Pro úplné laiky

1. Číselné soustavy

6.1. Desítková soustava

6.2. Dvojková soustava

1.1.1. Příklad

1011 (v dvojkové soustavě) =

$1*(2^3) + 0*(2^2) + 1*(2^1) + 1*(2^0) =$

8 + 0 + 2 + 1 = 11 (v desítkové soustavě)

Poznámka

Na světě je 10 druhů lidí - ti, co rozumějí dvojkové soustavě a ti, co jí nerozumějí.

1.1.2. Příklad

00110101 (= 53 desítkově)

10001100 (=140)

11000001 (=193) (v desítkové soustavě)

6.3. Šestnáctková soustava

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

1.1.3. Příklad

Mějme převést 1234_8 do šestnáctkové soustavy. Číslo nejdřív převedeme do desítkové soustavy takto:

$1234_8 = 1x8^3 + 2x8^2 + 3x8^1 + 4x8^0 = 1x512 + 2x64 + 3x8 + 4x1 = 668_{10}$

A dále pokračujeme do šestnáctkové soustavy:

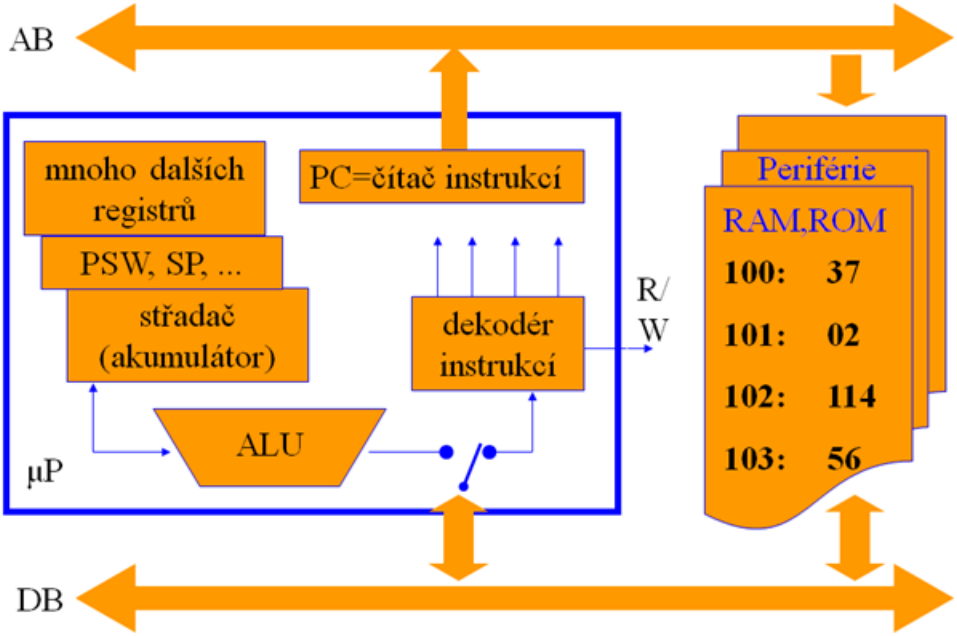
$668_{10} = 2x256 + 9x16 + 12x1 = 2x16^2 + 9x16^1 + 12x16^0 = 29C_{16}$

6.4. Doplňkový kód

V doplňkovém kódu je nula pouze 0000 0000, -1 je 1111 11111, -2 je 1111 1110, -3 je 1111 1101, a nejnižší možné číslo -128 je 1000 0000. Největší číslo pro byte se znaménkem je 127 (0111 1111).

2. Jak počítá počítač

6.5. Základní funkční bloky



Základní funkční bloky PC jsou **mikroprocesor** (µP), **paměti** (RAM, ROM), **periferní zařízení** (např. diskové jednotky, …)

navzájem jsou propojeny 3 sběrnicemi:

- **řídicí** (zde nakreslen jen vodič R/Ŵ)
- **adresová**
- **datová**

6.6. Adresová sběrnice

Jednosměrná: µP ->periférie

šířka 32 bitů: adresovací prostor 2^{32}

starší mikroprocesory: šířka jen 20 (24) bitů, tedy adresovací prostor 2^{20} = 1MB, zcela nedostačující, nutnost stránkování

6.7. Datová sběrnice

Obousměrná:

V každém okamžiku ale může mít jen jeden směr: µP ->periférie nebo periférie ->µP

o směru toku dat rozhoduje vodič R/Ŵ z řídicí sběrnice

směr (*čtení/zápis*) se vždy posuzuje z hlediska mikroprocesoru

6.8. Mikroprocesor

Obsahuje mnoho specializovaných pamětí (= registrů)

pro nás jsou důležité:

- střadač (=**akumulátor**),
- **čítač instrukcí** PC (=program counter, PC),
- **ukazatel zásobníku** (= stack pointer, SP),
- **stavový registr** (=Processor Status Word, PSW)
-

Dále obsahuje **ALU** = aritmeticko-logická jednotka, ta provádí výpočty (výsledek vesměs ukládá do střadače)

3. Instrukční cyklus

Popis instrukčního cyklu zahájíme tím, že čítač instrukcí (PC) obsahuje adresu instrukce, která se má provést. Tuto instrukci je potřeba nejprve načíst do µP (tzv. Fetch)

Obsah PC se po adresové sběrnici rozšíří do všech periférních zařízení

Ale adresa je jedinečná, proto jen 1 zařízení na ni může reagovat

Představme si, že adresa je 100

Na adresu 100 zareaguje paměť ROM. Zjistí, že na adrese 100 je hodnota 37. Protože vodič R/Ŵ je ve stavu Read, tak paměť hodnotu 37 přiloží na datovou sběrnici.

