8. lekce

Úvod do jazyka C – 2. část
Operátory a bitové funkce
Editace a kompilace programu

Schéma počítače

Miroslav Jílek

Operátory

Operátory

= **přiřazení**, např.: **a = 5;**

Operátory

=	**přiřazení**, např.:	**a = 5;**
+	**sčítání**, např.:	**c = a + 1;**

Operátory

=	**přiřazení**, např.:	**a = 5;**	
+	**sčítání**, např.:	**c = a + 1;**	
+=	**přičtení**, např.:	**a += 5;**	(stejné jako a = a + 5;)

Operátory

=	**přiřazení**, např.:	**a = 5;**	
+	**sčítání**, např.:	**c = a + 1;**	
+=	**přičtení**, např.:	**a += 5;**	(stejné jako a = a + 5;)
++	**inkrementace**, např.	**a**++ nebo ++**a**	(stejné jako a = a + 1;)

Operátory

=	**přiřazení**, např.:	**a = 5;**	
+	**sčítání**, např.:	**c = a + 1;**	
+=	**přičtení**, např.:	**a += 5;**	(stejné jako a = a + 5;)
++	**inkrementace**, např.	**a**++ nebo ++**a**	(stejné jako a = a + 1;)

(v běžném příkazu je stejný výsledek, v případě použití jako parametr funkce např. ***function MojeFunkce (++a)*** *se nejprve přičte +1 a až následně se zavolá funkce s novou hodnotou proměnné a, v případě* ***function MojeFunkce (a++)*** *se nejprve zavolá funkce s původní hodnotou a až po provedení funkce se hodnota proměnné a zvýší o 1)*

Operátory

= **přiřazení**, např.: **a = 5;**

+ **sčítání**, např.: **c = a + 1;**

+= **přičtení**, např.: **a += 5;** (stejné jako a = a + 5;)

++ **inkrementace**, např. **a++** nebo **++a** (stejné jako a = a + 1;)
(v běžném příkaze je stejný výsledek, v případě použití jako parametr funkce např. ***function MojeFunkce (++a)*** *se nejprve přičte +1 a až následně se zavolá funkce s novou hodnotou proměnné a, v případě* ***function MojeFunkce (a++)*** *se nejprve zavolá funkce s původní hodnotou a až po provedení funkce se hodnota proměnné a zvýší o 1)*

- **odčítání**, např.: **a = b – 1;**

Operátory

= **přiřazení**, např.: **a = 5;**

\+ **sčítání**, např.: **c = a + 1;**

+= **přičtení**, např.: **a += 5;** (stejné jako a = a + 5;)

++ **inkrementace**, např. **a++** nebo **++a** (stejné jako a = a + 1;)
*(v běžném příkaze je stejný výsledek, v případě použití jako parametr funkce např. **function MojeFunkce (++a)** se nejprve přičte +1 a až následně se zavolá funkce s novou hodnotou proměnné a, v případě **function MojeFunkce (a++)** se nejprve zavolá funkce s původní hodnotou a až po provedení funkce se hodnota proměnné a zvýší o 1)*

- **odčítání**, např.: **a = b - 1;**

-= **odečtení**, např.: **a -= 1;** (stejné jako a = a – 1;)

-- **dekrementace**, např.: **a --;** nebo **--a;**

(význam pro použití u funkcí je stejný jako u inkrementace)

-- **dekrementace**, např.: **a --;** nebo **--a;**
(význam pro použití u funkcí je stejný jako u inkrementace)

* **násobení**, např.: **a = b * 3;**

-- **dekrementace**, např.: **a --;** nebo **--a;**
(význam pro použití u funkcí je stejný jako u inkrementace)

* **násobení**, např.: **a = b * 3;**

*= **autonásobení**, např.: **a *= 3;** (stejné jako a = a * 3;)

-- **dekrementace**, např.: **a --;** nebo **--a;**
(význam pro použití u funkcí je stejný jako u inkrementace)

* **násobení**, např.: **a = b * 3;**

*= **autonásobení**, např.: **a *= 3;** (stejné jako a = a * 3;)

/ **dělení**, např.: **a = b / 3;**

-- **dekrementace**, např.: **a --;** nebo **--a;**
(význam pro použití u funkcí je stejný jako u inkrementace)

* **násobení**, např.: **a = b * 3;**

*= **autonásobení**, např.: **a *= 3;** (stejné jako a = a * 3;)

/ **dělení**, např.: **a = b / 3;**

/= **autodělení**, např.: **a /= 3;** (stejné jako a = a/3;)

-- **dekrementace**, např.: **a --;** nebo **--a;**
(význam pro použití u funkcí je stejný jako u inkrementace)

* **násobení**, např.: **a = b * 3;**

*= **autonásobení**, např.: **a *= 3;** (stejné jako a = a * 3;)

/ **dělení**, např.: **a = b / 3;**

/= **autodělení**, např.: **a /= 3;** (stejné jako a = a/3;)

% zbytek celočíselného dělení (modulo), např.: **a = b % 2;**

-- **dekrementace**, např.: **a --;** nebo **--a;**
(význam pro použití u funkcí je stejný jako u inkrementace)

* **násobení**, např.: **a = b * 3;**

*= **autonásobení**, např.: **a *= 3;** (stejné jako a = a * 3;)

