonávať inštrukcie programu, ktoré sú uložené v pamäti vo forme čísel. Inštrukciu musí najskôr z pamäte načítať, dekódovať, pokiaľ k jej vykonaniu sú potrebné ďalšie údaje, je treba ich pripraviť tiež, nasleduje vykonanie operácie a uloženie výsledku.

Celá táto činnosť sa nemôže vykonať v jedinom strojovom cykle (riadenie by muselo byť veľmi komplikované a riadiaci register veľmi veľký). Postupnosť strojových cyklov, ktoré zabezpečia vykonanie strojovej inštrukcie sa nazýva **inštrukčný cyklus**.

Inštrukčný cyklus je riadený pre každú inštrukciu špecifickou postupnosťou riadiacich slov, niekedy sa označuje aj **mikroprogram**. Mikroprogramy sa tvoria v štádiu navrhovania procesora a sú uložené v jeho neprepisovateľnej pamäti.

Videli sme, že niektoré inštrukcie môžu byť zložitejšie (násobenie, volanie podprogramu), niektoré zase jednoduchšie (NOP, nastavenie príznakov ...). Príslušné mikroprogramy pre ich vykonanie budú teda nerovnakej dĺžky. Celý inštrukčný cyklus delíme na niekoľko **štádií** (stages), cez ktoré prebieha každé spracovanie inštrukcie. Typické členenie musí obsahovať prečítanie inštrukcie z pamäte, jej dekódovanie a dekódovanie argumentov operácie, vykonanie požadovanej operácie a uloženie výsledku. V jednotlivých krokoch môže byť činnosť napríklad nasledujúca:

| IF | ID | OF | IE | WB |
|---|---|---|---|---|
| Čítanie inštrukcie | Dekódovanie inštrukcie | Príprava operandov | Vykonanie inštrukcie | Zápis výsledku |

*Typické štádiá spracovania inštrukcie*

### Čítanie inštrukcie.

Prvým štádiom spracovania je prečítanie inštrukcie (IF - *Instruction Fetch*). Adresa miesta v pamäti, kde sa nachádza kód spracovávanej inštrukcie, ktorú chceme spracovať, je uložená v programovom počítadle PC (*program counter*). Procesor teda v tomto štádiu prečíta obsah operačnej pamäte z adresy, uloženej v PC a prečítaný operačný kód inštrukcie uloží do dekódovacej jednotky. Upraví tiež hodnotu PC (zväčší počítadlo o 1, aby ukazovalo už na nasledujúcu inštrukciu, prípadne na hodnoty operandov inštrukcie).

### Dekódovanie inštrukcie.

V štádiu dekódovania inštrukcie (ID - *Instruction Decode*) je dekódovaný operačný kód a podľa neho je vybratá príslušná postupnosť mikroinštrukcií (riadiacich slov), ktoré budú riadiť jej realizáciu vrátane postupu zisťovania jej argumentov (adresovacích režimov) a postupu na uloženie výsledkov.

Prvé dve štádiá budú rovnaké pre každý inštrukčný cyklus. Ďalší priebeh už závisí od typu inštrukcie.

### Príprava operandov.

Ďalším štádiom je príprava operandov (OF - *Operand Fetch*). V nej je potrebné vyhľadať v pamäti podľa použitých adresovacích režimov všetky argumenty operácie. Adresovacie režimy sú známe z operačného kódu a operandy sa nachádzajú za operačným kódom. Pre ich prečítanie použijeme opäť počítadlo PC (a jeho hodnotu priebežne po použití zvyšujeme). Na konci bude počítadlo obsahovať adresu operačného kódu nasledujúcej inštrukcie.

Trvanie tohto štádia (počet strojových cyklov na jeho realizáciu) závisí od komplexnosti použitých adresovacích režimov. V prípade, že argumenty už máme v registroch (bol použitý registrový adresovací režim), môže byť toto štádium dokonca vynechané.

### Vykonanie inštrukcie

(IE - *Instruction Execution*) – vykoná sa vlastná operácia, zakódovaná v inštrukcii. Trvanie štádia závisí od zložitosti operácie. V niektorých prípadoch môže byť vynechané (napr. pre inštrukciu NOP), niekedy však môže trvať aj niekoľko desiatok strojových cyklov (napr. pri násobení).

### Zápis výsledku

(WB - *Write Back*) – štádium zápisu výsledkov operácie závisí od adresovacieho režimu uloženia výsledku. Opäť sú operácie, ktoré výsledok uložiť nepotrebujú (resp. výsledok v štádiu vykonania bol uložený v správnom výstupnom univerzálnom registri). V tomto štádiu sa tiež upravia príznaky v príznakovom registri a zisťuje sa, či nejaké vstupno-výstupné zariadenie nežiada o prerušenie (bližšie o prerušeniach v nasledujúcom module).

Rozdelenie inštrukčného cyklu na štádiá a ich počet je špecifický pre konkrétny model procesora. Po ukončení jedného inštrukčného cyklu začína ďalší štádiom čítania inštrukcie. Priebeh inštrukčného cyklu procesora môžeme ilustrovať na príklade jednoduchej inštrukcie ADD BX,(AX):

- IF IR:=[PC]; PC:=PC+1; {do inštrukčného registra uloží obsah operačnej pamäte, kde by mal byť kód inštrukcie, ktorá sa má vykonať, súčasne zvýši hodnotu ukazovateľa inštrukcii tak, aby ukazoval (obsahoval adresu) na prípadné operandy inštrukcie resp. na nasledujúcu inštrukciu}
- ID CTR:=D[IR]; {dekódovaním inštrukcie získa adresu v mikropamäti, kde je uložená postupnosť riadiacich slov pre jej realizáciu}
- OF AR:=[AX]; {pre dekódovanú inštrukciu musí vybrať argument zadaný obsahom registra z operačnej pamäte, uloží ho do pomocného registra AR}
- IE BX:=BX+AR; {vlastné vykonanie inštrukcie, výsledok uloží do BX}
- WB {pretože výsledok už má v správnom registri, toto štádium odpadá}

Pomocné registre – neprístupné používateľom

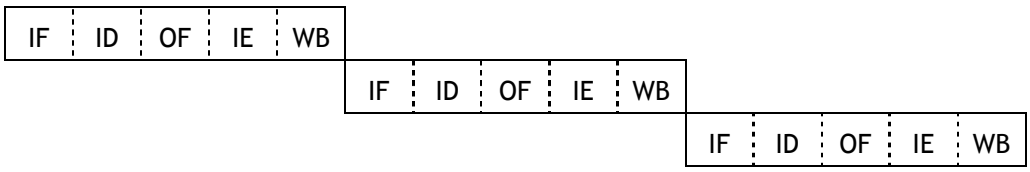
**Zadanie 10**

Popíšte podľa uvedenej schémy postup spracovania inštrukcie MOV AL,[1000]. Aspoň koľko strojových cyklov procesora (podľa nášho návrhu) by sme potrebovali na jej realizáciu ?