Hodnota z paměti se po datové sběrnici rozšíří po celém počítači, tedy i do µP

Uvnitř µP se hodnota načte do dekodéru instrukcí.

Dekodér instrukcí má za úkol rozšifrovat, o jakou se jedná instrukci a podle toho zařídit její provedení

Po vykonání instrukce čítne PC o +1

Takže na adresové sběrnici se objeví další adresa (101) a celý proces se opakuje

Většina instrukcí není tak jednoduchých

Kdyby se například jednalo o instrukci „*přičti číslo, které následuje, k obsahu střadače*“, muselo by se nejdříve načíst to „číslo, které následuje“.

To znamená, že instrukční cyklus by pokračoval: PC by na adresovou sběrnici přiložil adresu 101,

Paměť by na datovou sběrnici přiložila obsah adresy 101, v našem případě číslo 02

Číslo 02 by se načetlo do µP. Tentokrát by se ale hodnota nenačetla do dekodéru instrukcí, ale přes přepnutý vnitřní přepínač by se přivedla do ALU

ALU provede výpočet. Výsledek většinou zůstane ve střadači

Kdyby šlo o instrukci, která má data zapsat do paměti, průběh by byl obdobný až na to, že vodičem R/Ŵ by se namísto čtení z adresy 101 vyvolal zápis do adresy 101

Tedy: program je posloupnost čísel, která mohou znamenat jak data, tak instrukce - rozhoduje o tom pořadí (první je vždycky instrukční kód)

4. Nepodmíněný skok

Popsali jsme si průběh programu, který je zcela lineární. Takový program by nebyl k velkému užitku

(Nepodmíněné) skoky na jinou adresu se provádějí tak, že se do čítače instrukcí PC vloží cílová adresa skoku (často načtená z paměti, nebo vypočtená)

Program pak pokračuje od jiné adresy

5. Podmíněný skok

Důležité jsou skoky podmíněné - provádějí se jen tehdy, když nastaly určité podmínky

µP obsahuje sadu jednobitových pamětí, sestavených do registru PSW. Po každé instrukci se v PSW nastaví bity podle výsledku instrukce (např. jestli je výsledek nula, záporný, došlo k přetečení apod.)

Podmíněné skoky se provedou, když je v PSW správná hodnota

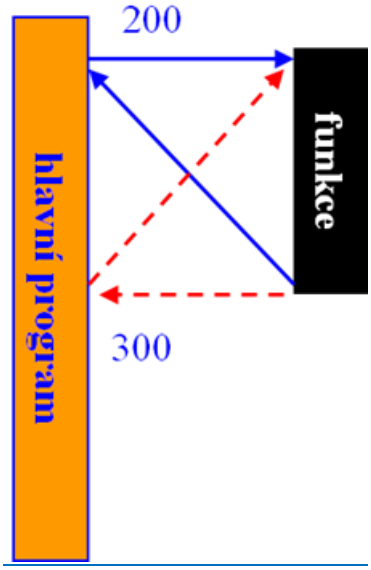
6. Funkce

6.1. Volání funkce

Často potřebujeme opakovat nějakou část programu - např. výpočet nějaké matematické funkce

Jednoduché: na začátek funkce by šlo skočit z mnoha míst

Ale problém: jak zajistit, aby skok **na konci** funkce směřoval, kam má?!