/ **dělení**, např.: **a = b / 3;**

/= **autodělení**, např.: **a /= 3;** (stejné jako a = a/3;)

% **zbytek celočíselného dělení (modulo)**, např.: **a = b % 2;**

%= **automodulo**, např.: **a %= 2;**

-- **dekrementace**, např.: **a --;** nebo **--a;**
(význam pro použití u funkcí je stejný jako u inkrementace)

* **násobení**, např.: **a = b * 3;**

*= **autonásobení**, např.: **a *= 3;** (stejné jako a = a * 3;)

/ **dělení**, např.: **a = b / 3;**

/= **autodělení**, např.: **a /= 3;** (stejné jako a = a/3;)

% **zbytek celočíselného dělení (modulo)**, např.: **a = b % 2;**

%= **automodulo**, např.: **a %= 2;**

! **negace**, např.: **a = ! b;**
Pozor: Když je hodnota b = 0, pak negací je b = 1 !!!
Když je b není rovno 0, pak negace je b = 0.

&& **Logické And**, např.: **a = b && c;**

*V případě, že „b „ a také „c“ je **true**, pak je výsledkem a = 1, v ostatních případech je výsledkem a = 0. Za **true** je považovaná jakákoli číselná hodnota **různá od nuly**, **false** je **rovno nule**.*

V případě, že „b“ je false, pak je celý výraz false a „c“ se již nevyhodnocuje! Toho můžeme využít při řazení dílčích podmínek.

&& **Logické And**, např.: **a = b && c;**

*V případě, že „b „ a také „c“ je **true**, pak je výsledkem a = 1, v ostatních případech je výsledkem a = 0. Za **true** je považovaná jakákoli číselná hodnota **různá od nuly**, **false** je **rovno nule**.*

V případě, že „b“ je false, pak je celý výraz false a „c“ se již nevyhodnocuje! Toho můžeme využít při řazení dílčích podmínek.

|| **Logické Or**, např.: **a = b || c;**

*V případě, že „b „ a také „c“ je **false**, pak je výsledkem a = **0**, v ostatních případech je výsledkem a = **1**.*

V případě, že „b“ je true, pak je celý výraz true a „c“ se již nevyhodnocuje! Toho můžeme využít při řazení dílčích podmínek.

&& **Logické And**, např.: **a = b && c;**

*V případě, že „b „ a také „c“ je **true**, pak je výsledkem a = 1, v ostatních případech je výsledkem a = 0. Za **true** je považovaná jakákoli číselná hodnota **různá od nuly**, **false** je **rovno nule**.*

V případě, že „b“ je false, pak je celý výraz false a „c“ se již nevyhodnocuje! Toho můžeme využít při řazení dílčích podmínek.

|| **Logické Or**, např.: **a = b || c;**

*V případě, že „b „ a také „c“ je **false**, pak je výsledkem a = **0**, v ostatních případech je výsledkem a = **1**.*

V případě, že „b“ je true, pak je celý výraz true a „c“ se již nevyhodnocuje! Toho můžeme využít při řazení dílčích podmínek.

>, <, <=, >=, == (logická rovnost), **!=** (logická nerovnost) **Porovnání**

Výsledkem je pravda (true) nebo nepravda (false)

Bitové funkce

Bitové funkce

& **bitový And na dvě čísla**

např.: pro a = 5, b = 3 bude v binárním kódu a = 101, b = 011, výsledek ***a & b*** bude ***001***, protože 1&0=0, 0&1=0, 1&1=1

Bitové funkce

& **bitový And na dvě čísla**

např.: pro a = 5, b = 3 bude v binárním kódu a = 101, b = 011,
výsledek ***a & b*** bude ***001***, protože 1&0=0, 0&1=0, 1&1=1

| **bitový Or na dvě čísla**

např.: pro a = 5, b = 3, bude v binárním kódu a = 101, b = 011
výsledek ***a | b*** bude ***111***, protože 1|0=1, 0|1=1, 1|1=1

Bitové funkce

& **bitový And na dvě čísla**

např.: pro a = 5, b = 3 bude v binárním kódu a = 101, b = 011, výsledek ***a & b*** bude ***001***, protože 1&0=0, 0&1=0, 1&1=1

| **bitový Or na dvě čísla**

např.: pro a = 5, b = 3, bude v binárním kódu a = 101, b = 011 výsledek ***a | b*** bude ***111***, protože 1|0=1, 0|1=1, 1|1=1

<< **bitový posun doleva**

např.: ***a = b << 2;*** posune desetinnou čárku o dvě pozice v binárním kódu doprava (platné číslice se od desetinné čárky posunou doleva), jestliže $\mathbf{b = 5_{10}}$, pak $\mathbf{101_2}$ se změní na $\mathbf{10100_2}$, potom bude $\mathbf{a = 20_{10}}$

Bitové funkce

& **bitový And na dvě čísla**
např.: pro a = 5, b = 3 bude v binárním kódu a = 101, b = 011, výsledek ***a & b*** bude ***001***, protože 1&0=0, 0&1=0, 1&1=1

| **bitový Or na dvě čísla**
např.: pro a = 5, b = 3, bude v binárním kódu a = 101, b = 011 výsledek ***a | b*** bude ***111***, protože 1|0=1, 0|1=1, 1|1=1

<< **bitový posun doleva**
např.: ***a = b << 2;*** posune desetinnou čárku o dvě pozice v binárním kódu doprava (platné číslice se od desetinné čárky posunou doleva), jestliže $\mathbf{b = 5_{10}}$, pak $\mathbf{101_2}$ se změní na $\mathbf{10100_2}$, potom bude $\mathbf{a = 20_{10}}$

>> **bitový posun doprava**
např.: a = b >> 2; posune desetinnou čárku o dvě pozice v binárním kódu doleva, jestliže b = 5, pak 101 se změní na 1,01 a to je a =1+1/4

Typy závorek

( ) parametry funkcí

{ } ohraničují části kódu (bloky), pokud definujeme proměnnou uvnitř těchto závorek, pak platí (je platná) pouze uvnitř závorek!