## 4.2 Zreťazenie spracovania inštrukcií

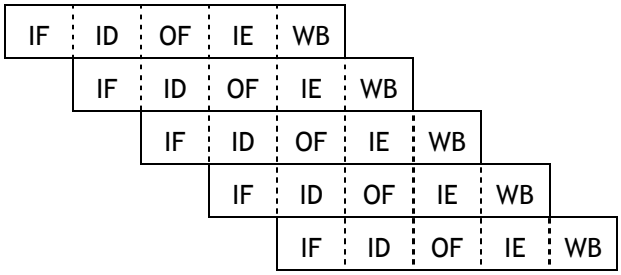
Štádiá spracovania inštrukcie sú dosť špecifické. Výhodnejšie by bolo, keby sme namiesto jedného univerzálneho programovateľného radiča navrhli viac radičov (výpočtových jednotiek), každý určený na konkrétne štádium spracovania. Inštrukcia by sa postupne "presúvala" od jednej jednotky k druhej ako na bežiacom páse. Hovoríme o zreťazení spracovania inštrukcie (*instruction pipelining*).

Pôvodný von Neumannov koncept hovorí, že inštrukcia sa môže začať spracovávať len vtedy, keď je ukončená predchádzajúca. Von Neumann si uvedomoval, že paralelné spracovanie inštrukcií môže viesť ku komplikáciám, pokiaľ inštrukcie budú pristupovať k rovnakým pamäťovým miestam.




*Sekvenčné spracovanie*

zreťazenie
prekrýva sa vykonávanie viacerých inštrukcií



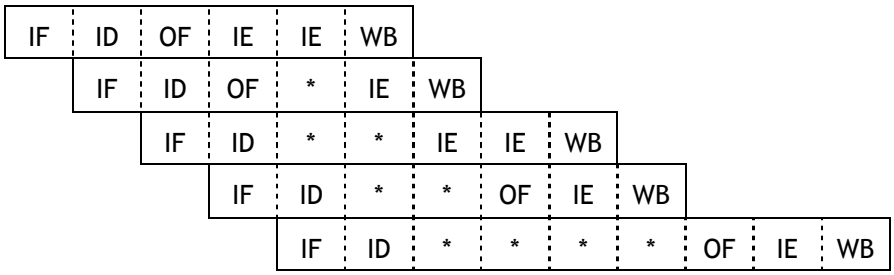

*Zreťazené spracovanie*

Pre efektívnosť zreťazenia je dôležitý aj počet inštrukcií, ktorý je možné súčasne otvoriť a pracovať na nich. Hovorí sa o počte **vydaní** (*issues*).

Pri ideálnom zreťazení, za predpokladu, že každé štádium spracovania trvá približne rovnaký čas, vykonanie napr. 5 inštrukcií po 5 štádiách bude zaberať namiesto 25 časových úsekov len 9. Reálne však má zreťazenie veľa limitujúcich faktorov, ktoré vnikajú organizačnými obmedzeniami (*structural hazards*), závislosťami medzi spracovanými údajmi (*data hazards*) a problémami, spojenými so zmenou riadenia toku inštrukcií (*control hazards resp. branch hazards*).

Jedným z organizačných obmedzení je nerovnaký čas spracovania inštrukcie v jednotlivých štádiách. Niektoré štádiá pri určitých typoch inštrukcií odpadajú, niektoré však trvajú podstatne dlhšie (napr. vykonanie operácie násobenia). Nastavenie časovania podľa najdlhšie trvajúceho štádia by ale bolo neefektívne. Jednoduchšie je ostať v problematickom štádiu viac časových úsekov, čo ale znemožní vstup do toho štádia ostatným. Tým sa celé spracovanie spomalí.

Obmedzením je tiež nemožnosť súčasného prístupu do pamäte pre dve rôzne jednotky. Aj keď oddelením inštrukčnej pamäte je možné čítanie inštrukcií realizovať nezávisle, v konflikte ostáva prístup do údajovej pamäte (štádiá OF a WB sa nemôžu vykonávať súčasne).




*Zreťazenie po zohľadnení organizačných hazardov*

Na obrázku vidíme, že už pri tomto zohľadnení môžu nastať situácie, že niektoré jednotky nepracujú. Nemožno teda reálne počítať s tým, že zvyšovaním počtu výpočtových jednotiek (štádií výpočtu) lineárne vzrastie výkonnosť procesora. Nevyužité jednotky (napr. v uvedenim príklade jednotka na vykonanie inštrukcie IE v 10. a 11. časovom úseku) by bolo možné vyťažiť zväčšením počtu otvorených inštrukcií – **vydaní** (issues). Neúmerné zvyšovanie vydaní ale zase komplikuje organizáciu práce. Niektoré používané hodnoty môžete nájsť v tabuľke.

| | Pipe Stages | Issue Width |
|---|---|---|
| Intel 486 | 5 | 1 |
| Intel Pentium | 5 | 2 |
| Intel Pentium Pro | 10 | 3 |
| Intel Pentium 4 Willamette | 22 | 3 |
| Intel Pentium 4 Prescott | 31 | 3 |
| Intel Core | 14 | 4 |
| Sun USPARC III | 14 | 4 |
| Sun T1 (Niagara) | 6 | 1 |

*Počty štádií zreťazenia a vydaní používané pre niektoré typy procesorov*

**Údajové hazardy** vznikajú používaním rovnakého registra v dvoch inštrukciách za sebou. Napríklad v postupnosti ADD CX,AX; MOV (CX),BX nie je možné otvoriť štádium prípravy operandov druhej inštrukcie, pokiaľ sa neukončí štádium zapisovania výsledkov prvej inštrukcie.

Zmeny v riadení sú najväčším problémom zreťazenia. **Hazardy riadenia** vznikajú už pri realizácii jednoduchého skoku. To, že dôjde ku skoku, sa totiž dozvieme až v štádiu dekódovania inštrukcie. V tom čase už prebehlo načítanie ďalšej inštrukcie v poradí, ktoré musíme anulovať a načítavať ďalšiu z nového miesta. Horšia je situácia pri vykonávaní inštrukcií podmienených skokov, kde nevieme určiť vopred pokračovanie programu, ale musíme čakať na výsledok posledných operácií pred skokom, kedy sa nastavia aj príslušné príznaky.

Na zreťazenie v prípade nášho riešenia v úvode modulu by sme potrebovali účinnejšie oddelenie jednotlivých častí – napr. pridaním oddeľovacích registrov, kde by sa ukladal priebežný stav na zbernici z predchádzajúceho taktu. Nevyužitý druhý takt by sme napr. mohli použiť na uloženie vybraných hodnôt bloku registrov, čím by sa uvoľnila možnosť zapisovať výsledky hneď v nasledujúcom takte.

V súvislosti s využívaním rýchlych vyrovnávacích pamätí (budú spomenuté v časti 4.5) vstupuje do hry ešte nerovnaký čas prístupu do pamäte, ktorý sa môže dynamicky meniť a mal by sa v organizácii zreťazenia zohľadniť.

## 4.3 Predikcia, vykonávanie inštrukcií mimo poradia

Pasívnym riešením hazardov efektivita zreťazenia veľmi klesá. Uvedieme niekoľko aktívnych postupov, ktoré môžu tento pokles zmenšiť.

Na riešenie hazardov, súvisiacich s podmienenými skokmi, sa používa **predikcia**. V zreťazenom (špekulatívnom) vykonávaní inštrukcií sa pokračuje v zvolenej (predikovanej) vetve, no zápis výsledkov sa odloží až do chvíle, keď je známe, či sa skok má uskutočniť alebo nie. Ak sa skok uskutoční podľa predikcie – máme pripravený ďalší postup. Ak sa predpoveď nepodarila, musíme zabudnúť na všetky prípravy a zreťazenie začať od nového miesta.