Nápad: při skoku na podprogram si poznamenat, odkud se skok prováděl

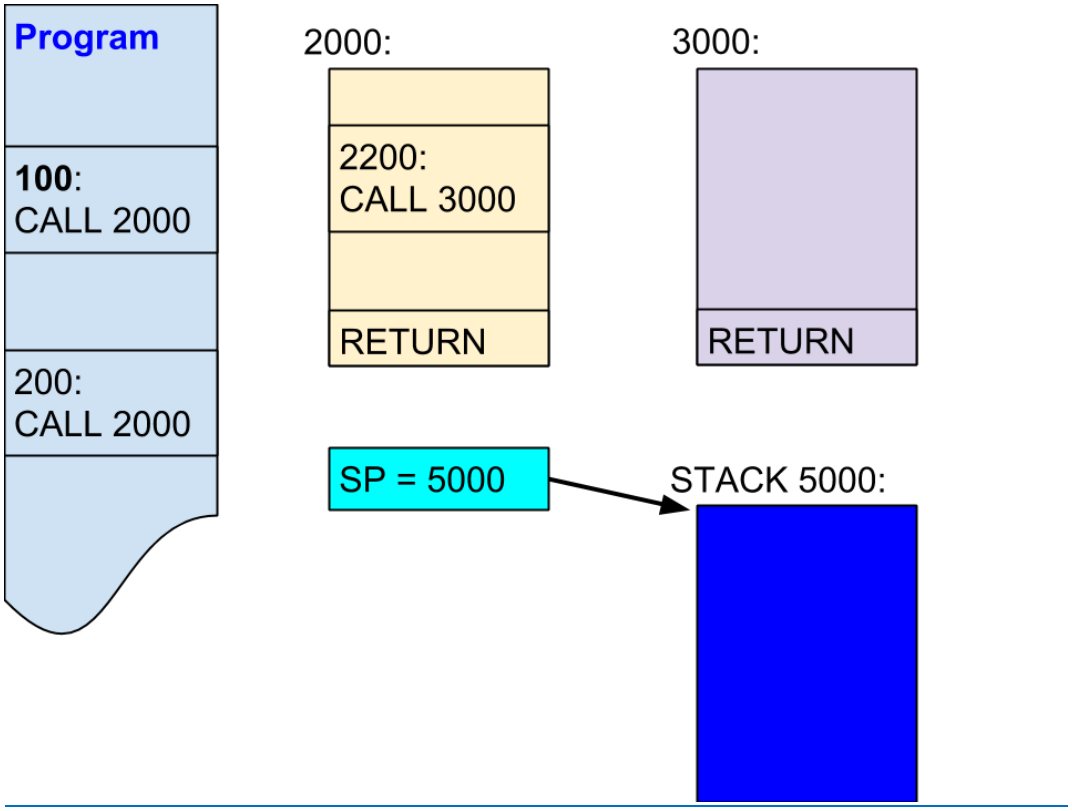
Nevýhoda: kdyby si µP poznamenával návratové adresy do registrů, mohl by realizovat jen tolik úrovní podprogramů, kolik by měl registrů ☹

Figure 1 Zásobník (stack)

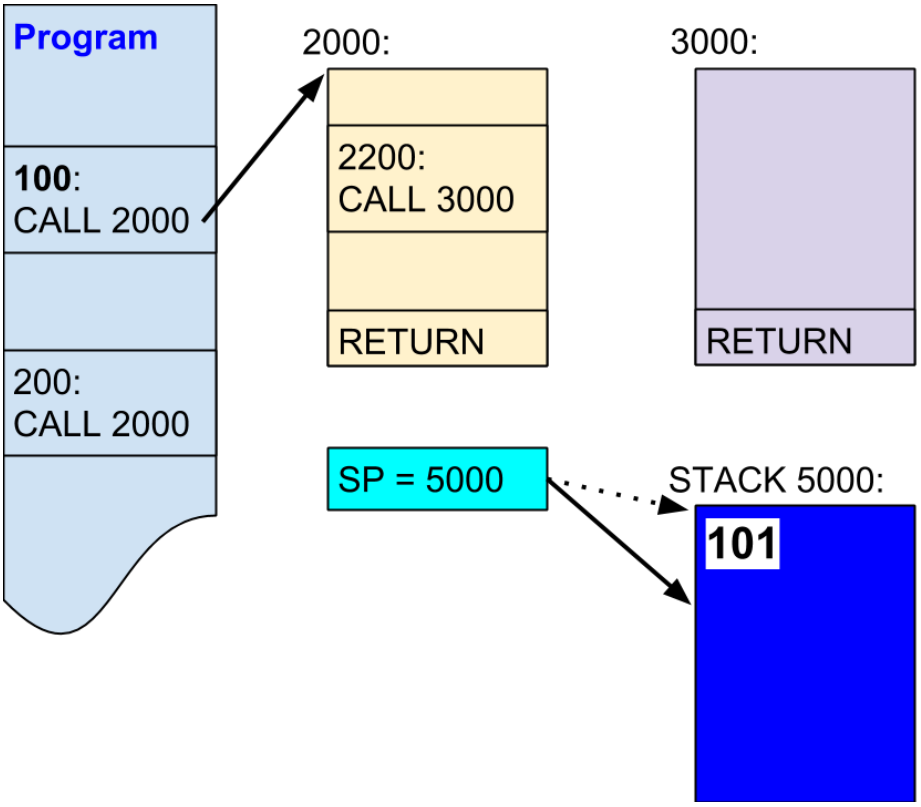
Řešení: v paměti RAM vytvořit **zásobník** (*stack*), velikost podle potřeby

Registr SP (=Stack Pointer) bude **vždy ukazovat na vrchol zásobníku**.

