

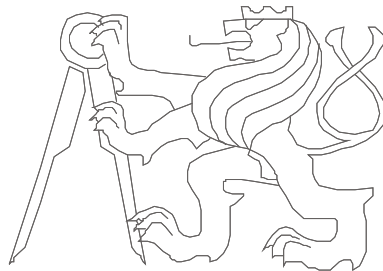
Architektura počítačů

TEST v 9. týdnu!!!

1. I/O podsystém II.

- Paměťově mapované I/O
- PCI

2. Volání funkcí a předávání parametrů – Úvod

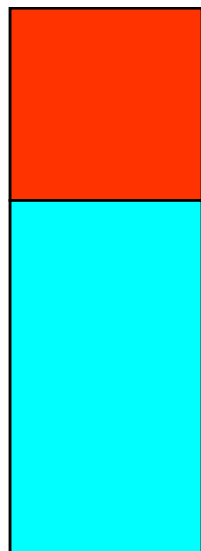


České vysoké učení technické, Fakulta elektrotechnická

Paměťově mapované I/O

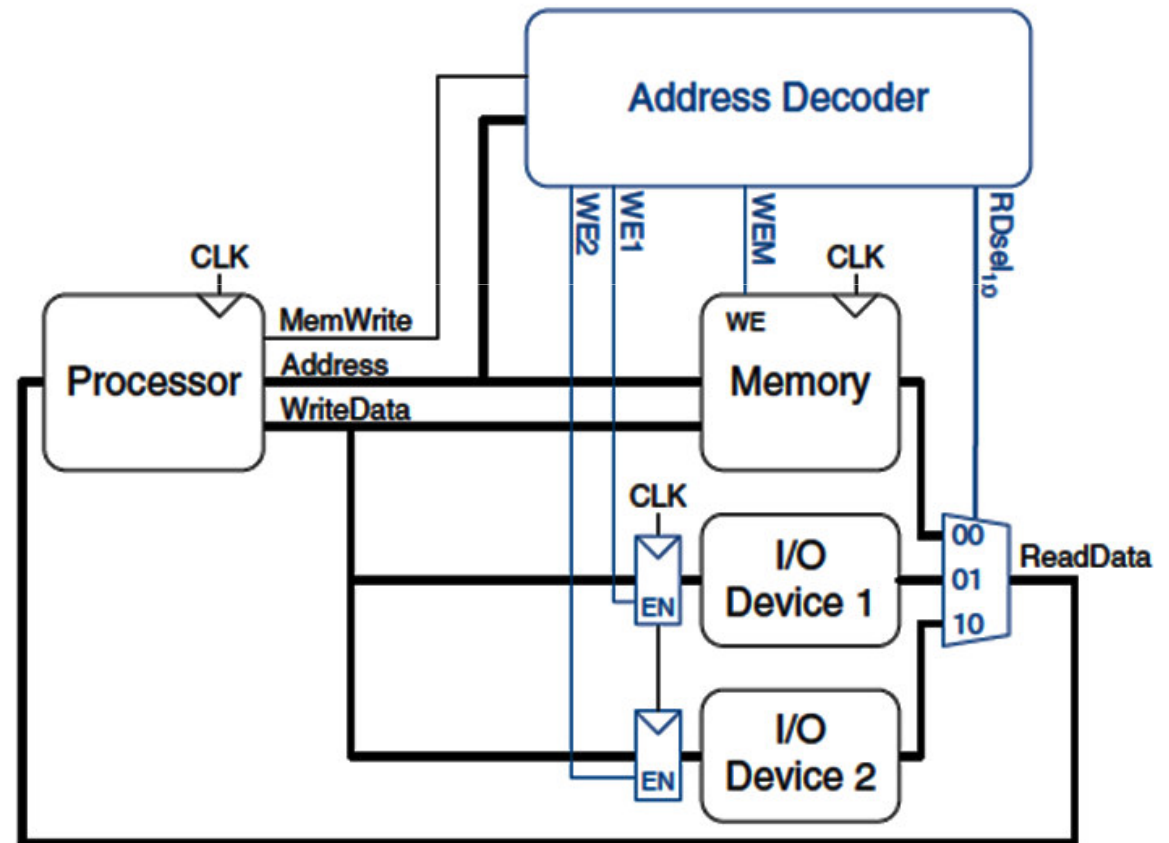
- Idea: K tomu abychom komunikovali s vstupně/výstupními periferiemi (klávesnice, monitor, tiskárna) můžeme použít stejné rozhraní jako pro komunikaci s pamětí (MIPS: instrukce lw, sw).