[ ] definují indexy polí

Kombinace kláves

Alt Gr* + *D *zapíše* ***&***
Alt Gr* + *X *zapíše* ***#***
Alt Gr* + *W *zapíše* ***|***
Alt Gr* + *B *zapíše* ***{***
Alt Gr* + *N *zapíše* ***}***
Alt Gr* + *F *zapíše* ***[***
Alt Gr* + *G *zapíše* ***]***

Editace kódu programu

Komentář v kódu programu

// text	- pro jeden řádek
/*	- na začátku prvního řádku komentáře
*/	- na konci posledního řádku komentáře

Za každým řádkem, *pokud kód nepokračuje složenými závorkami****, musí být středník!***

Hlavní program:

```
int main (void)
{
    kód programu
    return 0;
}
```

int	funkce je typu integer (v hlavním programu musí být int)
return	v případě úspěšného běhu programu vrací nulu, jinak číslo chyby
(void)	bez vstupních parametrů (pokud žádné nepoužijeme)

nebo

```
int main (int argc, char ** args)
{
    kód programu
    return 0;
}
```

argc je počet parametrů (arguments count), args je pole parametrů, **args znamená odkaz na odkaz na char – to znamená, že pod odkazem je dvourozměrné pole charů

Procedura:

datový_typ jmeno_procedury (parametry)

… a dále jako u hlavního programu

Můj první program

```
program muj_prvni_program;
#include stdio.h
int main(void)
{
      printf(“Muj prvni program. Hura!\n”);
      return 0;
}
```

Můj druhý program

```
program muj_prvni_program;
#include stdio.h
int main(void)
{
      printf(“Zadej cele cislo “);
      int c;
      scanf(“%d“,&c);
      if (c<0)
      {
            printf(“To je zaporne cislo.\n“);
      }
      else if (c>0)
      {
            printf(“To je kladne cislo.\n“);
      }
      else
      {
            printf(“Cislo je nula.\n“);
      }
      return 0;
}
```

```
program muj_prvni_program;
#include stdio.h
int main(void)
{
	printf(“Zadej cele cislo “);
	int c;
	scanf(“%d“,&c);
	if (c<0)  printf(“To je zaporne cislo.\n“);else if (c>0) printf(“To je kladne cislo.\n“);else
		printf(“Cislo je nula.\n“);
	return 0;
}
```

Připojení souboru s kódem (program je napsán ve více souborech):

Program může být napsán ve více souborech.
To můžeme použít, když potřebujeme program rozdělit do menších celků, např. z důvodu potřeby rozdělení na logické celky.

#include “cesta k souboru“

Pokud je druhý soubor ve stejné složce, jako soubor první (hlavní):

např.: **#include “program2.c“** (Windows)
#include “./program2.c“ (Linux)

Editační prostření

Kód programu lze editovat v jakémkoli textovém editoru.

Editační prostření

Kód programu lze editovat v jakémkoli textovém editoru.
Postačí poznámkový blok.

Editační prostření

Kód programu lze editovat v jakémkoli textovém editoru.
Postačí poznámkový blok.
Na FIT je používaný editor Gedit.

Editační prostření

Kód programu lze editovat v jakémkoli textovém editoru.
Postačí poznámkový blok.
Na FIT je používaný editor Gedit.

Soubory s kódem programu se ukládají s koncovkou **.c**

Kompilace programu

Nejprve je třeba nainstalovat kompilátor **gcc**.

http://tdm-gcc.tdragon.net/download - dawnload kompilátoru gcc

Kompilace programu

Nejprve je třeba nainstalovat kompilátor **gcc**.

http://tdm-gcc.tdragon.net/download - dawnload kompilátoru gcc

Vytvořený zdrojový kód (soubor) musíme před spuštěním zkompilovat!

Kompilace programu

Nejprve je třeba nainstalovat kompilátor **gcc**.

http://tdm-gcc.tdragon.net/download - dawnload kompilátoru gcc

Vytvořený zdrojový kód (soubor) musíme před spuštěním zkompilovat!

Kompilace:

– otevřeme příkazový řádek ve složce, kde máme zdrojový soubor

Kompilace programu

Nejprve je třeba nainstalovat kompilátor **gcc**.

http://tdm-gcc.tdragon.net/download - dawnload kompilátoru gcc

Vytvořený zdrojový kód (soubor) musíme před spuštěním zkompilovat!