Na začátku např. SP=5000



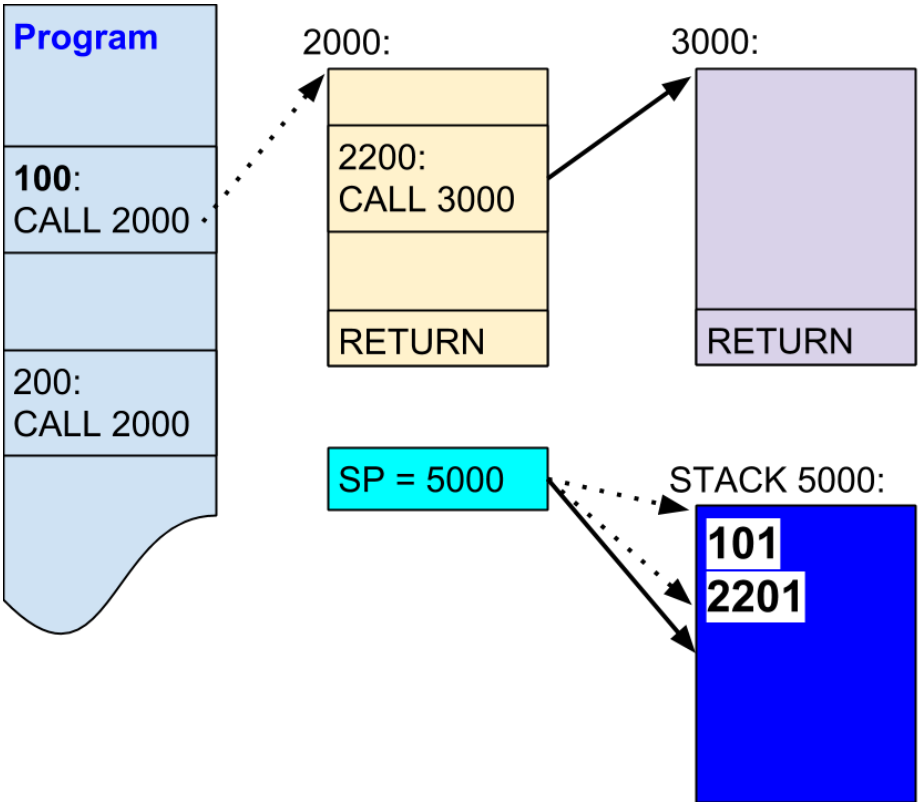
Zavedeme novou instrukci, **volání** podprogramu (Call).



Kdyby v hlavním programu na adrese 100 bylo **volání** funkce, tak se:

- **návratová adresa** (v našem případě 101) zapíše do zásobníku tam, kam ukazuje SP (to znamená na adresu 5000)
- **posune SP** tak, aby zase ukazoval na volné místo (tedy na 5001)
- **provede skok na podprogram** (v našem příkladu na adresu 2000)

Kdyby uvnitř funkce (třeba na adrese 2200) bylo další volání (jiného) podprogramu, začínajícího na 3000, volání by proběhlo zcela obdobně