Inštrukcie NOP sa používajú za niektorými podmienenými skokmi, aby sa nemuselo v prípade odskoku cúvať z rozpracovaných inštrukcií.

Najjednoduchšie je predpovedať postup podľa typu skoku. Obyčajne sa skoky vzad (do určitej vzdialenosti) predikujú pozitívne. Väčšinou ide o menšie cykly, ktoré sa niekoľkokrát opakujú a pokračujú ďalej až po splnení nejakej podmienky. Niektoré procesory používajú aj špeciálny *loop buffer,* kde nechávajú údaje o niekoľkých posledne spracovaných inštrukciách, ktoré môžu v prípade spätných skokov využiť.

Podmienené skoky vpred sa obyčajne predikujú negatívne, teda volí sa pokračovanie za inštrukciou podmieneného skoku. Špekulatívne vykonávanie je dobré pozastaviť, keby malo spôsobovať zmeny v aktuálnom obsahu cache pamäte. Otázkou je aj reakcia na výnimky počas špekulatívneho vykonávania (napr. delenie nulou).

Vykonávanie inštrukcií mimo poradia porušuje presné pravidlá ošetrovania prerušení a výnimiek. Problémy sa môžu odstrániť tým, že sa inštrukcie spracujú mimo poradia, ale uzatvárať sa budú v pôvodnom poradí (*in-order completion*).

Sofistikovanejším riešením je sledovanie štatistík splňovania podmienok pre niekoľko (posledne vykonávaných) podmienených skokov v špeciálnych skokových registroch.

Údajové hazardy možno ovplyvňovať pravidlami výberu poradia spracovania inštrukcií (*instruction issue policy*). V niektorých prípadoch - napr. v postupnosti ADD CX,AX; MOV (CX),BX; MOV AX,DX – druhá inštrukcia musí pozastaviť prípravu operandov až do uloženia výsledku prvej inštrukcie (ako bolo spomenuté vyššie). Tretia inštrukcia ale nie je od ostatných dvoch závislá a môže využiť nečinné jednotky medzi spracovaním prvej a tretej inštrukcie. Namiesto pôvodnej postupnosti sa bude pracovať na postupnosti ADD CX,AX; MOV AX,DX; MOV (CX),BX – výsledok bude rovnaký. V tomto prípade hovoríme o vykonávaní inštrukcií **mimo poradia** (*out-of-order execution*).

uzavretie (commiting, retiring) inštrukcie

Ďalším často používaným postupom je premenovanie registrov (napr. v našom prípade inštrukcia MOV AX,DX môže zapísať do ľubovoľného nepoužívaného registra, ktorý sa premenuje na AX, aby nevznikal hazard s použitím AX v predchádzajúcej inštrukcii). V skutočnosti súčasné procesory majú registrov viac (v prípade viacvláknovej podpory minimálne pre každé vlákno svoje) a sú schopné prepínať kontexty vláken rýchlym premenovaním registrov.

Možnosťami preusporiadania inštrukcií (aj napr. predradenie spracovania inštrukcie, ktorá má vplyv na podmienený skok), premenovania registrov, predikcie skokov a optimalizácie prístupu do pamäte sa zaoberá v súčasných procesoroch špeciálna jednotka.

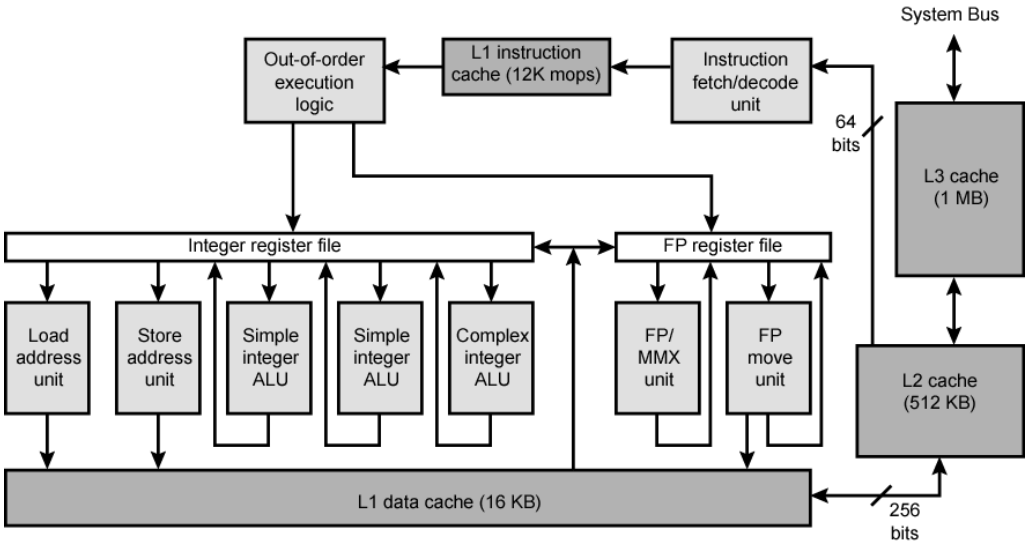
## 4.4 Viacvláknové a viacjadrové architektúry

Obyčajné procesory, vykonávajúce naraz len jednu operáciu, sa nazývajú **skalárne** procesory a procesory, vykonávajúce jednu operáciu ale súčasne s viacerými údajmi, nazývame **vektorové** procesory.

Vyššiu efektivitu zreťazenia dosiahneme paralelizáciou na úrovni jednotlivých štádií spracovania inštrukcie. Procesory, umožňujúce vykonávanie viacerých inštrukcií súčasne, nazývame tiež **superskalárne** procesory (*superscalar processors*). Najjednoduchšie sa inštrukčný paralelizmus dosahuje pridaním viacerých aritmeticko-logických jednotiek, špeciálnej jednotky FPU pre prácu s číslami s pohyblivou rádovou čiarkou, špeciálnych jednotiek pre prístup do údajovej pamäte.

Do všetkých pravidiel a techník, spomínaných v predchádzajúcej časti, prináša superskalárne spracovanie nové možnosti na urýchlenie výpočtu. Pri paralelnom spracovaní vzrastá efektivita otváraním väčšieho počtu vydaní inštrukcií a ich premysleným riadením. Viac nezávislých výpočtových reťazcov môže napríklad sledovať obidve vetvy podmieneného skoku a po výpočte podmienky urobiť uzavretie inštrukcií len v tej správnej.

Špecifický charakter má aj reťazec spracovania inštrukcie, pracujúcej s číslami s pohyblivou rádovou čiarkou. Najskôr je potrebné prekódovať číslo z normalizovaného tvaru do tvaru, v ktorom prebieha výpočet (spomenutý v prvom module) a po vlastnom výpočte je potrebné výsledok zase normalizovať. Tieto úkony väčšinou vykonáva samostatná jednotka pre prácu s reálnou aritmetikou - FPU (floating point unit), ktorej sa vo fáze vykonania inštrukcie presunú operandy.