Společný adresní prostor pro I/O a paměť



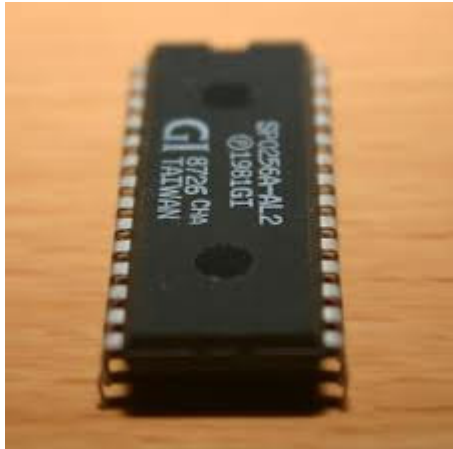
V/V brány jsou mapované do paměti

paměť

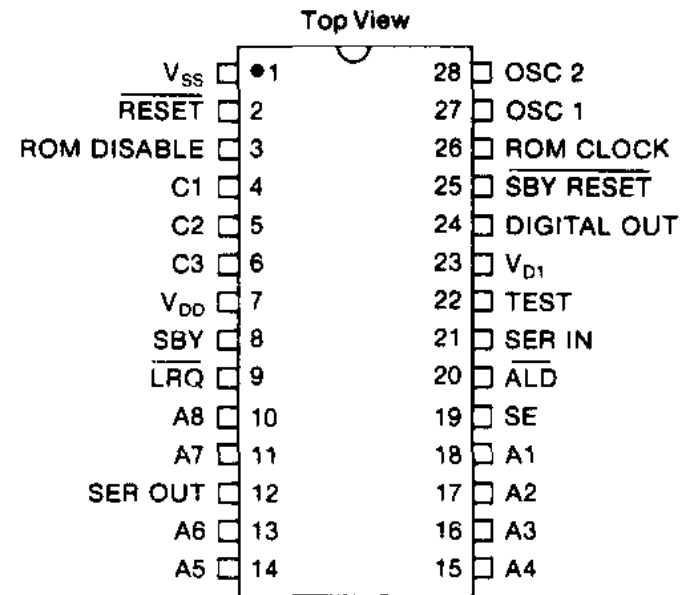


Příklad – Syntetizátor řeči

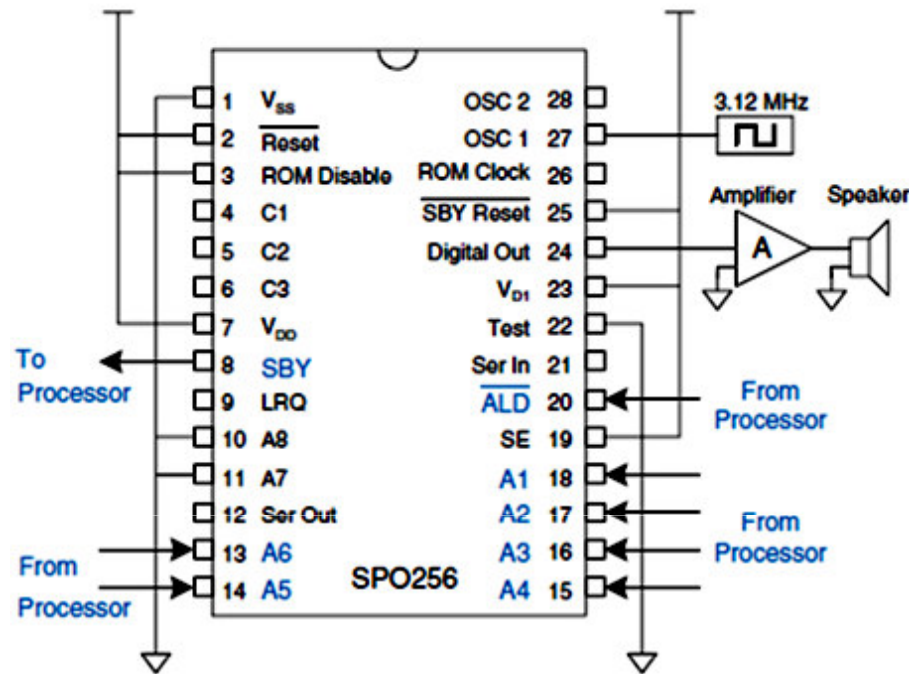
- V angličtině se vyskytuje 64 různých elementárních hlasových „zvuků“ (angl.: allophones), ze kterých se skládají jednotlivá slova. Čip SP0256 je schopen generovat tyto jednotlivé zvuky.
- **Úkol: Napišme ovladač pro tento čip.**
- Zjednodušující předpoklady: Ovladač bude číst 5 položek (allophonů) od adresy 0x10000000 a ty pak pošle jednu za druhou do čipu SP0256.



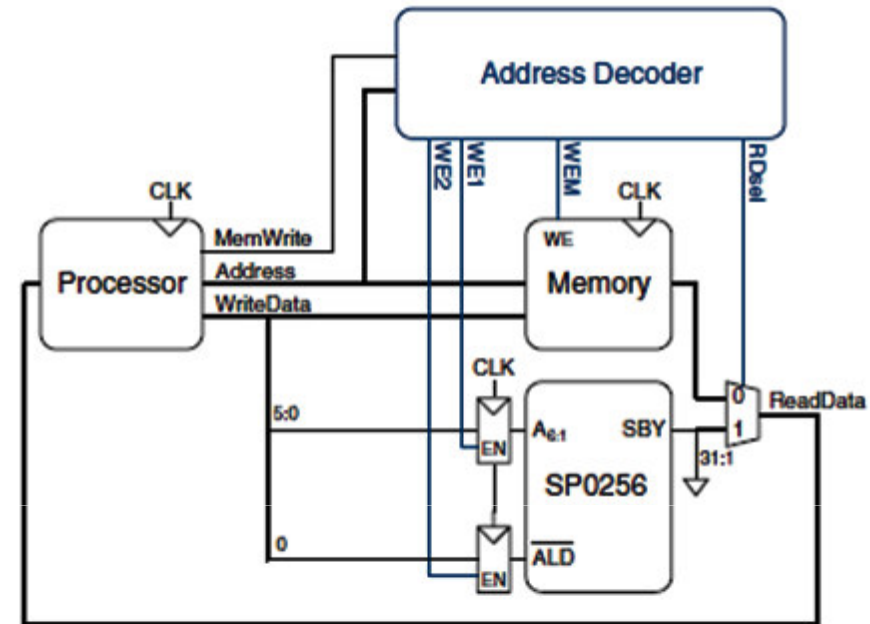
<http://little-scale.blogspot.cz/2009/02/sp0256-al2-creative-commons-sample-pack.html>



Příklad – Syntetizátor řeči



SPO256 speech synthesizer chip pinout



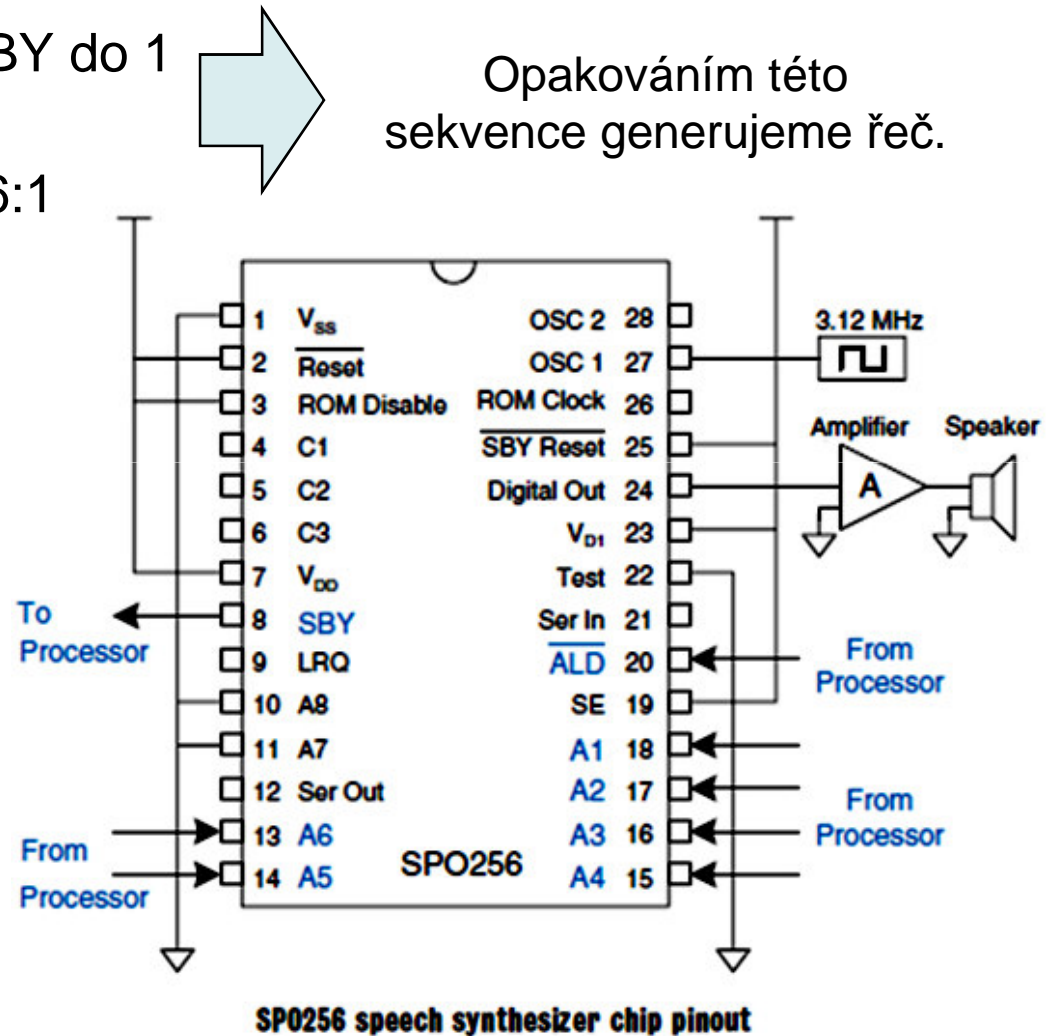
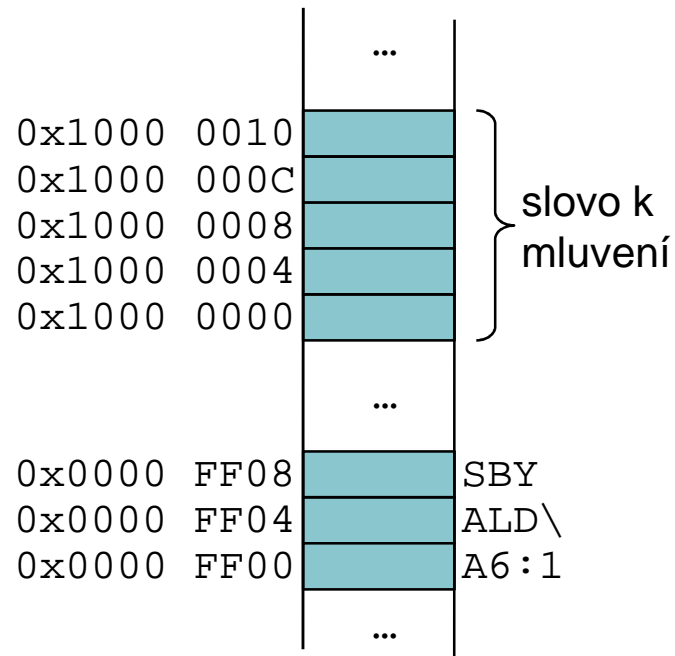
Hardware for driving the SPO256 speech synthesizer

- Pokud je výstup **SBY** roven 1, pak zařízení čeká na příjem dalšího allophonu. Na sestupné hraně vstupu **ALD** pak přečte allophon (jeho hodnota je na vstupech **A6:1**).
- Zvolíme si mapování: Port **A6:1** bude namapován do adresy **0xFF00**, **ALD** do adresy **0xFF04**, a **SBY** do **0xFF08**.