Kompilace:

- otevřeme příkazový řádek ve složce, kde máme zdrojový soubor
- spustíme příkaz: **gcc jménosouboru.c**

Kompilace programu

Nejprve je třeba nainstalovat kompilátor **gcc**.

http://tdm-gcc.tdragon.net/download - dawnload kompilátoru gcc

Vytvořený zdrojový kód (soubor) musíme před spuštěním zkompilovat!

Kompilace:

- otevřeme příkazový řádek ve složce, kde máme zdrojový soubor
- spustíme příkaz: **gcc jménosouboru.c**
- vygenerovaný soubor se vždy jmenuje **a.exe** (windows) nebo **a.out** (linux)

Kompilace programu

Nejprve je třeba nainstalovat kompilátor **gcc**.

http://tdm-gcc.tdragon.net/download - dawnload kompilátoru gcc

Vytvořený zdrojový kód (soubor) musíme před spuštěním zkompilovat!

Kompilace:

- otevřeme příkazový řádek ve složce, kde máme zdrojový soubor
- spustíme příkaz: **gcc jménosouboru.c**
- vygenerovaný soubor se vždy jmenuje **a.exe** (windows) nebo **a.out** (linux)

Příkaz lze také spustit s parametry, např.:

gcc -Wall -pedantic jménosouboru.c

Kompilace programu

Nejprve je třeba nainstalovat kompilátor **gcc**.

http://tdm-gcc.tdragon.net/download - dawnload kompilátoru gcc

Vytvořený zdrojový kód (soubor) musíme před spuštěním zkompilovat!

Kompilace:

- otevřeme příkazový řádek ve složce, kde máme zdrojový soubor
- spustíme příkaz: **gcc jménosouboru.c**
- vygenerovaný soubor se vždy jmenuje **a.exe** (windows) nebo **a.out** (linux)

Příkaz lze také spustit s parametry, např.:
gcc -Wall -pedantic jménosouboru.c

Tento způsob kompilace vypíše, kromě chyb, také varování (neinicializovaná proměnná, podmínka, která je vždy true, přiřazení proměnné jiného datového typu do proměnné s jiným datovým typem….).

Spuštění programu

Soubor s programem se spouští v příkazovém řádku otevřeného v příslušné složce příkazem:

jménosouboru.exe (windows)
./jmenosouboru.out (linux)

Otevření příkazového řádku:

- Windows 10 – Start – cmd
- Linux – Start – Administration - Terminal

Práce s příkazovým řádkem:

Windows:	**cd jméno_složky**	- vnoření do složky
	cd ..	- výstup do nadřazené složky
	dir	- výpis obsahu složky
Linux:	**cd jméno_složky**	- vnoření do složky
	cd ..	- výstup do nadřazené složky
	ls	- výpis obsahu složky

Poznámka: Pokud je jméno složky nebo souboru složeno z více slov oddělených mezerou, pak se název souboru nebo složky zapisuje v uvozovkách!