*Zjednodušený pohľad na tok údajov procesora Intel Pentium 4 (podľa [7])*

MMX (MultiMedia eXtension) jednotka pre SIMD (Single Instruction Multiple Data) inštrukcie – ktoré sú schopné vykonávať tie isté operácie s viacerými údajmi (vektorové spracovanie)

Na uvedenej schéme údajových tokov procesora Intel Pentium si môžeme všimnúť dekódovaciu jednotku, ktorá vygeneruje zo strojových inštrukcií postupnosť mikrooperácií (na každú strojovú inštrukciu jeden až štyri), ktoré ďalej prechádzajú paralelne (superskalárnym zreťazením) výkonnými jednotkami, komunikujúcimi s údajovou pamätou. O poradí spracovania, ošetrení hazardov, mapovaní registrov (celkove na 128 celočíselných a 128 FPU) a v konečej fáze aj o uzatváraní (retiring) inštrukcií v poradí rozhoduje špeciálna jednotka na riadenie inštrukčného paralelizmu (*out-of-order execution logic*).

SSE (Streaming SIMD Extension) špeciálne inštrukcie architektúry x86 pre paralelné spracovanie údajov.

Dve aritmetické jednotky umožňujú týmto typom procesora podporovať aj simultánne dvojvláknové výpočty pomocou špeciálnej technológie s označením Hyper-Threading Technology. Pokiaľ je podporovaná aj operačným systémom, používateľom sa počítač javí ako dvojprocesorový.

HTT – paralelizmus na úrovni vláken (*thread level parallelism*)

Statický inštrukčný paralelizmus, ktorý môže organizovať programátor (na rozdiel od spomínaného dynamického, ktorý organizuje procesor) možno dosiahnuť VLIW inštrukciami (Very Long Instruction Word). VLIW inštrukcie kombinujú niekoľko operácií, pričom sa predpokladá, že procesor ich spracuje paralelne. Dajú sa nimi redukovať aj problémy podmienených skokov (môžu mať tvar ... ak platí podmienka, vykonaj operáciu ...).

SMP Shared memory processors - *process level parallelism*

Plnú paralelizáciu na úrovni procesov dosiahneme viacerými kópiami celého jadra procesora na jednom integrovanom obvode. Jadrá môžu zdieľať spoločnú vyrovnávaciu pamäť (alebo mať každý vlastnú) a spoločnú hlavnú pamäť, prostredníctvom ktorej môžu komunikovať.

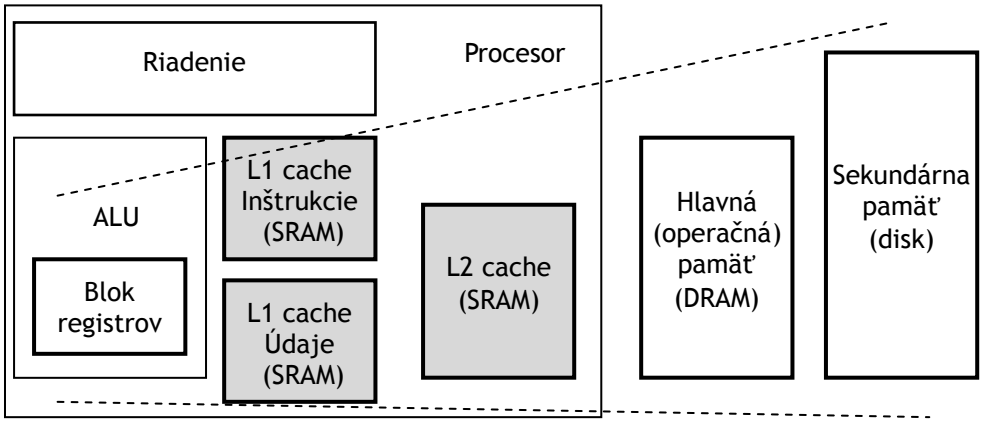
## 4.5 Využitie vyrovnávacích pamätí

Nároky na veľkosť spracovávaných údajov rastie spolu so zvyšovaním rýchlosti procesorov. Zväčšovanie kapacít dostupných pamäťových médií prebieha v súlade s týmito nárokmi. Problém však je v rýchlostiach prístupu k údajom. Pri taktovacej frekvencii procesora 2 GHz trvá jeden takt 0,5 ns. Prístupová doba k súčasným veľkokapacitným DRAM pamätiam (budeme sa nimi zaoberať v ďalšom module) je približne 50 ns. To znamená, že na jeden prístup do pamäte by sme museli čakať 100 taktov.

Aj hierarchizácia pamäte je v spore s von Neumannovym konceptom rovnakej (uniformnej) pamäte

Urýchlenie dosiahneme použitím drahšej ale rýchlejšej pamäte (SRAM), ktorú umiestnime medzi operačnú pamäť a pamäte priamo prístupné riadeniu procesora (inštrukčnú a údajovú). Využijeme fakt, že v susedných krokoch programu pristupujeme aj k susedným miestam v pamäti (v mieste uloženia údajov aj v mieste uloženia inštrukcií). Vyrovnávacia pamäť (*cache*) býva integrovaná s procesorom v jednom obvode (staršie typy počítačov ju majú umiestnenú samostatne).

SRAM (Static Random Access Memory) – vytvorená z RS preklápacích obvodov – navrhli sme na konci predchádzajúceho modulu a používali v návrhu procesora na začiatku tohto modulu.




64 B / 0,25 ns  32 kB / 0,5 ns  1 MB / 5 ns  4 GB / 50 ns  1 TB / 5 ms

porovnateľná cena – TB disk (70e), GB DRAM (20e), MB cache (20e)

*Typické veľkosti a prístupová doba k jednotlivým vrstvám hierarchickej pamäte*

Často používané miesta vyrovnávacej pamäte budeme mať prístupné rýchlo – úspešnému prístupu sa hovorí cache **hit**. Ak budeme potrebovať údaje z iného miesta (prístup bude neúspešný – cache **miss**), uskutoční sa prístup do pamäte vyššej úrovne a prenesie sa okrem hľadanej hodnoty aj niekoľko bajtov v okolí – riadok (line). Tento prístup už ale trvá podstatne dlhšie. Pri ukladaní riadku do menšej pamäte môžem mať problém, čo vyhodiť. Často sa použije pravidlo **LRU** (Least Recently Used) a uvoľní sa riadok, ktorý sa najdlhšie nepoužíval. Ak boli urobené v uvoľňovanom riadku zmeny, musia sa predtým ešte preniesť do pamäte vyššej úrovne.

Aj k sekundárnej (diskovej) pamäti sa pristupuje blokovo – po sektoroch dĺžky 512 B.

V novších typoch viacjadrových procesorov sa pridáva ďalšia vrstva L3 cache pamäťe. Býva potom vlastná L2 vyrovnávacia pamäť pre každé jadro (obyčajne 256 MB) a spoločná L3 vyrovnávacia pamäť (obyčajne 8 MB). Prístup sa uskutočňuje pomocou blokového prenosu po riadkoch dĺžky 64 B.