Příklad – Syntetizátor řeči – Ovladač

1. Nastav *ALD* na 1
2. Čekej dokud čip nenastaví *SBY* do 1 (indikace, že je připraven)
3. Zapiš 6-bitový allophon do *A6:1*
4. Nastav *ALD* na 0 (začni generovat zvuk)

Opakováním této sekvence generujeme řeč.



Příklad – Syntetizátor řeči – Ovladač

1. Nastav *ALD* na 1
2. Čekej dokud čip nenastaví *SBY* do 1 (indikace, že je připraven)
3. Zapiš 6-bitový allophon do *A6:1*
4. Nastav *ALD* na 0 (začni generovat zvuk)



```

init:
    addi t1,$0,1      // t1 = 1 (do ALD#)
    addi t2,$0,20     // t2 = velikost pole (20 B)
    lui  t3,0x1000    // t3 = adresa zacatku pole
    addi t4,$0,0      // t4 = 0 (index do pole)
start:
    sw t1,0xFF04($0) // ALD#=1
loop:
    lw t5,0xFF08($0) // t5 = SBY (zjistí stav)
    beq $0,t5,loop   // opakuj dokud SBY == 1
    add t5,t3,t4     // t5 = adresa allophonu
    lw t5,0(t5)      // t5 = allophon
    sw t5,0xFF00($0) // A6:1 = allophone
    sw $0,0xFF04($0) // ALD# = 0 (zacni mluvit)
    addi t4,t4,4     // posun na dalsi allophon
    beq t4,t2,done   // jsme na konci?
    j start          // opakuj
done:
    
```

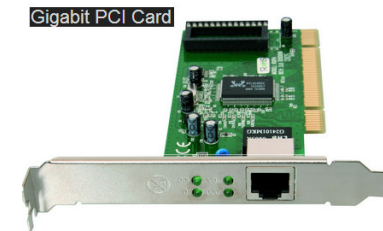
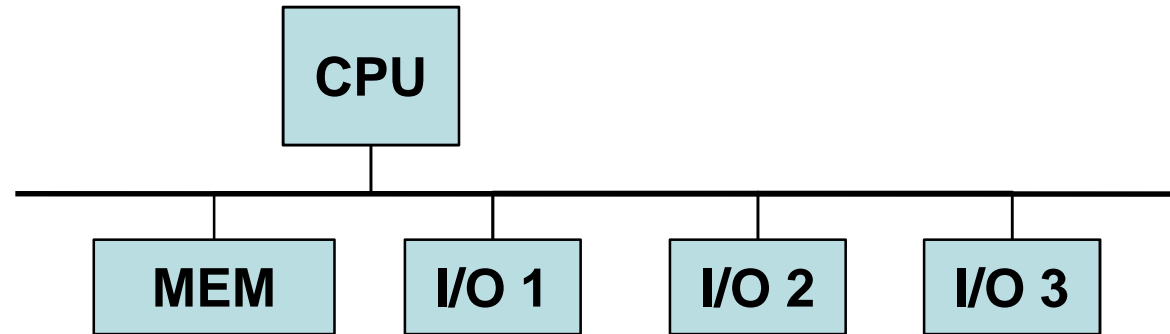
Všimněte si, že procesor sleduje, zda je zařízení připraveno (sleduje hodnotu *SBY*). Tomu se říká **polling**. Bylo by však lepší nastavit na *SBY* **přerušeni** a nechat procesor dělat i jinou činnost.

Závěry z příkladu č.1 - Obecněji:

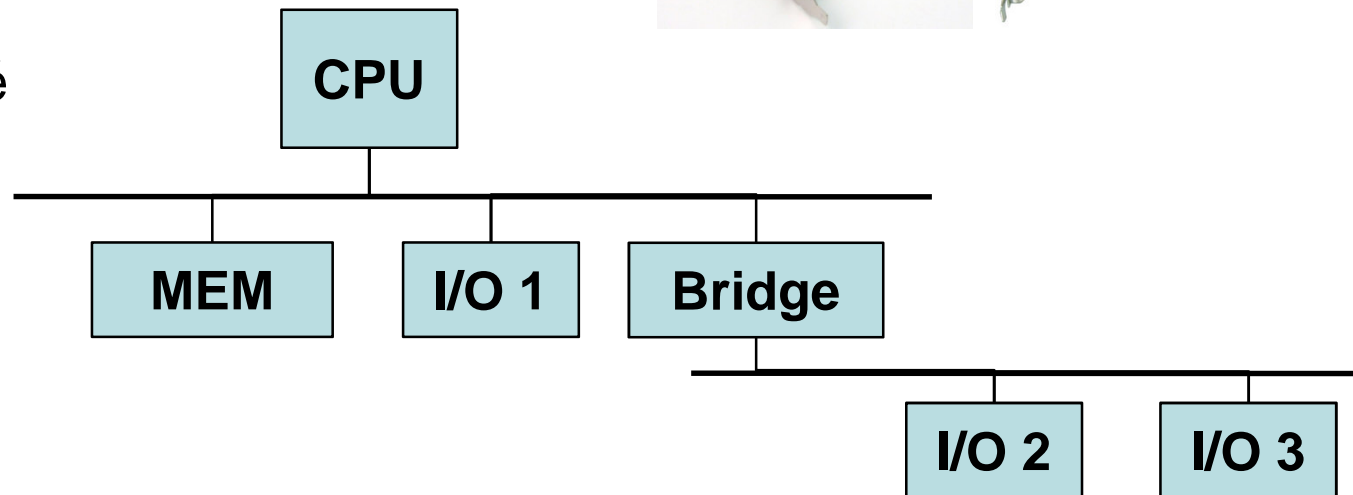
- Existují dvě metody přístupu k I/O zařízením:
 - paměťově mapované I/O
 - speciální I/O instrukce
- Pro paměťově mapované I/O jsou části adresního prostoru vyhrazeny (přirazeny) pro vstupně/výstupní zařízení. Čtení/záписy z/do těchto adres jsou interpretovány jako příkazy nebo přenosy dat z/do těchto zařízení. Paměťový systém ignoruje tyto operace (zná rozsah adres použitý pro I/O). Řadič I/O zařízení nicméně vidí tyto operace a reaguje na ně (ví které adresy jsou mu přirazeny).
- Upozornit procesor na to, že I/O vyžaduje „pozornost“ můžeme:
 - sledováním jeho stavu (stavový registr) – **polling (software-driven I/O)** – neustálé sledování (busy waiting – to byl náš případ) vs. periodické (občasné) sledování - procesor pak ale reaguje s prodlevou
 - použitím mechanismu přerušení – **interrupt-driven I/O** – bude asynchronní (může se objevit kdykoliv)
- **Všimli jste si jakou funkci plní dekodér adres ???**