Např.: **cd “Nová složka“**

Schéma počítače

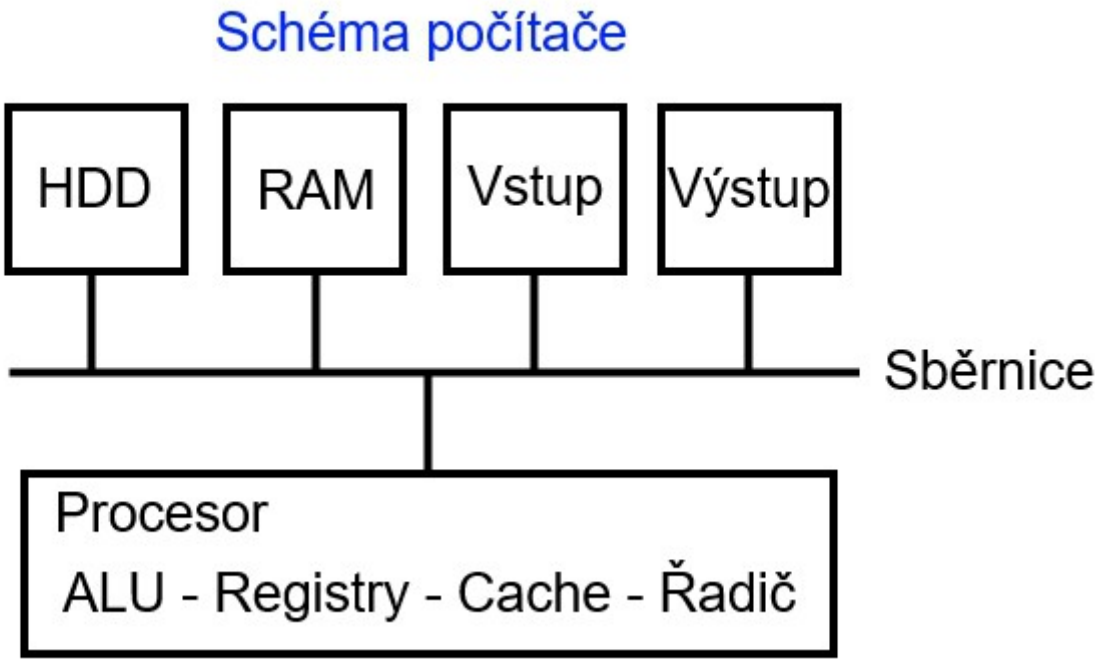
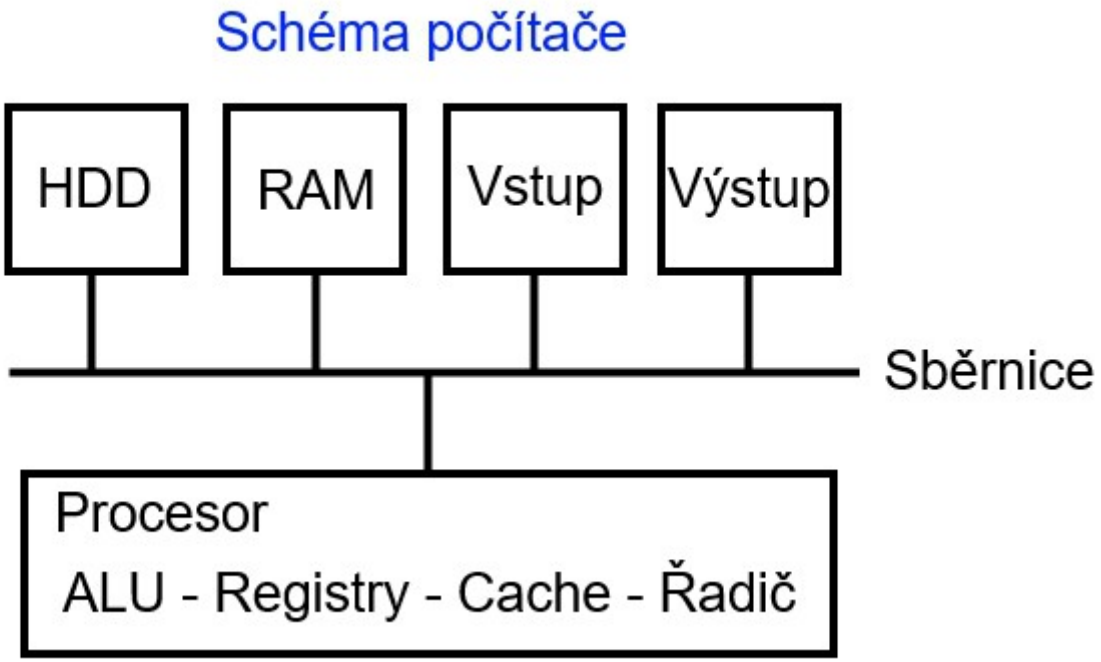
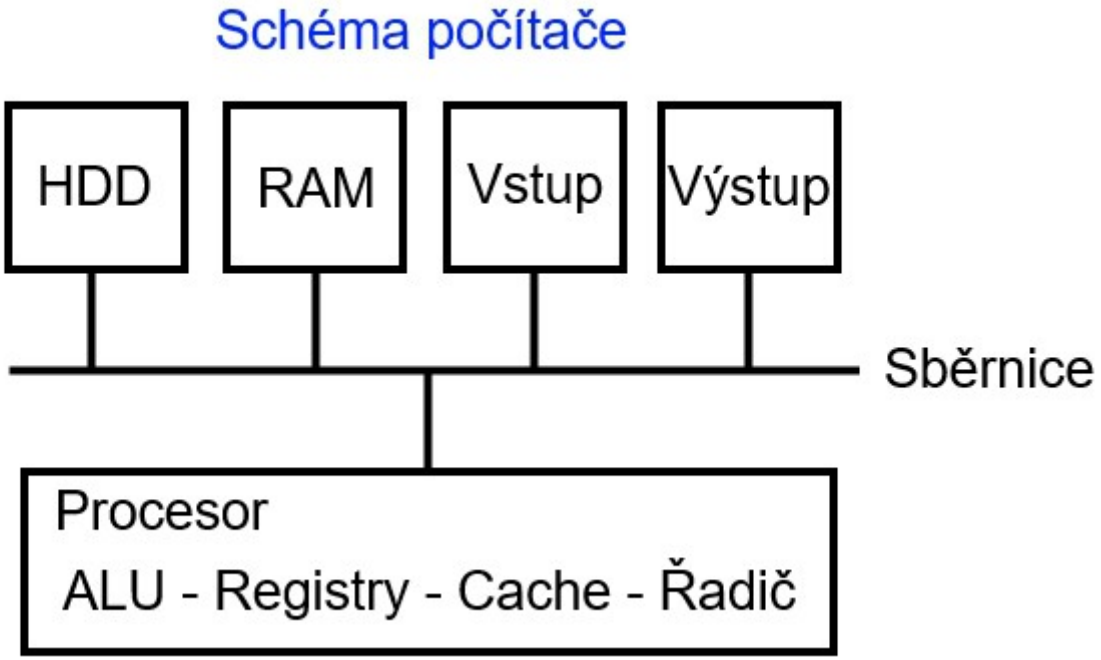