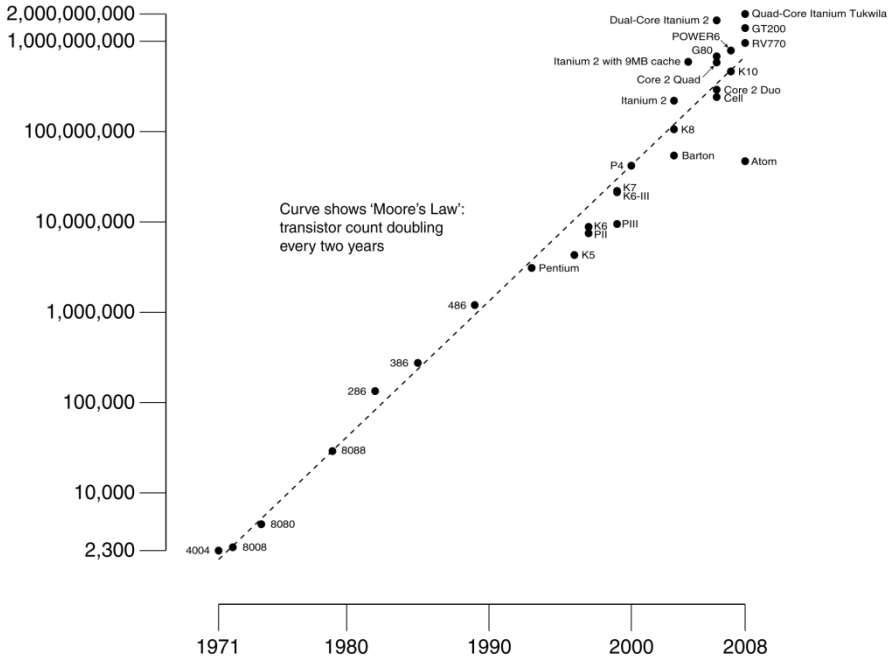
### Čo sme sa naučili

Oboznámili sme sa s časovým priebehom spracovania strojovej inštrukcie a s možnosťami jeho urýchlenia. Priblížili sme si aj problémy, ktoré s urýchľovaním prichádzajú.

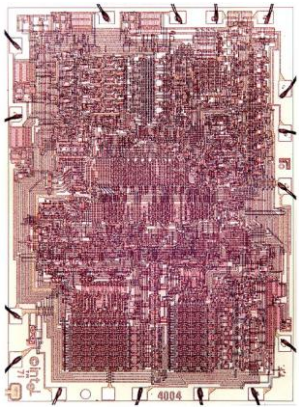
# 5 Perspektívy vývoja procesorov

Exponenciálny vývoj počítačových technológií je najlepšie badať na zvyšovaní hustoty integrácie tranzistorových prvkov v procesoroch. V nasledujúcom grafe možno sledovať tento vývoj už od čias výroby prvého **mikroprocesora** (centrálnej výpočtovej jednotky CPU (*Central Processing Unit*) počítača na jednom integrovanom obvode).

Tento jav komentoval už v 60-tych rokoch Gordon Moore (neskorší spoluzakladateľ firmy Intel Corporation) a na základe pozorovaní rozbehu technológií integrovaných obvodov formuloval predpoveď, že hustota integrácie sa bude každé dva roky zdvojnásobovať. Aj keď to bolo od začiatku dosť nepredstaviteľné, jeho predpoveď ostáva platná aj do dnešných dní, čo ilustruje nasledujúci graf.



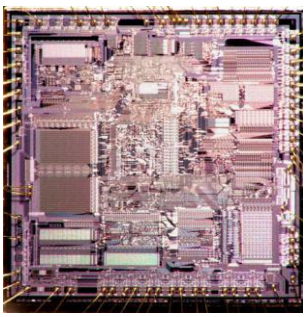

*Moorov "zákon" zvyšovania hustoty integrácie v mikroprocesoroch (http://en.wikipedia.org/wiki/Moore's_law)*



*Mikroprocesor Intel 4004 (1970) 2300 tranzistorov*

V súčasnosti používané výrobné technológie integrovaných obvodov (momentálne na hranici rozlíšenia 32 nm) vidia svoje hranice na rozlíšení 16 nm (používané kremíkové atómy majú rozmery okolo 0,25 nm). V poslednom čase sa zvyšovanie hustoty dosahuje hlavne paralelným zapojením viacerých jadier a integrovaním veľkokapacitných vyrovnávacích pamätí.

Na hraniciach je aj schopnosť odvádzať vytvorené teplo, ktoré vzniká koncentráciou výkonu na ploche cca jedného centimetra štvorcového. Na prekročenie týchto bariér bude potrebné o takých desať rokov prikročiť pravdepodobne k podstatným zmenám v technologických postupoch, založených možno na nanotechnológiách, možno na iných fyzikálnych či biofyzikálnych postupoch.



*Mikroprocesor Intel 80286 (1982) 134 tis. tranzistorov*

## 5.1 Vývoj architektúr procesorov Intel

Prvý mikroprocesor – teda procesorová jednotka vytvorená na ploche jedného integrovaného obvodu – bol 4-bitový procesor (pracoval so 4 bitovými registrami) s označením 4004, navrhnutý v roku 1970 v mladej firme Intel Corporation. Firmu (pôvodne pod názvom Integrated Electronics Corporation) založili v roku 1968 Robert Noyce (jeden zo spoluvynálezcov integrovaného obvodu) a Gordon Moore (známy autor Moorovho "zákona"). Čoskoro nato (v roku 1974) bol pripravený aj 8-bitový procesor Intel 8080 (s cca 5000 tranzistormi), ktorý sa stal na dlhšiu dobu svetovým štandardom.

Začiatkom 80-tych rokov (1979) nasledovali 16-bitové varianty 8086 (30 tis. tranzistorov) a 80286 (130 tis. tranzistorov). Dali základy architektúre x86 a v duchu myšlienky zachovania kompatibility, programy, napísané v strojovom kóde pre ne, je možné používať aj v súčasných typoch procesorov tejto architektúry.

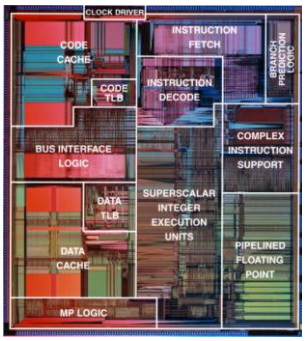
Tak to bolo aj v roku 1986 pri prechode na 32-bitový mikroprocesor 80386 (275 tis. tranzistorov), 32-bitový procesor s integrovanou jednotkou pre prácu s pohyblivou rádovou čiarkou 80486 (1,2 mil. tranzistorov) v roku 1989 a Pentium s 64-bitovými dátovými prenosmi v roku 1993 (3,1 mil. tranzistorov).

Označenie Pentium potom na niekoľko rokov ostalo, objavil sa procesor PentumPro (1995) s integrovanou cache pamäťou (6 mil. tranzistorov), Pentium II/III, Pentium 4 (2000) s možnosťou hyperthreadingu (42 mil. tranzistorov).

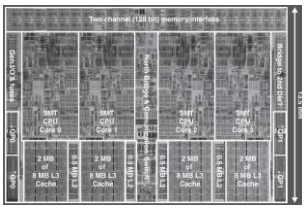
Viacjadrové riešenia priniesol (tiež aj v súvislosti s konkurenčným bojom s najväčším konkurentom – firmou AMD (Advanced Micro Devices)) procesor Intel Core (2003), Core 2 Duo (291 mil. tranzistorov), Core 2 Quad.