Dekodér adres - idea

- Logická struktura: (iluze)

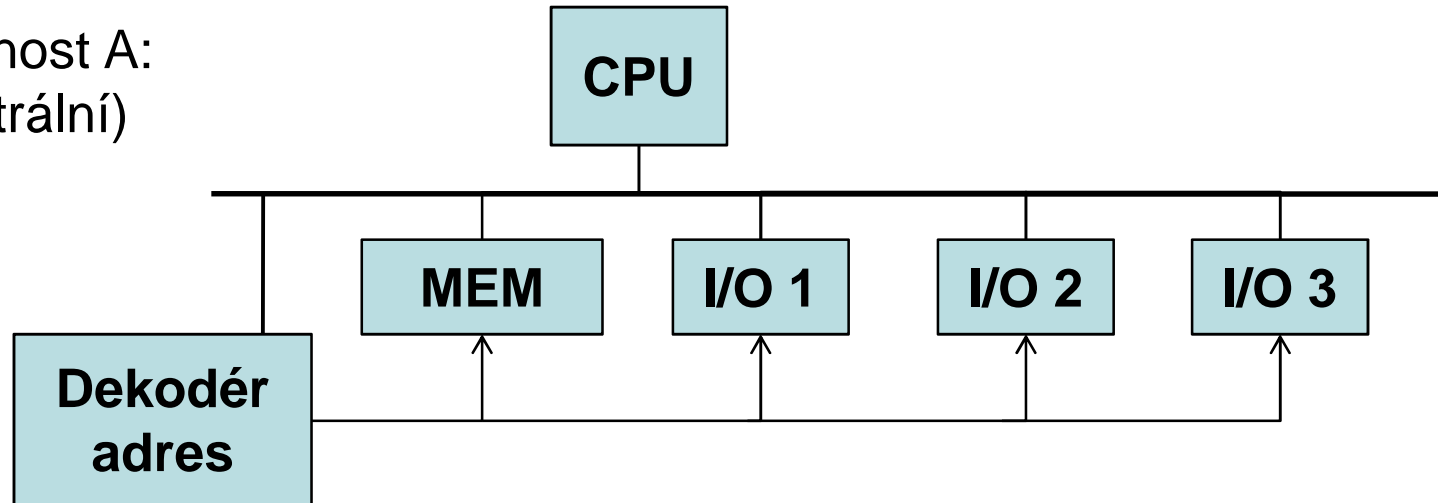


- Možné fyzické uspořádání:

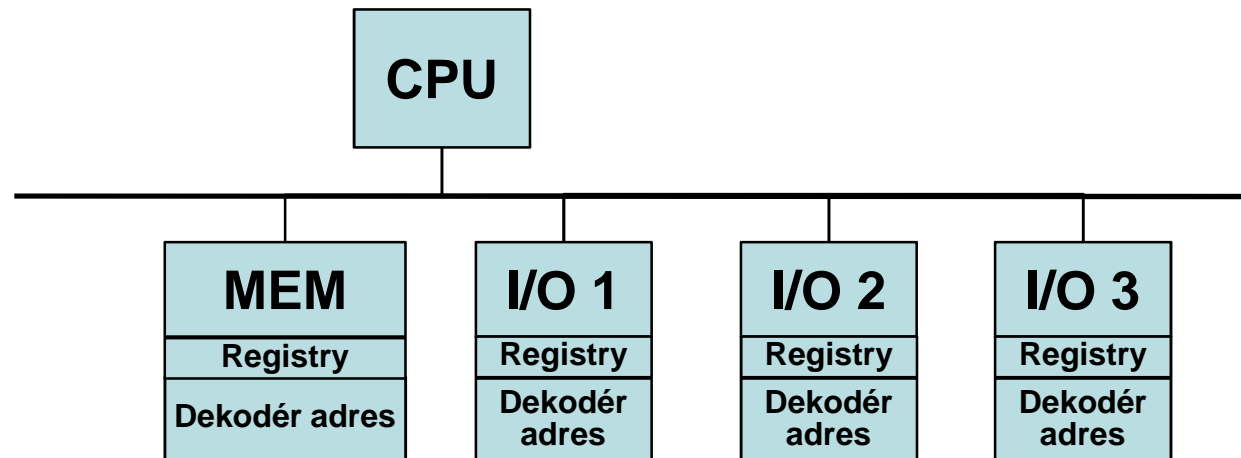


Dekodér adres - idea

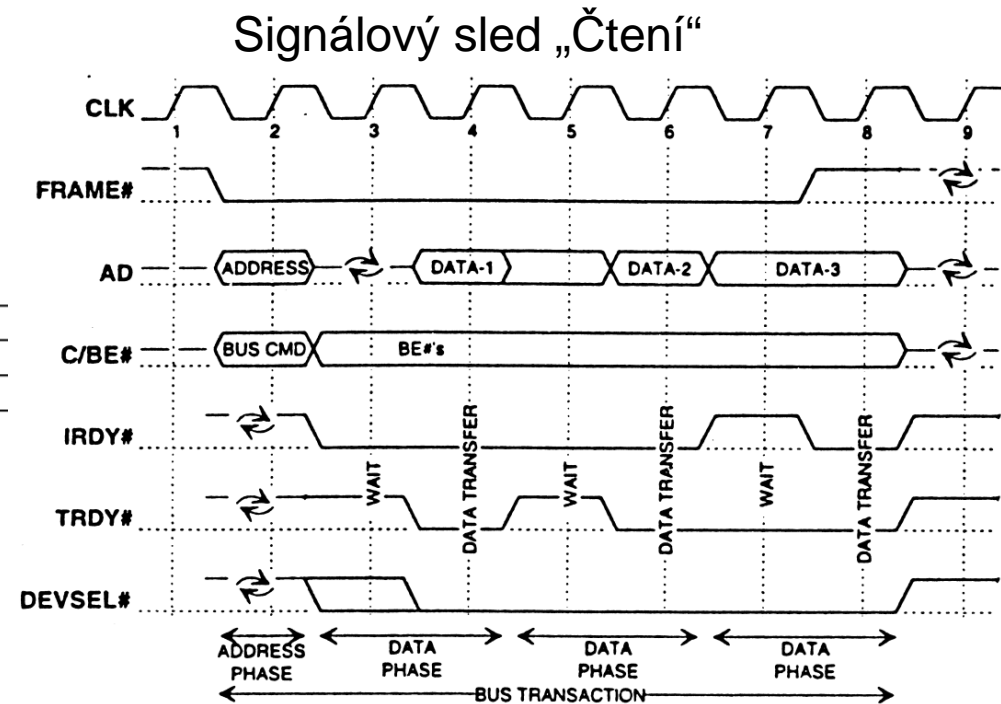
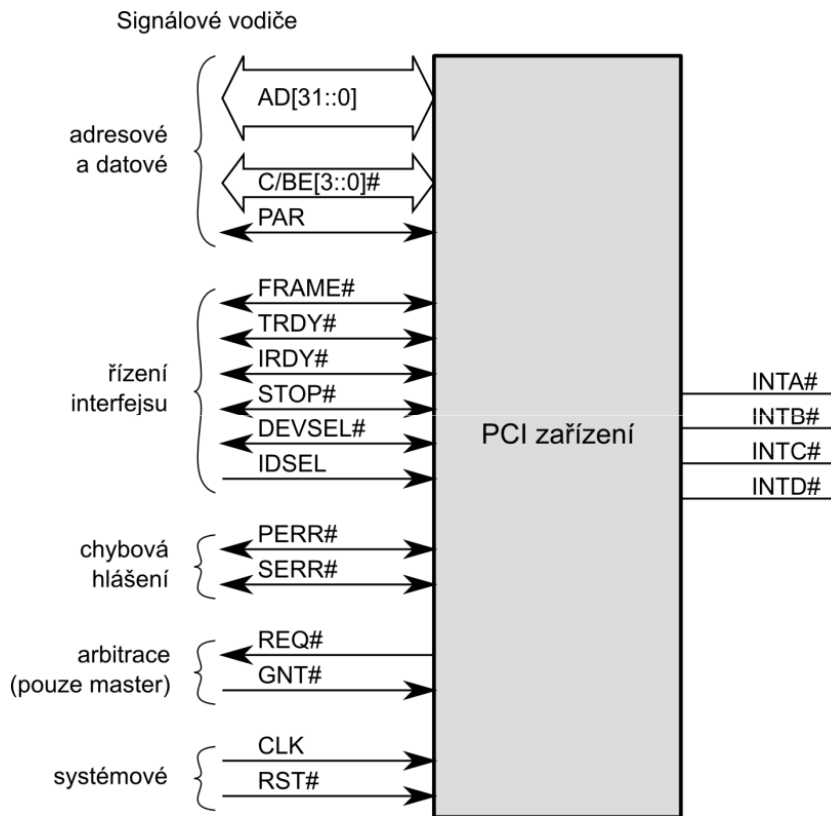
- Možnost A:
(centrální)