Schéma počítače



Sběrnice (BUS) přenáší data mezi elementy počítače

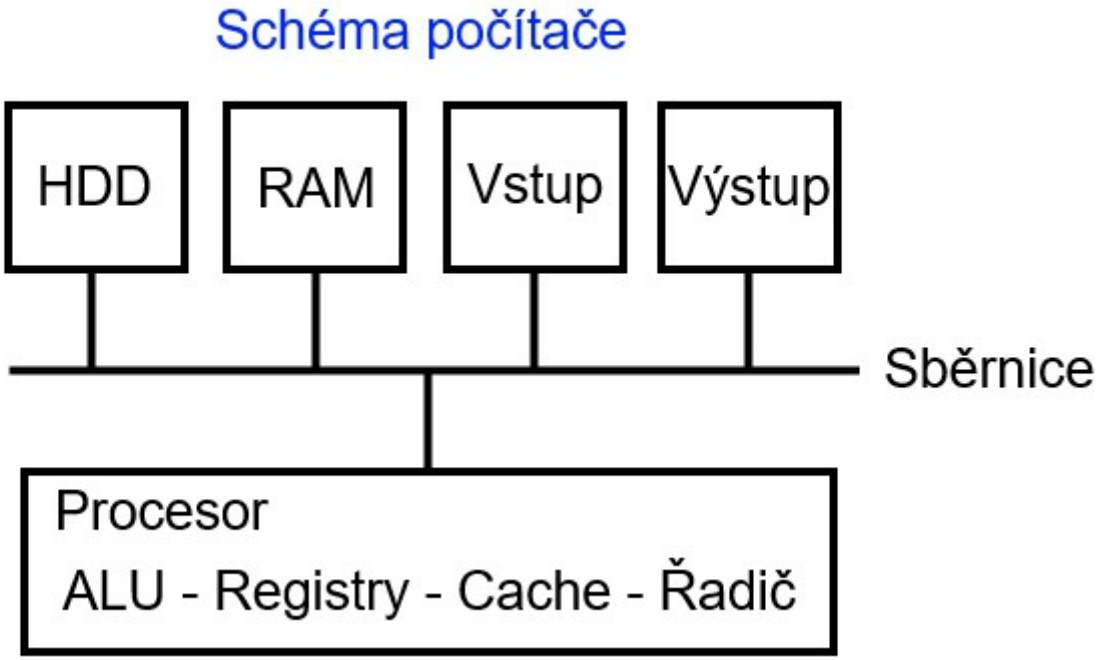
Schéma počítače



Sběrnice (BUS) přenáší data mezi elementy počítače

HDD pevný disk: uložení informací (aplikace, data) – přístupová rychlost asi 8 ms

Schéma počítače

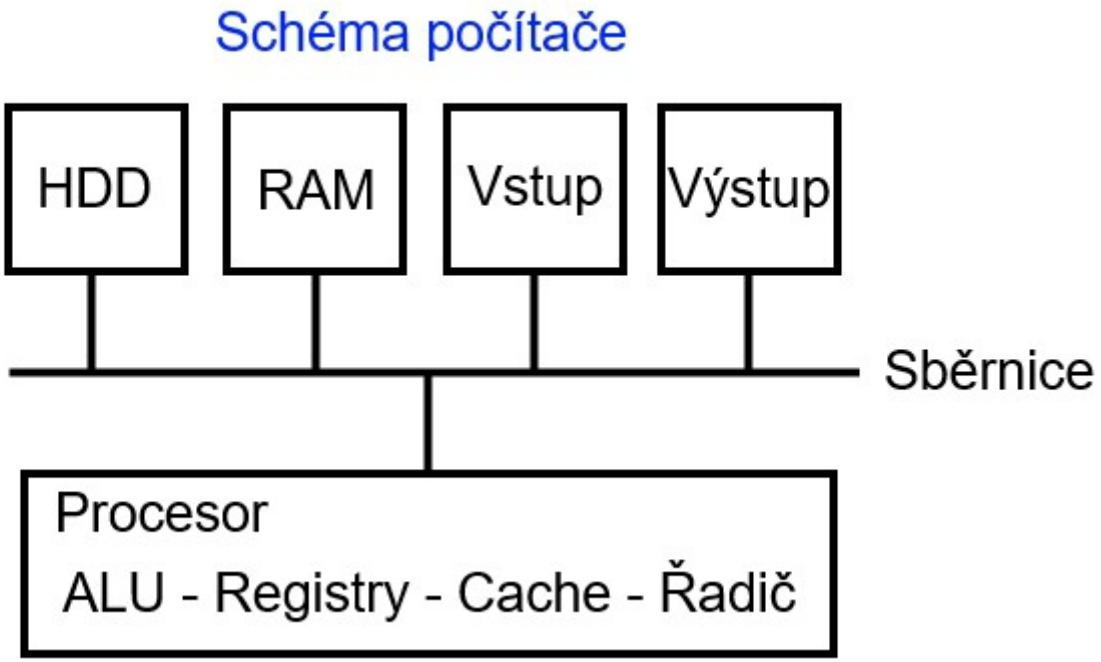


Sběrnice (BUS) přenáší data mezi elementy počítače

HDD pevný disk: uložení informací (aplikace, data) – přístupová rychlost asi 8 ms

RAM operační paměť: uložení informací (aplikace, data) - přístupová rychlost asi 100 ns