Vzniká podpora pre 64 bitové výpočty – architektúra x86-64, so zachovaním určitej kompatibility s x86, štvorjadrové procesory Core i7 (2009) (731 mil. tranzistorov) a osemjadrové 8-Core Xeon (2010) - 2,3 mld tranzistorov.

Snaha o úplne nové prístupy sa prejavila v návrhu 64-bitovej architektúry **IA-64** (v spolupráci s firmou HP). Táto architektúra ponúka celkom nový inštrukčný súbor s označením EPIC (Explicitly Parallel Instruction Computing) – čo sú vlastne inštrukcie s viacerými operáciami (VLIW), ktoré sú spracované paralelne (paralelizmus môže ovplyvňovať aj programátor), 128 64-bitových registrov, ďalších 32 registrov ako zásobník, 128 fp registrov a 128 aplikačných registrov. Procesory tejto architektúry sú označované ako Intel Itanium (2006 Dual Core 1,7 mld. tranzistorov, 2010 Quad Core 2 mld. tranzistorov).




*Intel Pentium (1993) 3 mil. tranzistorov*



*Intel Nehalem Lynnfield (2009) 700 mil. tranzistorov*

## 5.2 CISC a RISC architektúra

V súvislosti s komplexnosťou inštrukcii sa zvykne rozlišovať medzi **CISC** (Complex Instruction Set Computer) a **RISC** (Reduced Instruction Set Computer). Idea návrhu procesorov s obmedzenou množinou inštrukcií – RISC - vznikla koncom 70-tych rokov v univerzitnom prostredí (Berkeley) ako reakcia na úvahy o ďalšom zvyšovaní počtu inštrukcií a približovaní sa strojového jazyka vyšším programovacím jazykom (aby sa zjednodušila práca na kompilátoroch).

Skratka MIPS (oficiálne Microprocessor without Interlocked Pipeline Stages) vznikla ako pripomenutie toho, že frekvencia MIPS procesorov v MHz obyčajne znamená ich výkon v MIPS (Million Instructions Per Second).

Reálne bola použitá začiatkom 80-tych rokov v procesoroch MIPS (Stanford, MIPS Techology Inc.) a SPARC (Scalable Processor Architecture, Sun Microsystems). Redukovaná inštrukčná množina pozostáva z cca 50 typov inštrukcií (na rozdiel od 200-300 inštrukcií v počítačoch IBM, DEC VAX, Intel).

Zložité inštrukcie sa vykonávajú len výnimočne a ak ich potrebujem, môžem zapísať ich postup pomocou sekvencie jednoduchých inštrukcií.

**Jednoduché inštrukcie** umožňujú jednoduché vykonávanie – v RISC architektúrach sa nepredpokladá rozdeľovanie spracovania inštrukcie na mikroinštrukcie – každá inštrukcia musí byť taká jednoduchá, aby sa dala spracovať **v jednom strojovom cykle**. Jednoduché spracovanie je umožňované aj **jednotným formátom**, **rovnakou veľkosťou kódu inštrukcií** a veľmi **malým množstvom adresovacích režimov**. Prístup do pamäte sa v RISC architektúre tiež obmedzuje len na jednoduché operácie LOAD a STORE, výpočty sa robia výlučne v procesore.

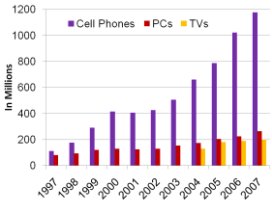
Jednoduché spracovanie dovolí lepšie optimalizovať aj inštrukčný paralelizmus – aj keď namiesto jednej CISC inštrukcie treba tri-štyri RISC inštrukcie, celková rýchlosť spracovania je menšia. Zreťazenie spracovania inštrukcií dovolí ukončovať strojový cyklus v každom takte procesora. Jednoduchosť inštrukcií zvyšuje nároky na efektívne preklady z vyšších jazykov. Menší počet typov inštrukcií sa kompenzuje **väčším počtom univerzálnych registrov**, kde je možné efektívne spravovať medzivýsledky (v RISC architektúrach slúžia skôr ako cache pamäť).

Registre sa často organizujú do tzv. kruhových buffrov – pre volaný podprogram ostane časť mojich registrov (môžeme nastaviť parametre volaných procedúr), ktoré mi volaný program vráti s novým obsahom.

Mnohé myšlienky RISC architektúr sa presadili aj do architektúr súčasných CISC procesorov (napr. do architektúry Intel IA-64).

Príkladmi RISC procesorov sú procesory MIPS R4000, Sun SPARC, IBM Cell, ale aj procesory AMD, ktoré sú na úrovni mikroinšrukcií postavené ako RISC, pričom interpretujú CISC inštrukcie a štruktúru procesorov architektúry Intel x86.

## 5.3 Zabudované procesory




*Výroba mobilných telefónov, PC a TV ročne podľa [5]*

Veľkými odberateľmi procesorov sú v posledných rokoch rôzne odvetvia spotrebnej elektroniky - od komunikačných zariadení, cez multimediálne prehrávače, herné konzoly až k inteligentnému ovládaniu priemyselných zariadení. Sú to vlastne programovateľné radiče (nie je možné meniť program), zabudované (*embedded*) v jednotlivých zariadeniach.

Podľa [5] sa napr. v roku 2007 vyrobilo 1,2 miliárd mobilných telefónov, asi 200 miliónov TV prístrojov so set-top-boxmi a len asi 250 miliónov PC. V roku 2009 bola používaná viac ako miliarda PC, ale súčasne viac ako 4 miliardy mobilných telefónov a viac ako 2 miliardy kábelových resp. digitálnych televíznych prijímačov.

Trh so zabudovanými (embedded) procesormi tvoria hlavne 8-bitové mikroradiče Intel 8051 a 32-bitové procesory ARM

### 8051

Mikroradič 8051 vznikol rozšírením zapojenia 8-bitového procesora Intel 8080 o 4 kB programovej pamäte ROM a ďalších 128 kB RAM pre údaje na jeden integrovaný obvod. Takýto obvod obsahuje okolo 60 tisíc tranzistorov a jednorázovým naprogramovaním inštrukčnej pamäte je pripravený na použitie. Jeho cena (vo väčších objemoch okolo 10 centov) ho predurčuje na široké využitie tam, kde netreba rýchly a zložitý výpočet.

### ARM

ARM (Advanced RISC Machine) je 32-bitová RISC architektúra procesorov, používaná v súčasnosti ako vnorený (*embedded*) procesor vo väčšine elektronických zariadení, pre ktoré 8-bitový radič 8051 nestačí. Nájdeme ho vo väčšine mobilných telefónoch, PDA (Personal Digital Assistant) zariadení, hudobných prehrávačoch, set-top-boxoch, herných konzolách, radičoch pevných diskov, smerovačoch počítačových sietí ...

ARM architektúra sa dá licencovať, preto mnoho značiek ako Atmel, Broadcom, Cirrus Logic, LG, Marvell Technology Group, NEC, NXP (Philips), Oki, Qualcomm, Samsung, Sharp, Symbios Logic, Texas Instruments, VLSI Technology, Yamaha ju využíva pod svojim menom.