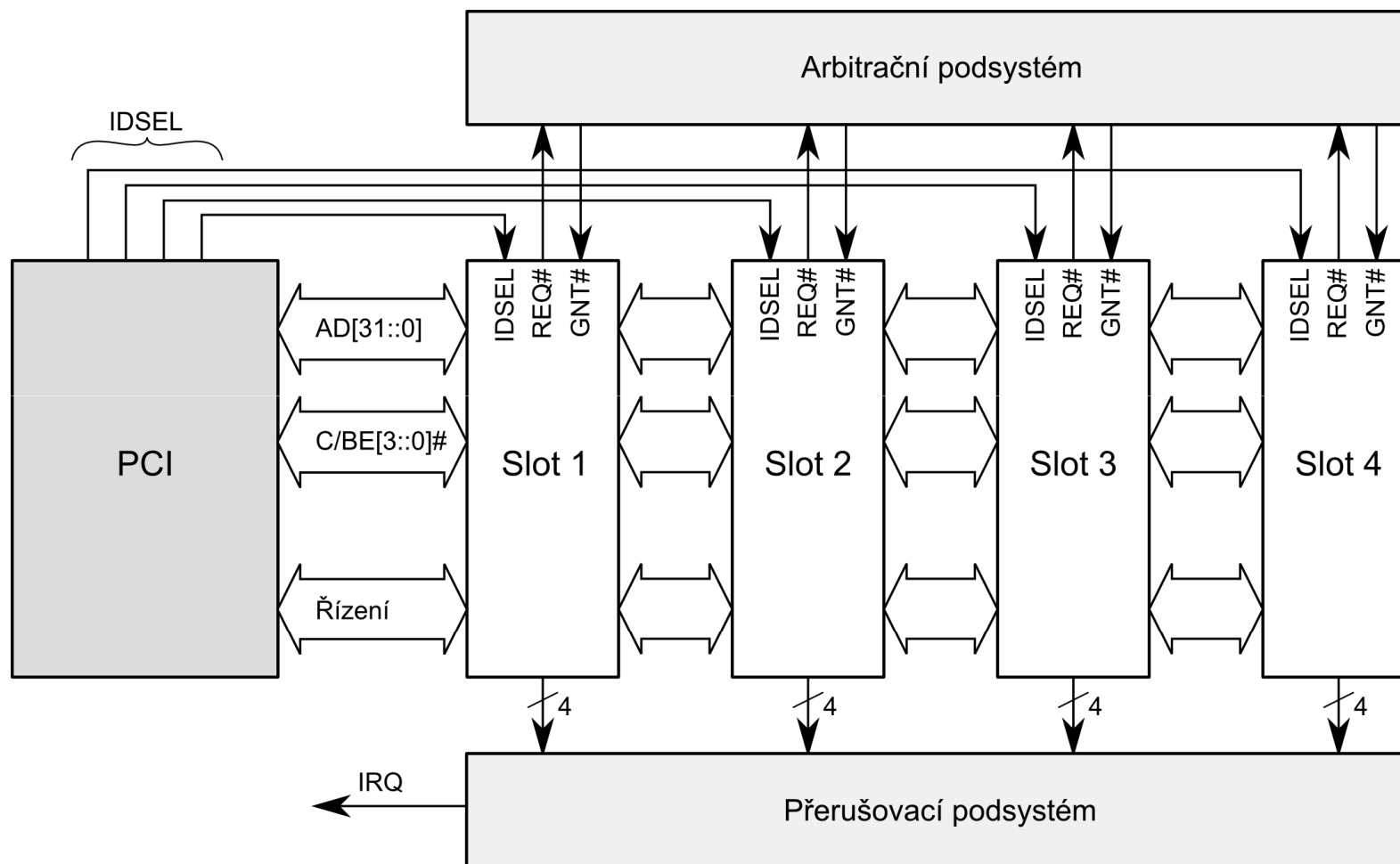
- Možnost B:
(autonomní)