Schéma počítače

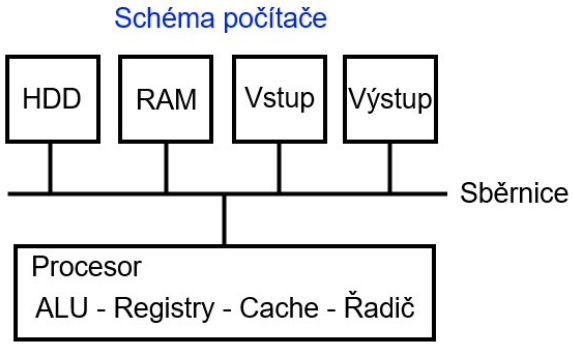


Sběrnice (BUS) přenáší data mezi elementy počítače

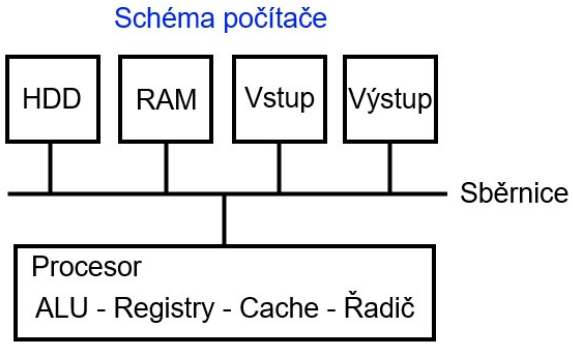
HDD pevný disk: uložení informací (aplikace, data) – přístupová rychlost asi 8 ms

RAM operační paměť: uložení informací (aplikace, data) - přístupová rychlost asi 100 ns

Vstup / Výstup rozhraní - klávesnice, myš, tiskárna

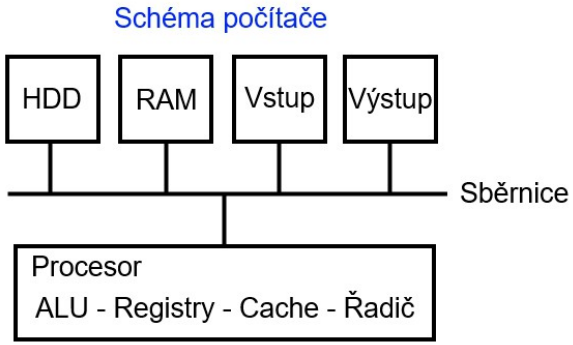


Procesor vykonává instrukce



Procesor vykonává instrukce

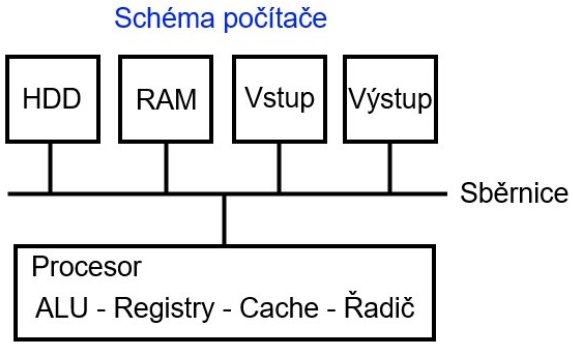
ALU aritmeticko logická jednotka – provádí matematické (+ a -, všechny další matematické operace se provádí pomocí + nebo - a bitových operací) a logické (AND, OR, XOR, NOT, bitový posun) operace



Procesor vykonává instrukce

ALU aritmeticko logická jednotka – provádí matematické (+ a -, všechny další matematické operace se provádí pomocí + nebo - a bitových operací) a logické (AND, OR, XOR, NOT, bitový posun) operace

Cache skrytá paměť: uložení informací (aplikace a odděleně data - dvě cache) - přístupová rychlost asi 10 ns

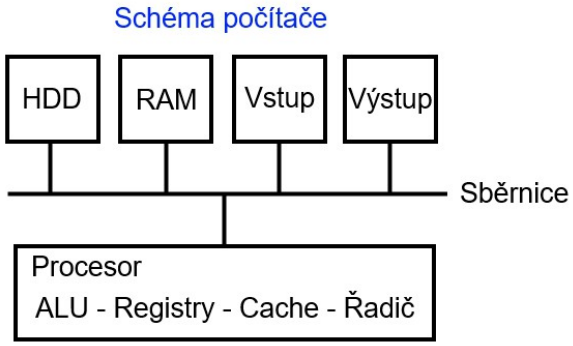


Procesor vykonává instrukce

ALU aritmeticko logická jednotka – provádí matematické (+ a -, všechny další matematické operace se provádí pomocí + nebo - a bitových operací) a logické (AND, OR, XOR, NOT, bitový posun) operace

Cache skrytá paměť: uložení informací (aplikace a odděleně data - dvě cache) - přístupová rychlost asi 10 ns