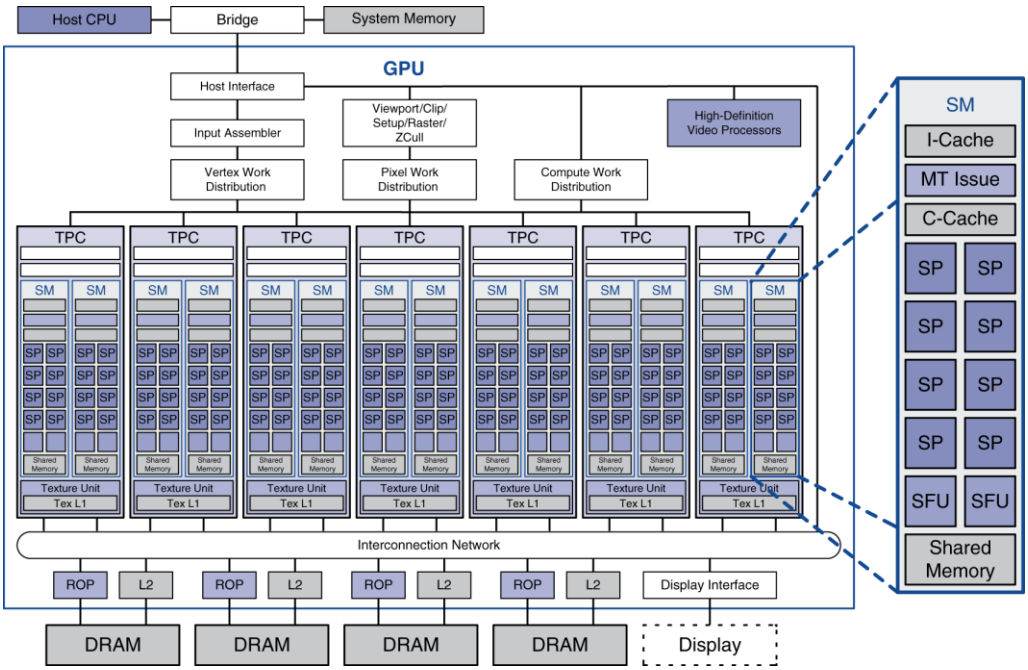
Architektúra používa 16 32-bitových univerzálnych registrov, inštrukcie pevnej 32-bitovej dĺžky so zreťazením spracovania, väčšina inštrukcií sa spracuje v jednom strojovom cykle. Inštrukcie využívajú load/store prístup bez zložitých adresovacích režimov, obsahujú podmienené jednorázové príkazy (redukuje sa tak nutnosť predikcie podmienených skokov), k inštrukcii je možné pripojiť príkaz pre posúvanie resp. rotáciu výsledku.

## 5.4 Grafické procesory

Už dlhodobo sa prudko rozvíjajú (a v mnohých detailoch i predbiehajú vývoj štandardných mikroprocesorov) zariadenia na spracovanie grafických údajov. Grafické procesory – GPU (Graphics Processing Unit) - sú používané v grafických adaptéroch ako akcelerátory grafických výpočtov. Pomáhajú CPU pri rasterizácii

obrazu, výpočte farieb bodov povrchu objektov pri ich osvetlení, rozdeľovaní zložitých povrchov objektov na trojuholníky, zisťovaní ich viditeľnosti, počítaní polohy objektu pri jeho otočeniach a posunutiach...

Ich výhodou je, že nemusia byť univerzálne a nemusia zachovávať spätnú kompatibilitu – vždy sa len pripravia vhodné ovládače, ktoré sú schopné zrealizovať na novej architektúre aj staršie funkcie. GPU spracúvajú štvorrozmerné vektory – súradnice v homogénnom tvare a farbu v troch zložkách plus priehľadnosť (alpha) – na čo používajú vysoký stupeň paralelizmu (veľký počet SIMD inštrukcií). Pri výpočtoch využívajú vláknové riešenia, no nemajú veľkú spoločnú vyrovnávaciu pamäť (cache).




*NVIDIA Tesla multiprocesor(podľa [5])*

Procesor NVIDIA Tesla (na obrázku) pozostáva zo 16 multiprocesorov. Jeden multiprocesor SM (Streaming Multiprocessor) je samostatnou výpočtovou jednotkou, obsahuje vlastnú údajovú i inštrukčnú vyrovnávaciu pamäť, 8 skalárnych procesorov SP procesorov (*Streaming Processor*), každý s jednou celočíselnou jednotkou a jednou jednotkou pre prácu s číslami s pohyblivou rádovou čiarkou , 2 špeciálne jednotky na výpočet zložitejších geometrických funkcií SFU (*Special Function Unit*) a spoločnú zdieľanú pamäť 16 kB.

Každé dva SM procesory zdieľajú spoločnú textúru prostredníctvom TPC klastra (*Texture/Processor Cluster*). Osem TPC tvorí potom celé pole procesorov SPA (*Streamig Processor Array*). Komunikácia s CPU je cez PCI-Express zbernicu.

Práca je organizovaná vo zväzkoch 32 vláken (*warp*), ktorým sa pridelí jeden blok, kde sú paralelne spracovávané. Každý SP spracováva inštrukcie v 1024 vlastných 32-bitových registroch. Celý procesor podporuje spracovanie 24 warpov naraz. Teoretický maximálny výkon pri frekvencii 1,5 GHz je asi 576 Gflops.

Výpočty na grafických procesoroch sa označujú ako GPGPU (General Purpose computing on Graphics Processing Units). Na popis práce vláken sa pre pocesory NVIDIA používa jazyk CUDA (Compute Unified Device Architecture). Ak sa podarí algoritmus dobre paralelizovať, jeho vykonanie na GPU je oproti vykonávaniu v CPU 10 – 100 násobne rýchlejšie.

## 5.5 Porovnanie výkonnosti procesorov

Na porovnanie výkonu procesorov sa používajú rôzne testovacie programy (benchmarky). Z najznámejších je testovanie neziskovou organizáciou SPEC (Standard Performance Evaluation Corporation). Výsledky testovania pre jednotlivé procesory testami SPEC CPU2006 možno nájsť na www.spec.org . SPEC skóre pre celočíselné operácie sa stanovuje podľa testov SPECint2006 a pre operácie s pohyblivou rádovou čiarkou podľa SPECfp2006.

Rýchlosť výpočtu procesora môžeme určiť aj na základe frekvencie časovača systému. Frekvencia (udávaná v Hertzoch – napr. 2 GHz) ale hovorí len o počte taktov (*clock cycles*) za sekundu. Výkonnosť systému odhadneme presnejšie, keď vieme, koľko taktov trvá strojový cyklus a koľko strojových cyklov potrebujeme priemerne na spracovanie inštrukcie (náš návrh v prvej časti modulu používa štvortaktové strojové cykly a ak počítame priemerne päť strojových cyklov na inštrukciu, dostávame priemerne 20 taktov na inštrukciu, teda celkove 100 miliónov inštrukcií za sekundu).

Priemerný počet taktov, potrebných na spracovanie inštrukcie sa zvykol udávať aj v CPI (clock *cycles per instruction*). Konkrétnu výkonnosť (v počte spracovaných inštrukcií za sekundu) potom môžeme počítať pre ľubovoľnú frekvenciu časovača jeho delením hodnotou CPI. Dnešné procesory väčšinou využívajú zreťazenie inštrukcií, čo veľmi výrazne zefektívni výpočet. Pokiaľ sa k tomu pridá ešte paralelizmus, dostaneme hodnoty CPI pod jednotkou.