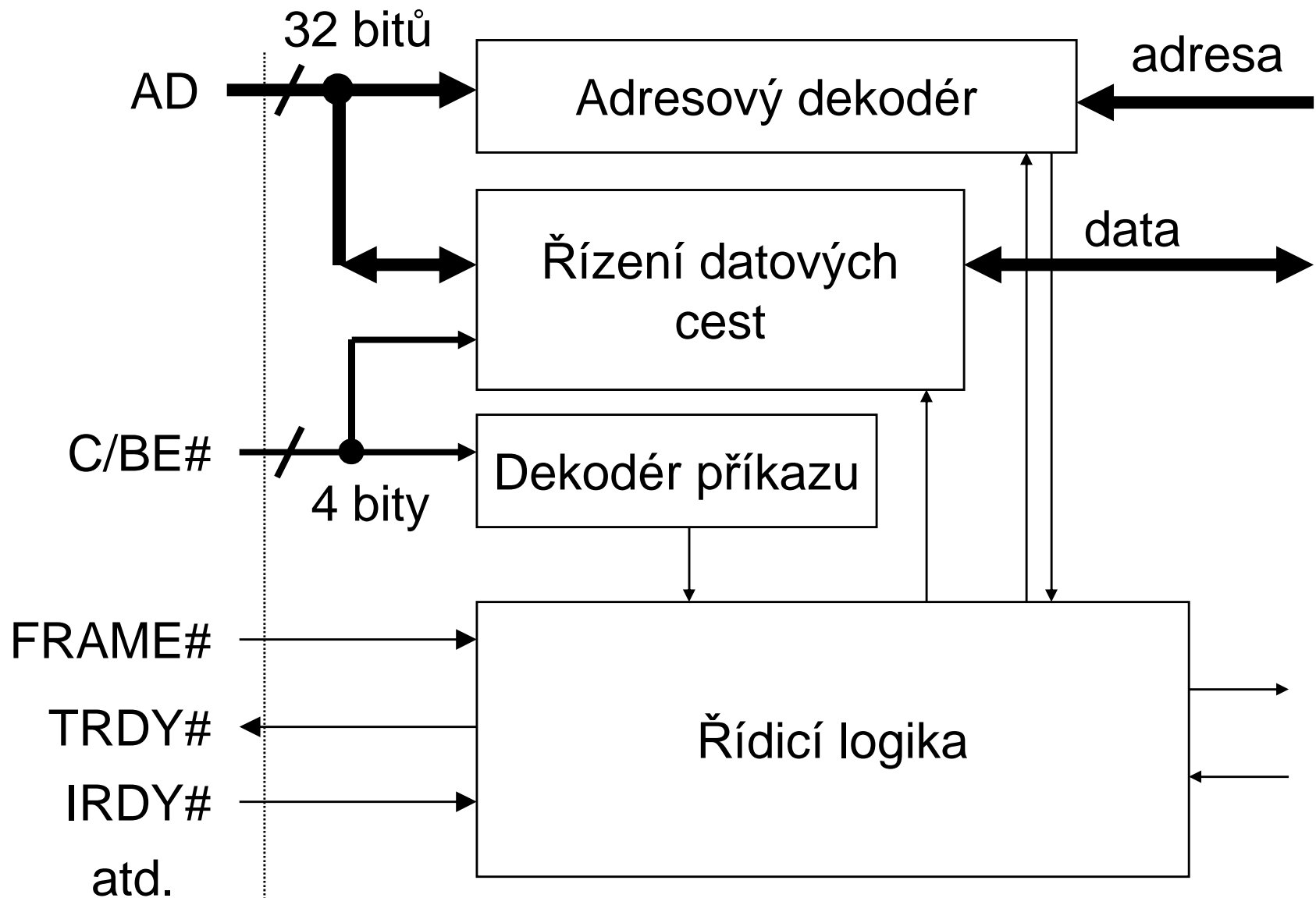


Připomeňme si z minulé přednášky...



Připomeňme si z minulé přednášky...

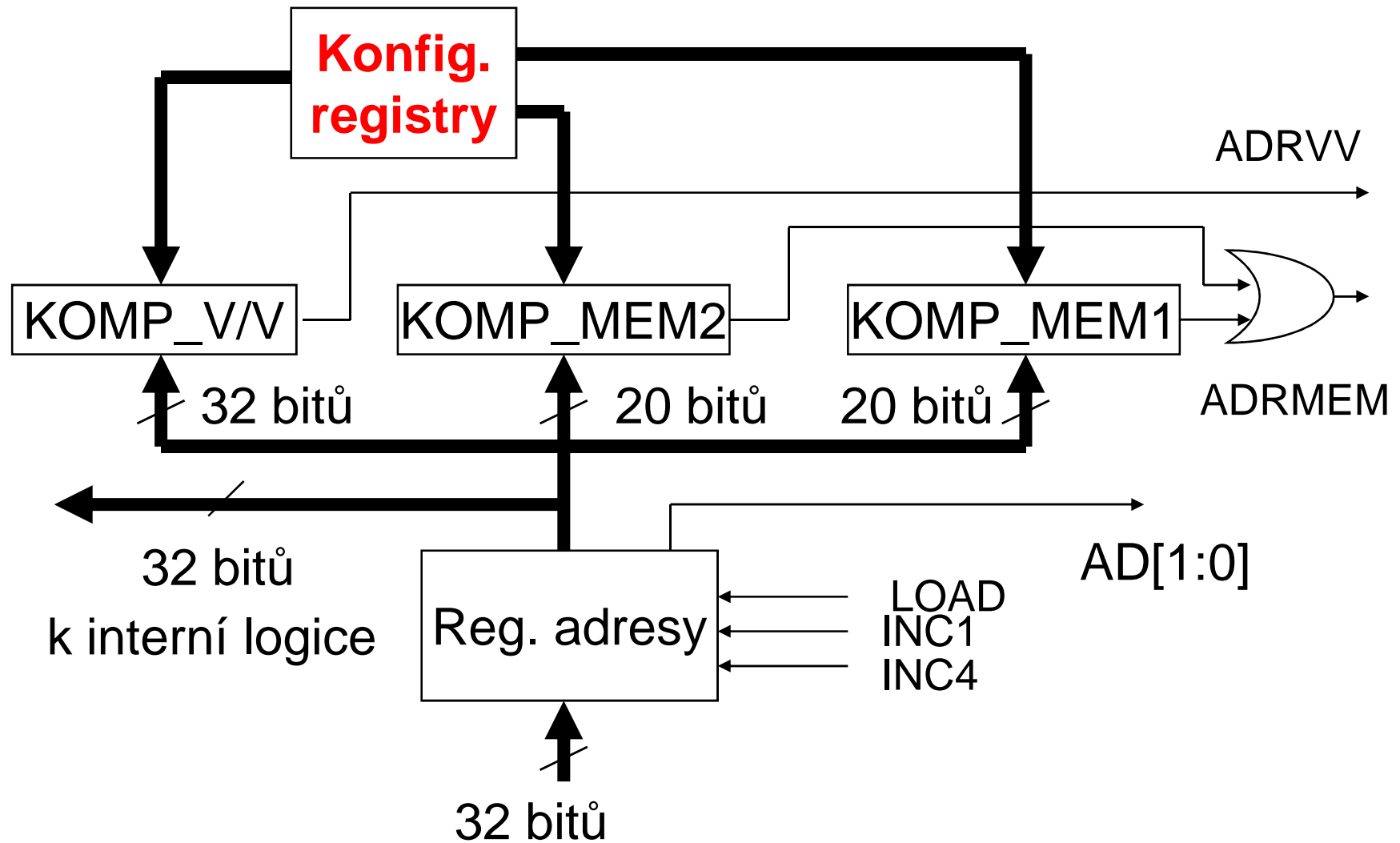




Adresový dekodér

- Základem je komparátor adres:
 - porovnává adresu na adresové sběrnici a adresu v některém bázovém registru konfiguračního prostoru (pro každý registr jeden komparátor).
- Adresa je na AD sběrnici pouze v první fázi sběrnicevého cyklu \Rightarrow adresa se musí zapamatovat v *registru adresy*.
- Blokovaný přenos dat s autoinkrementací \Rightarrow registr adresy bude mít podobu *čítače s paralelním přednastavením (LOAD)*.
- Relokovatelný adresový dekodér? Zrcadlení?