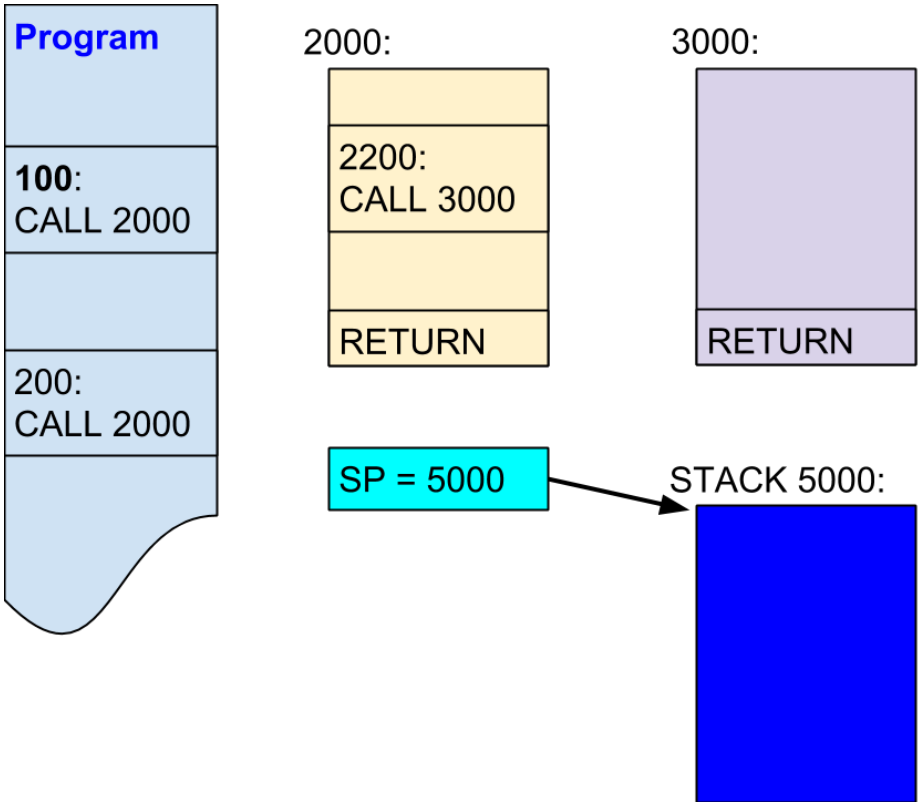


Návrat z podprogramu

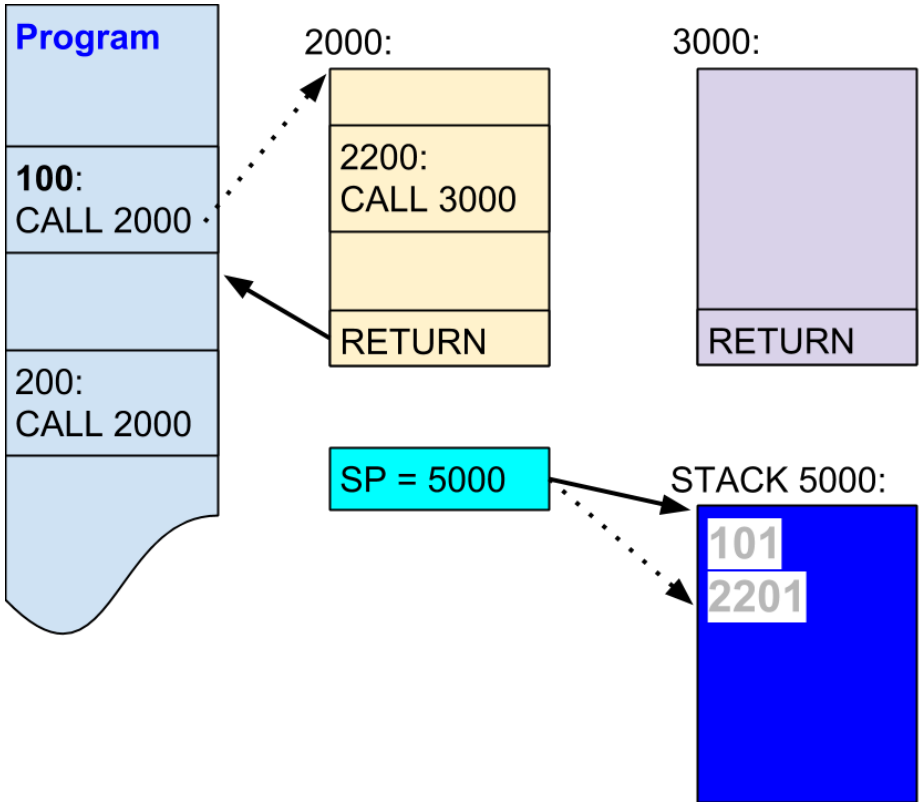
Na konci podprogramu musí být zvláštní instrukce **RET** = návrat z podprogramu

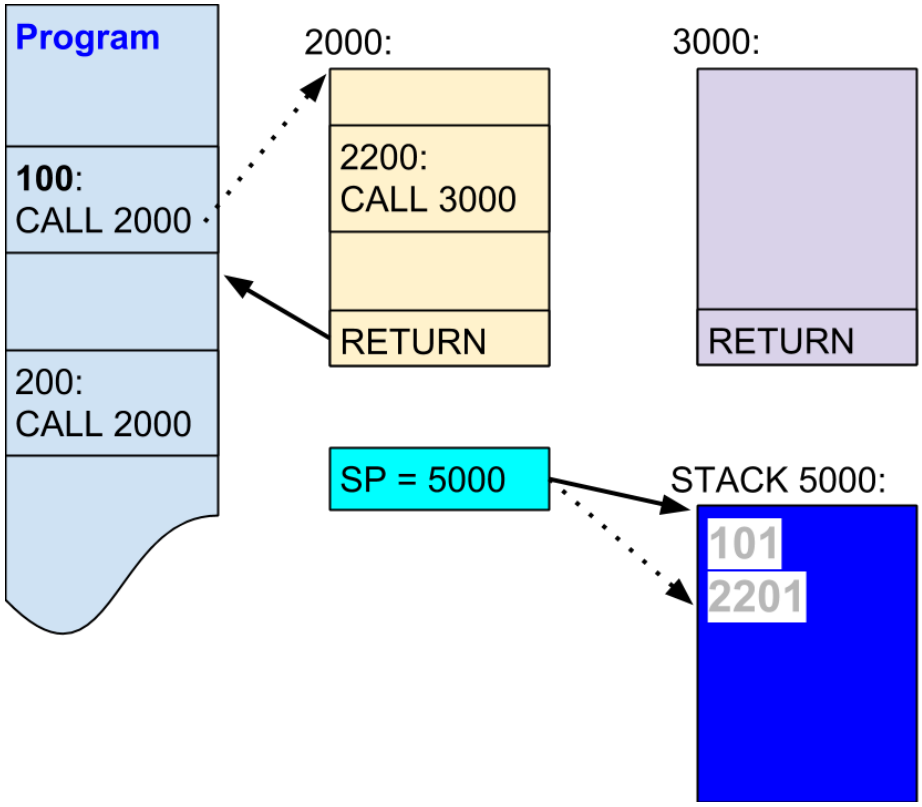
Operace se zásobníkem se dělají inverzně k volání, to znamená:

- SP se vrátí na předcházející pozici (je tam číslo 2201)

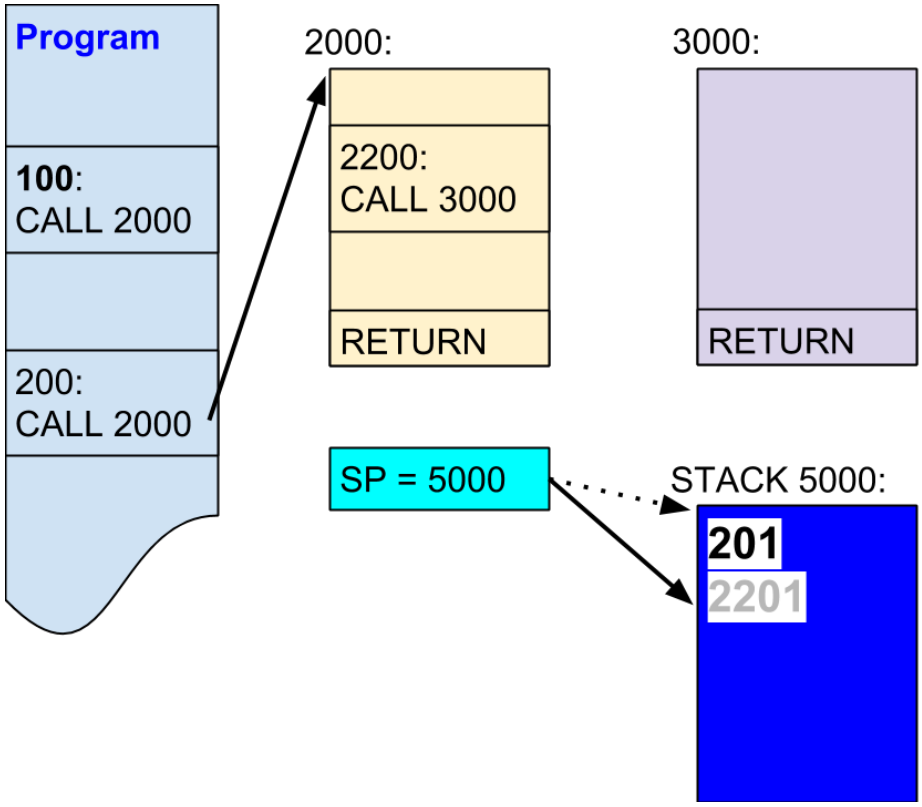


- obsah té pozice (číslo 2201) se načte
- a použije se jako adresa, na kterou se skočí.
- Takže skok na adresu 2201 znamená, že jsme se vrátili těsně za poslední volání podprogramu
- Až na konci podprogramu najdeme RET, postup se opakuje
- Takže nastane návrat na adresu 101, tedy za volání funkce.


Program
100:
CALL 2000
200:
CALL 2000
2000:
2200:
CALL 3000
RETURN
3000:
RETURN
SP = 5000
STACK 5000:
101
2201



Povšimněte si, že čísla v zásobníku se nemažou. Není to potřeba, protože jsou přepsána novými, jakmile dojde k dalšímu volání. Na obrázku je ukázána situace, kdy je znovu volána funkce 2000, ale z jiného místa programu.



1.1.4. Teorie a praxe

Popsaný postup byl jen schéma; realita se dost liší:

- zásobník (*stack*) roste k **menším adresám**, t.zn. narůstá nahoru,
- návratové adresy mají více bytů, a proto zásobník roste/klesá po více bytech,
- na zásobník se ukládají i četné jiné údaje.

6.2. Přerušení

Typickým příkladem, kdy se pracuje s přerušením, je obsluha pomalých zařízení (tiskárna)

Přerušení funguje podobně jako volání procedury; vyvolává se však hardwarově

Na rozdíl od CALL je přerušení asynchronní, může nastat kdykoliv ⇨ musíme se postarat o PSW ⇨ na zásobník se ukládá i PSW ⇨ pro návrat musí být zvláštní instrukce **IRET**

6.3. Zapnutí počítače

Jakmile je zapnuto napájení, mikroprocesor se připraví k činnosti a jakmile mu to je dovoleno (hardwarovým signálem na vodiči reset), začne pracovat právě popsaným způsobem od startovací adresy.

Startovací adresa se nachází v paměti ROM. Program je v bootovací ROM uložen na hardwarové úrovni: nejde ho změnit, nemusí se nijak inicializovat a je k dispozici okamžitě po zapnutí napájení.

Program ROM provádí základní oživení počítače: zjistí vybavení a konfiguraci základní desky, otestuje paměť RAM, zjistí, odkud má „natáhnout“ operační systém atd. Toto je ten program, se kterým primitivním způsobem komunikujete, když zadáváte, jestli se operační systém má natáhnout z disku, z *flash* paměti anebo ze sítě.