Registry uložení dat - přístupová rychlost asi 1 ns



Procesor vykonává instrukce

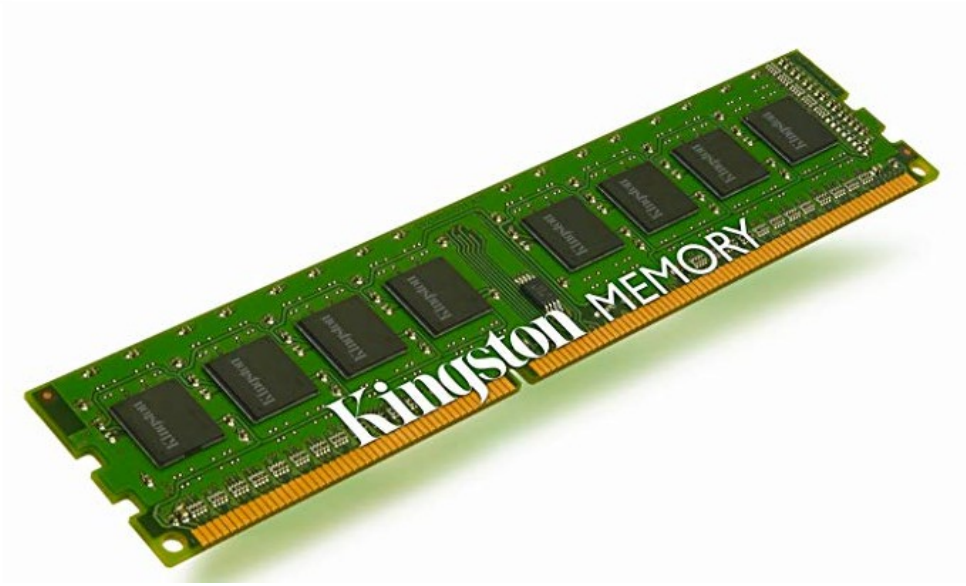
ALU aritmeticko logická jednotka – provádí matematické (+ a -, všechny další matematické operace se provádí pomocí + nebo - a bitových operací) a logické (AND, OR, XOR, NOT, bitový posun) operace

Cache skrytá paměť: uložení informací (aplikace a odděleně data - dvě cache) - přístupová rychlost asi 10 ns

Registry uložení dat - přístupová rychlost asi 1 ns

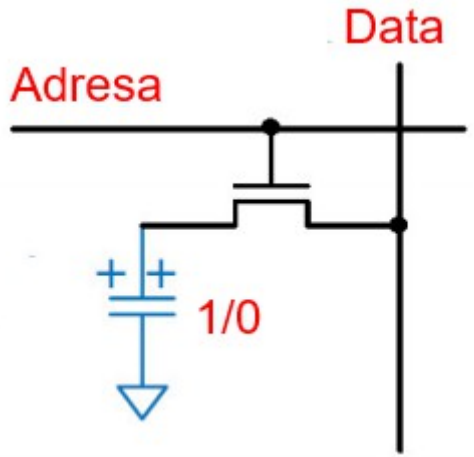
Řadič (Driver) dekóduje binární instrukce a na základě instrukce vydá pokyny (řídí) pro procesor

Operační paměť



DRAM – dynamic random access memory - dynamická paměť s přístupem na libovolné místo

Paměťová buňka (element RAM)



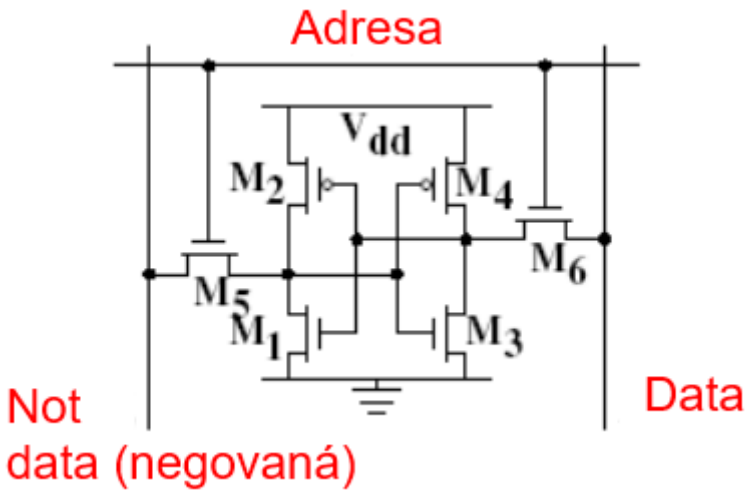
Uložení 1 do paměti: adresa 1, data 1
Uložení 0 do paměti: adresa 1, data 0

Adresa v operačním systému 32 bit je 32 bitů – např. 00A8ff75
1 znak jsou 4 bity – 8 znaků adresy x 4 bity je 32 bitů, to jsou 4 byty
Adresa v operačním systému 64 bit je 64 bitů – např. 00A8ff75abab0011
1 znak jsou 4 bity – 16 znaků adresy x 4 bity je 64 bitů, to je 8 bytů
Adresu paměťové buňky určuje procesor, adresa je definovaná ve vyhrazené paměti aplikace (heap+stack+systémové informace), která pracuje s těmito daty.
Např. 4 GB RAM má 2^35 paměťových buněk (4 GB = 2^32 B = 2^35 b)
(4 GB = 2^2 . 2^30 B = 2^32 B = 2^32 . 2^3 b = 2^35 b)

Pozor:

32 bitový operační systém může adresovat maximálně 16 GB paměti,
protože má 2^32 adres a adresuje se každý čtvrtý byte (základem je integer),
každý čtvrtý proto, že 32 bitová sběrnice může přenášet 32 bitů = 4 byty.
64 bitový operační systém může adresovat maximálně 64 EB (exa=2^60 bytů) paměti,
protože má 2^64 adres a adresuje se také každý čtvrtý byte.

Princip Cache paměti (SRAM – statická paměť)



Negovaná data automaticky umí klopný obvod (flip flop) – může použít, když potřebuje negovaná data.

Protože statická paměť nemá kondenzátory (capacitor), je rychlejší - nepotřebuje čas na změnu stavu (state). Nepoužívá se na RAM, protože je mnohem dražší, než statická.