Adresový dekodér

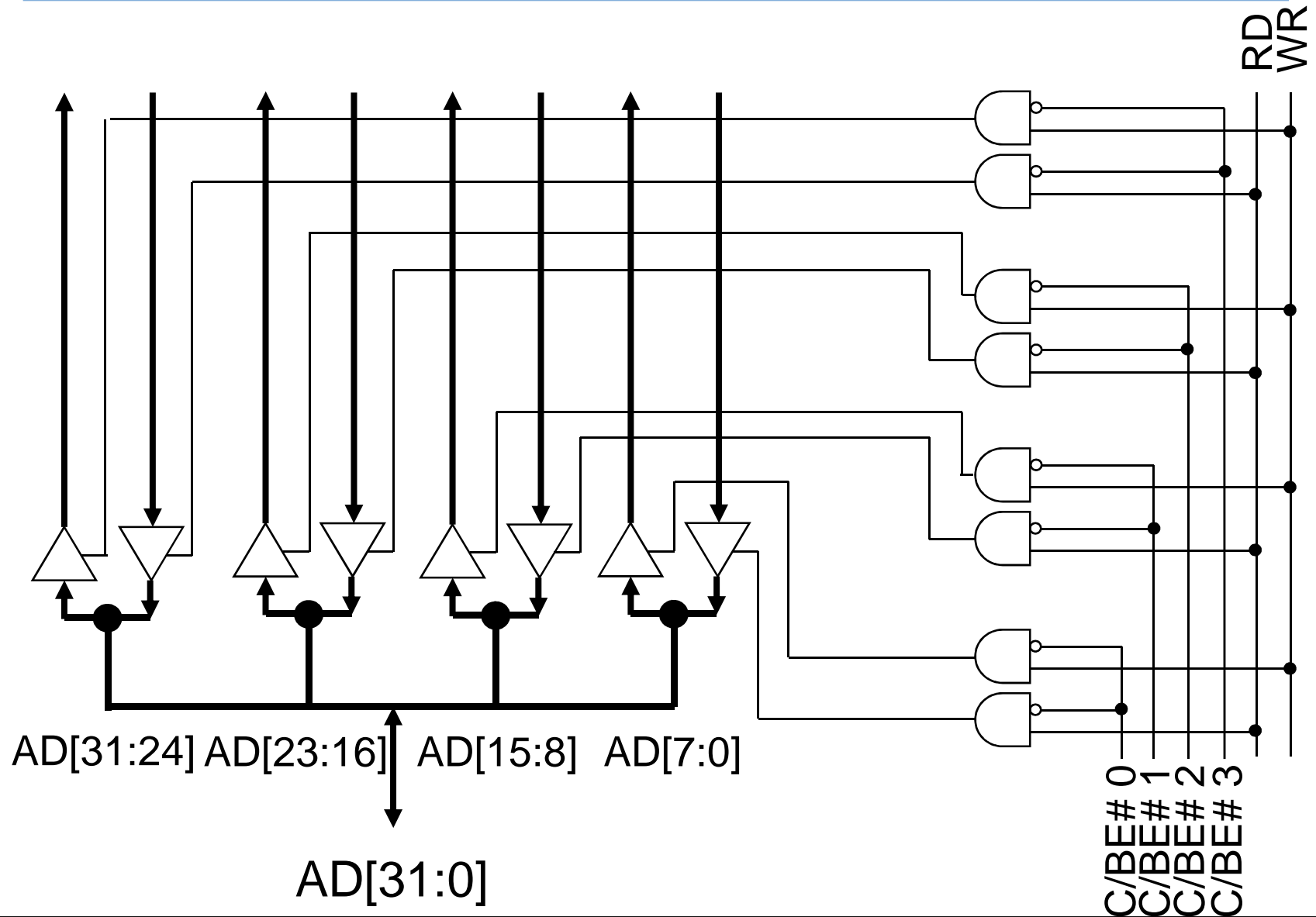


- **ADRVV**
 - rovnost adresy V/V prostoru.
- **ADRMEM**
 - rovnost některé adresy paměťového prostoru.
- Registr adresy je synchronní čítač s paralelním přednastavením (synchronním),
 - LOAD - synchronní zápis adresy,
 - INC1 - zvýšení hodnoty o 1,
 - INC4 - zvýšení hodnoty o 4.
- AD[1:0] - k interní logice - informuje o typu burst módu.

Datové cesty a jejich řízení

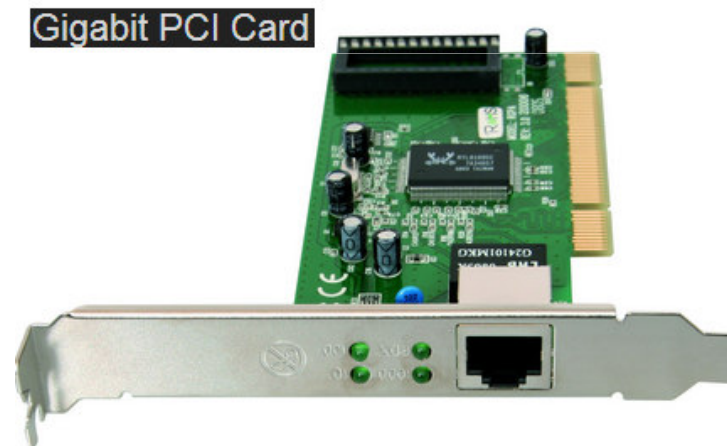
- Obousměrná datová sběrnice \Rightarrow rozhraní budou tvořit **obousměrné třístavové budiče**.
- Datové přenosy 8, 16, 32 bitů \Rightarrow řízení směru a třetího stavu bude odvozeno od typu příkazu (čtení/zápis) a od **masky platnosti dat C/BE#** (po osmi bitech).

Datové cesty



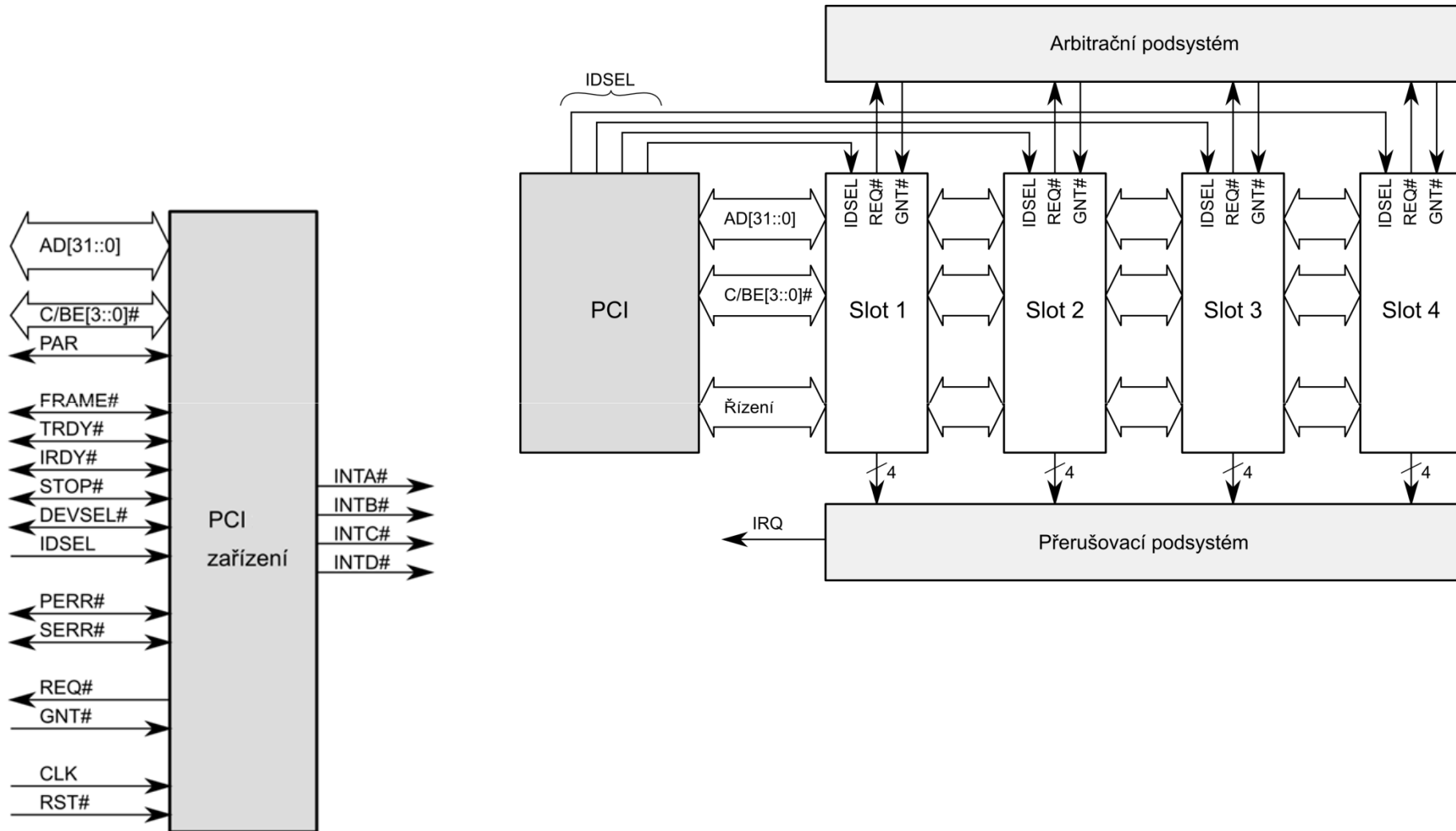
Závěr

- Každé I/O zařízení na PCI sběrnici obsahuje své vlastní speciální registry, ve kterých je uložena adresa (adresy), na které má toto zařízení reagovat...



PCI - pokračování

Připomeňme si z minulé přednášky...