Lepšia organizácia práce a samozrejme paralelné spracovanie vedú k tomu, že sa tento parameter udáva dnes skôr v obrátenej hodnote – IPC – priemerným počtom ukončených inštrukcií za strojový cyklus (pri jednoduchom paralelizme s dvoma nezávislými vláknami dostanem 2 IPC). V konečnom dôsledku rýchlosť (v počte inštrukcií za sekundu) dostanem potom prenásobením IPC frekvenciou časovača. Výkon sa zvykne udávať v jednotkách MIPS (million instructions per second).

Prvýkrát bol výkon 1 Gflops prekonaný 4-procesorovým superpočítačom CRAY X-MP v roku 1984.

Dnes taký výkon ponúkajú bežné kancelárske počítače.

Pri stanovovaní počtu vykonaných inštrukcií sa berú do úvahy len obyčajné celočíselné operácie. Existuje aj variant hodnotenia výkonu pre prácu s číslami v pohyblivej desatinnej čiarke, nazývaný flops (floating point instructions per second). Hodnota Gflops znamená vykonanie miliardy inštrukcií s pohyblivou rádovou čiarkou za sekundu. Pre súčasné typy dvojjadrových procesorov sa pohybuje tento výkon (v závislosti na základnej frekvencii procesora) niekde v hodnotách 30 Gflops (pokiaľ vezmeme na porovnanie 350 násobení za sekundu v prvom elektronickom univerzálnom počítači ENIAC, výkon vzrástol za 50 rokov 100 miliónov ráz).

kilo (k) – $10^3$ - tisíc
mega (M) – $10^6$ - milión
giga (G) – $10^9$ - miliarda
tera (T) – $10^{12}$ - bilión
peta (P) – $10^{15}$ - biliarda
exa (E) – $10^{18}$ - trilión

Výkonné grafické procesory sú pre svoje špecifické zameranie oveľa výkonnejšie - napr. výkon grafického adaptéra ATI Radeon HD 5870 dosahuje 3 Tflops.

Predpona peta (P) vyjadruje násobok $10^{15}$ - teda biliarda.

Poradie výkonnosti veľkých výpočtových systémov (postavených obvykle z viacerých procesorov) je určované pomocou Linpack benchmarkov (http://www.netlib.org/benchmark/hpl/) a možno ho nájsť na http://top500.org. Od novembra 2009 kraľuje systém Jaguar z Oak Ridge National Laboratory (Tennessee, USA) na zariadení Cray XT5-HE s 224162 jadrami, rozloženými v 18 688 uzloch s dvoma šesťjadrovými procesormi AMD Opteron a 12 GB RAM pamäťou, ktorý po prepočítaní dosahuje výkon 1,759 Pflops (teda 1 759 000 000 000 000 inštrukcií s pohyblivou rádovou čiarkou za sekundu).

### Čo sme sa naučili

V záverečnej časti sme získali stručný prehľad o používaných architektúrach procesorov súčasného sveta, o ich uplatnení a možnostiach porovnávania ich výkonnosti.

# Čo sme sa naučili v tomto module

## Zhrnutie

V tomto module sa účastníci oboznámili s činnosťou procesorovej jednotky počítača na úrovni logických obvodov a úrovni riadenia strojovými inštrukciami.

Absolvent modulu vie popísať priebeh strojového a inštrukčného cyklu a pozná rozdiel medzi úrovňou strojových inštrukcií a mikroinštrukcií.

Pozná charakteristické črty architektúry Intel x86 a vie začleniť inštrukcie v strojovom kóde do vyššieho programovacieho jazyka v prostredí Lazarus.

Pozná problémy urýchlenia spracovania inštrukcií zreťazením a využitím vyrovnávacích pamätí a má predstavu o ich riešení.

Má základný prehľad o histórii a súčasnom stave používaných architektúr procesorov. Vie sa orientovať v základných parametroch používaných mikroprocesorov.

## Preverenie výstupných vedomostí

Účastník vie popísať priebeh strojového a inštrukčného cyklu pre konkrétne zadanie. Vie analyzovať rôzne parametre, súvisiace s výkonovými a funkčnými charakteristikami procesorov. Konkrétne zadania budú pripravené v elektronickej forme.

# Literatúra a použité zdroje


[1] Bernard, M.J., Hugon, J. (1986) *Od logických obvodů k mikroprocesorům*, SNTL Praha, 1986

[2] Gvozdjak, L., Hanulová, Ľ., Jankovičová, A., Molnár, Ľ. (1987) *Počítače a programovanie*, Alfa Bratislava, 1987

[3] Hillis, W.D. (1998) *The Pattern on the Stone*, Basic Books, New York 1998 (slov. preklad *Obrazce v kameni*, Kalligram, 2002, ISBN 80-71490-59-8)

[4] Jirovský, V. (2000) *Principy počítačů*, Matfyzpress Praha, 2000, ISBN 80-85863-51-0

[5] Patterson, D.A., Hennesy J.L. (2007) *Computer Organization and Design*, 3. ed., Morgan Kaufmann, 2007, ISBN 978-0-12-370606-5

[6] Predko, M., (2008) *Digitální elektronika (bez předchozích znalostí)*, CPress, 2008, ISBN: 978-80-251-2124-5

[7] Stallings, W. (2005) *Computer Organization and Architecture: Designing for Performance*, 7. ed., Prentice Hall, 2005, ISBN 978-0-13-185644-8

[8] Simulačný program Logisim dostupný pod licenciou GNU GPL: http://ozark.hendrix.edu/~burch/logisim/index.html

[9] Tanenbaum, A.S. (2005) *Structured Computer Organization*, 5. ed., Prentice Hall, 2005, ISBN 978-0-13-148521-1

[10] White, R.; Downs, T. (2002) *How Computers Work*, 6. ed., Pearson, 2002 (čes. preklad *Jak fungují počítače*, SoftPress, 2003, ISBN 80-86497-48-8)

[11] Dandamudi, P.S. (2003) Fundamentals of Computer Organization and Design, Springer, 2003, ISBN 978-0387952116

Tento študijný materiál vznikol ako súčasť národného projektu Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika v rámci Aktivity „Vzdelávanie nekvalifikovaných učiteľov informatiky na 2. stupni ZŠ a na SŠ“.

Autori © RNDr. Jozef Jirásek, PhD.

Názov Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika

Podnázov Počítačové systémy 2

Študijný materiál prešiel recenzným pokračovaním.

Recenzenti doc. Ing. Peter Fabián, PhD.
doc. Ing. Ľudovít Trajteľ, PhD.

Počet strán 40

Náklad 300 ks

**Prvé vydanie, Bratislava 2010**



Vydal Štátny pedagogický ústav, Pluhová 8, 830 00 Bratislava, v súčinnosti s Univerzitou Pavla Jozefa Šafárika v Košiciach, Univerzitou Komenského v Bratislave, Univerzitou Konštantína Filozofa v Nitre, Univerzitou Mateja Bela v Banskej Bystrici a Žilinskou univerzitou v Žiline

Vytlačil BRATIA SABOVCI, s r.o., Zvolen

**ISBN 978-80-8118-033-0**