Program ukončí svou práci tím, že loads operační systém. Operační systém je dlouhý a složitý systém, navíc existuje celá řada různých operačních systémů. Ale existuje zde určitá unifikace: na disku existuje dohodnutá stopa, ve které je krátký *zaváděcí program*. Je pro všechny operační systémy na stejném místě a dělá stejnou činnost: načte a spustí zbytek operačního systému.

Operační systém se pak postará o celou další práci počítače.

7. Architektura von Neumannova

Popsaná architektura počítače se nazývá von Neumannova

Další klasická architektura je harvardská (má oddělenou sběrnici pro přenos dat od sběrnice, po které se přenáší program – vyšší rychlost, zejména při zpracování dat)

Moderní vývoj zaměřen na zvyšování výkonu ⇨ hlavně víceprocesorová řešení

Příklad: Zajímavou multiprocesorovou architekturu používá společnost Google.

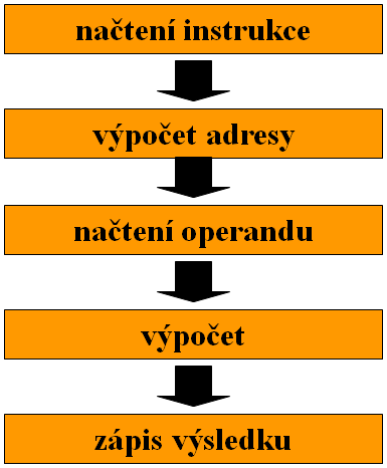
8. Víceprocesorové počítače

Hlavní problém: i malé množství sekvenčních instrukcí dokáže výrazně zpomalit výpočty

Několik koncepcí, např.

- tatáž instrukce x N procesorů (operují na různých datech)
- maticové procesory (hyperkrychle)
- RISC (=*Reduced Instruction Set Computer*)

9. RISC



zjednodušená sada instrukcí: všechny instrukce mají stejný počet taktů (např. 5)

současně se vykonává několik (např. 5 instrukcí), ale každá je v jiné fázi

důsledek: během každé fáze se dokončí 1 instrukce

Vývoj programovacích prostředků...

1. Historie

Na úplném začátku bylo "drátování" programu do hardware

Objev, že program lze uložit do paměti podobně jako data:

- strojový kód
- assembler (=jazyk symbolických adres)
- vyšší programovací jazyky (FORTRAN, BASIC, COBOL,...)

- Snaha odstranit omezení, která na programátora kladl programovací jazyk:
- generace jazyků umožňujících "skoro všechno" (Fortran 77, PL/1, ADA)
- Zklamání: programátor věnuje psaní kódu jen 20% času, pouhých 10% vymýšlení algoritmů a celých 70% ladění !
- Snaha definovat jazyky, které by automatizovaly ladění:
- jazyky 3. generace (PASCAL, ...)
- Problém: tyto jazyky nepodporují spolupráci více programátorů na jednom projektu!

2. Pokusy o řešení:

- CASE TOOLs a CASE - prostředky pro organizační podporu programátorské práce
- modulární programování (MODULA, SIMULA62)
- nealgoritmické programování (jazyky 4GL, programming by example, AI metody, deklarativní programování).

3. Krize programování (již přes 50 let)

Statistiky USA z počátku 80.let ukazují, že:

- 2% programů se používají tak, jak byly vytvořeny,
- 2-3% se používají po mírném dopracování, nepřekračujícím 10-15% zdroj. textů,
- 20% bylo nutno přepracovat zásadním způsobem,
- 20% vráceno a přepracováno (vesměs na základě nových kontraktů),
- **50% zákazník program nikdy nepoužil**
- **5% program shledán nepoužitelným**

Výsledky graficky znázorňuje obrázek.

používají se, jak byly vytvořeny
používají se po mírném dopracování
bylo nutno přepracovat
vráceno a přepracováno
zákazník nikdy nepoužil
program shledán nepoužitelným

4. Souvislosti

6.1. Hledání cest z krize

Hledání cest z krize programování vede jednak směrem na revoluční změny hardware (multiprocesory, neuronové sítě,...), jednak na objektové architektury počítačů. Tyto architektury řeší 2 zásadní problémy:

- problém primitivnosti paměti (nelze rozeznat, co paměťová buňka obsahuje, resp. její obsah můžeme různě interpretovat: byte x char x 1/2 word aj.),

- problém globálnosti paměti (i ta nejmenší změna v paměti mění celkový stav paměti => dosah změny je těžko lokalizovatelný).

5. Program = data + algoritmy

6.1. Data a datové typy

Příklad: s čísly se pracuje úplně jinak, než s písmeny. Čísla jde násobit (3*7=21), písmena nikoliv. Písmena jde seskupovat do slov, čísla těžko.

Moderní programovací jazyky zavádějí „striktní typovou kontrolu“. Cílem je, co nejdřív odhalit chybnou manipulaci s daty. A odhalit to automaticky, při překladu.

Příklad: když kompilátor odhalí, že se pokoušíme vynásobit dva texty, tak je asi něco špatně ☺

Program = data + algoritmy

Algoritmus = pracovní postup

Definice algoritmu

Algoritmus je exaktní postup, řešící určitou třídu úloh neboli problémovou oblast, který je reprezentován konečnou posloupností příkazů. Algoritmus garantuje, že po provedení konečného počtu kroků na množině přípustných vstupních dat bude generován požadovaný výsledek.

Akademičtější definice ;-)

Algoritmus lze chápat jako implementaci transformační funkce f, která provádí zobrazení z množiny vstupních dat (A) do množiny výstupních dat (B). Zapsáno matematicky:

f: A → B.

6.2. Historie

první zmínky o algoritmu a algoritmickém řešení algebraických úloh se datují do doby kolem roku 810 n. l., kdy se objevily v pracích uzbeckého matematika Al-Chórezmího.

Příklad – kvadratická rovnice

Rovnici ve tvaru ax2 + bx +c = 0 (přičemž a≠0) začneme v oboru reálných čísel řešit vypočtením diskriminantu (D), na jehož základě posléze stanovíme množinu kořenů.

Ze středoškolské matematiky si pamatujeme vzorec, jenž nám pomůže s určením diskriminantu: D=b2-4ac.

Jestliže je D<0, množina kořenů (K) je v oboru reálných čísel (R) disjunktní (prázdná). Algoritmus výpočtu nás tedy dovádí k výsledku K = Ø.

V případě, že je diskriminant roven nule, existuje jeden (dvojnásobný) kořen, který najdeme následovně: K= - b/2a

Za těchto okolností je množina kořenů tvořena jedním prvkem, K = {x}.

Při kladné hodnotě diskriminantu existují právě dva kořeny, které vyhovují řešení kvadratické rovnice. Jejich nalezení je otázkou rozluštění níže popsaných rovnic:

x1= (-b+ √D)/2a

x2= (-b- √D)/2a

Pro množinu kořenů pak platí K = {x1, x2}.

6.3. Vlastnosti algoritmů

Kritéria pro algoritmus

1. Všeobecnost
2. Jednoznačnost
3. Konečnost
4. Rezultativnost
5. Správnost
6. Opakovatelnost
7. Efektivnost

6. Hodnocení algoritmů

Časová efektivnost algoritmů.

Jaký význam by kupříkladu měl algoritmus, který by sice splňoval požadavek konečnosti realizovaných kroků, ovšem jehož doba zpracování by zabrala 1 rok?

6.4. Paměťová efektivnost algoritmů.

Významnost paměťové efektivnosti algoritmů se pojí zejména s minulostí, kdy byla operační paměť vpravdě nedostatkovým zbožím.

6.5. Další vlastnosti

Jeden úkol (problém) lze obvykle vyřešit mnoha různými algoritmy.

Příklad: prvočísla jdou najít více způsoby (hrubá síla x Eratosthenovo síto)

Pro některé problémy neexistují algoritmická řešení.

Příklad: neexistuje algoritmus, jak napsat „krásnou“ hudbu

6.6. Faktorizace

O faktorizaci se zmíním jako o příkladu algoritmu, a to hlavně proto, že se s ní budeme později setkávat.

Základní věta aritmetiky, kterou vyslovil již Eukleidés, praví, že každé přirozené číslo lze vyjádřit v podobě součinu jednoznačně daných prvočísel.

(Na prvočísla tak můžeme nahlížet jako na stavební prvky, z nichž jsou složena všechna ostatní přirozená čísla.)

Faktorizace libovolného přirozeného čísla je prakticky neproveditelná v případech, kdy je toto číslo součinem příliš velikých prvočísel.

Na tom je založena například kryptografie.

6.7. Příklad faktorizace

Máte číslo 33729211 a máte ho rozložit na součin dvou prvočísel.

Na tuto úlohu známe jen jediný algoritmus, hledání „hrubou silou“ (brute force). To znamená, vyzkoušet všechny možnosti.

Otázka k zamyšlení: kde je horní hranice cyklu?

Výsledek je zde

Cvičení

Vyzkoušejte si testy v Moodle.

Stáhněte si applianci k virtuálnímu počítači.

Nainstalujte si virtuální počítač a importujte do něj aplianci.

Prohlédněte si multi-počítačovou a multiprocesorovou architekturu u firmy Google.

Testy

Převeďte číslo 1234 na binární

10011010010

do zásobníku

Převeďte číslo -52 na druhý doplněk (1 byte)

11001100

do zásobníku

do paměti flash

do paměti ROM

do akumulátoru (střadače)

do ukazatele zásobníku

PSW je:

specializovaný registr se stavem procesoru

specializovaná část operační paměti

pointer na slovo zásobníku

Základní vlastnosti algoritmů jsou: >

Všeobecnost

Jednoznačnost

Konečnost

Rezultativnost

Správnost

Opakovatelnost

Efektivnost

Adaptivnost

Uzavřenost

Dědičnost

Přenositelnost

Operativnost

Senzitivnost

Obsah