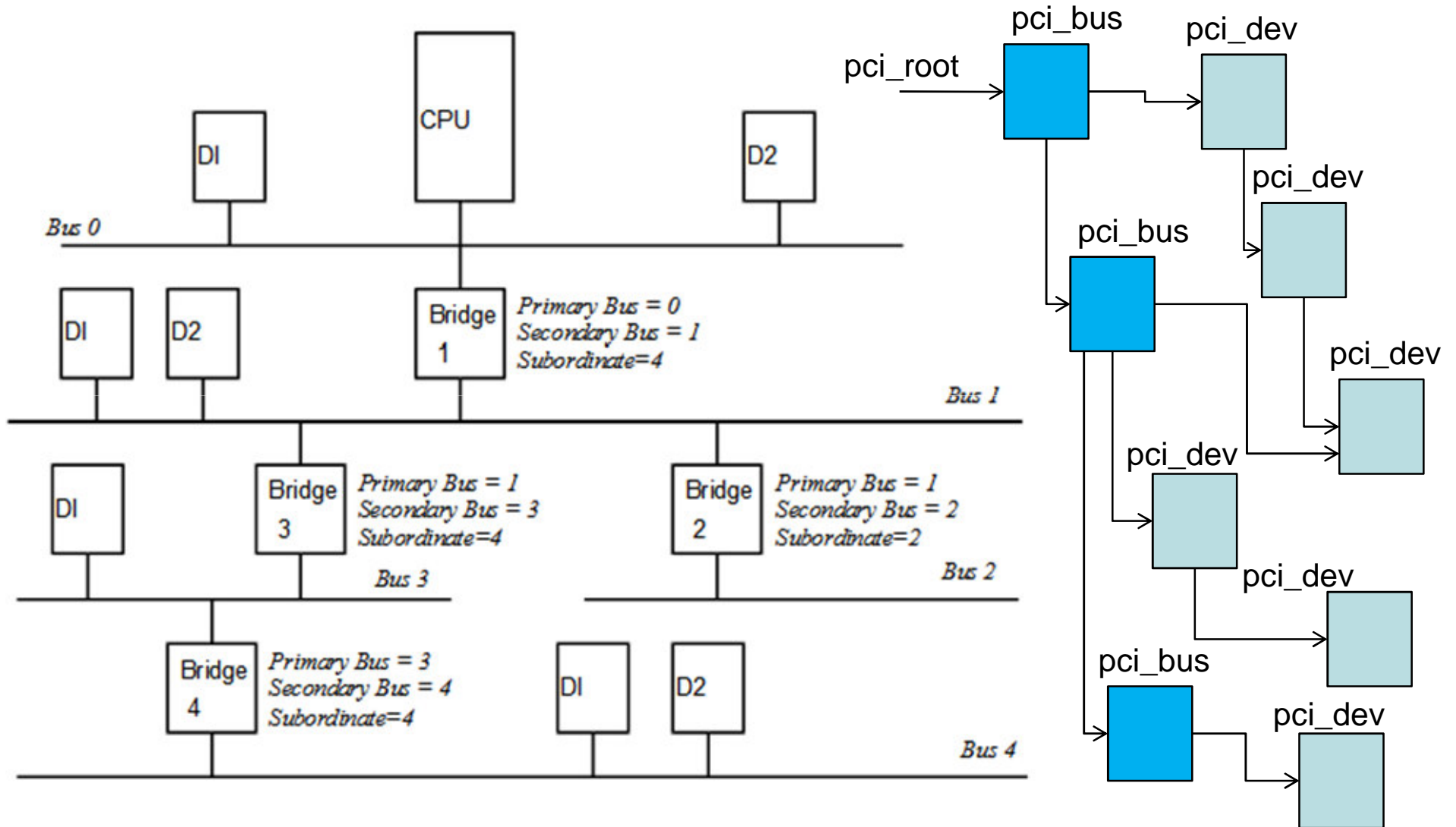


Co se děje po spuštění počítače (z pohledu PCI)?

1. Po spuštění počítače se BIOS postupně dotazuje na připojená zařízení ke sběrnici vysíláním IDSEL (Initialization Device Select) signálu ke každému PCI slotu (bus/device/function)
2. Z hlaviček jednotlivých zařízení přečte jejich identifikátory (Vendor ID, Device ID) a požadavky (na vyhrazení adresního prostoru – pro paměť na kartě, I/O prostor, přerušení) – to všechno je uloženo v konfiguračním prostoru PCI karty
3. BIOS pak přidělí prostředky jednotlivým zařízením tak, aby nedocházelo ke konfliktům – tím, že modifikuje tzv. BAR konfigurační registry karty (BAR = *Base Address Register*) – zůstanou platné až do vypnutí počítače (OS může i modifikovat)
4. spustí se OS; údaje o konfiguraci a adresách všech zařízení si OS vyčte z konfiguračního prostoru, podle identifikace zařízení vyhledá ovladače pro danou kartu

Pozn.: Proces vyhledávání zařízení se označuje jako enumerace (PCI sběrnice, USB sběrnice, atd.)

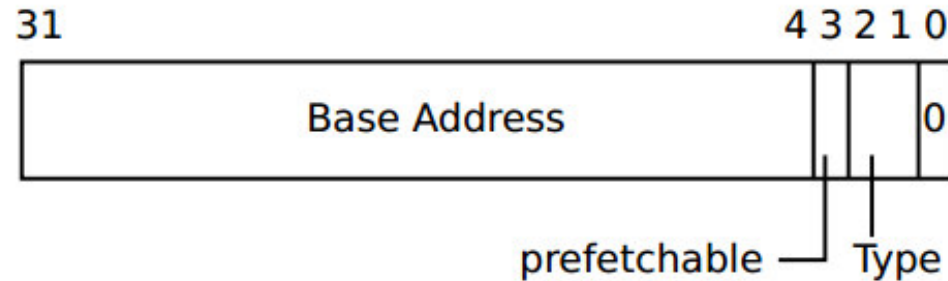
Hierarchie PCI sběrnic



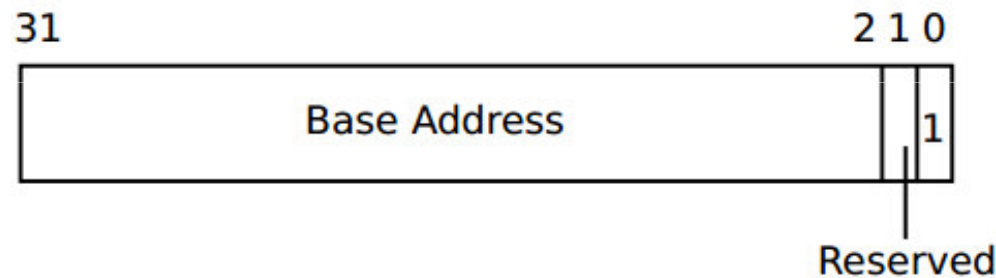
Adresní prostor

- PCI má tři adresní prostory: **paměťový, I/O a konfigurační.**
- Adresy paměti jsou 32-bitové (64-bitové).
- Adresy I/O existují z důvodu kompatibility s adresním prostorem I/O portů architektury x86.
- PCI konfigurační prostor (256 Bajtů) poskytuje přístup do konfiguračních registrů daného PCI zařízení. Prvních 64 Bajtů tvoří hlavičku zařízení (standardní formát), význam zbylého prostoru závisí od výrobce.
- Každé koncové PCI zařízení má k dispozici 6 registrů bázové adresy - Base Address Registers (BARs), kterými může určit celkem 6 nezávislých oblastí (rozsahů adres) do adresního prostoru I/O portů, nebo paměťově mapovaného adresního prostoru. V BAR registru je uložena bázová adresa (adresa začátku oblasti).

Hlavička PCI Zařízení

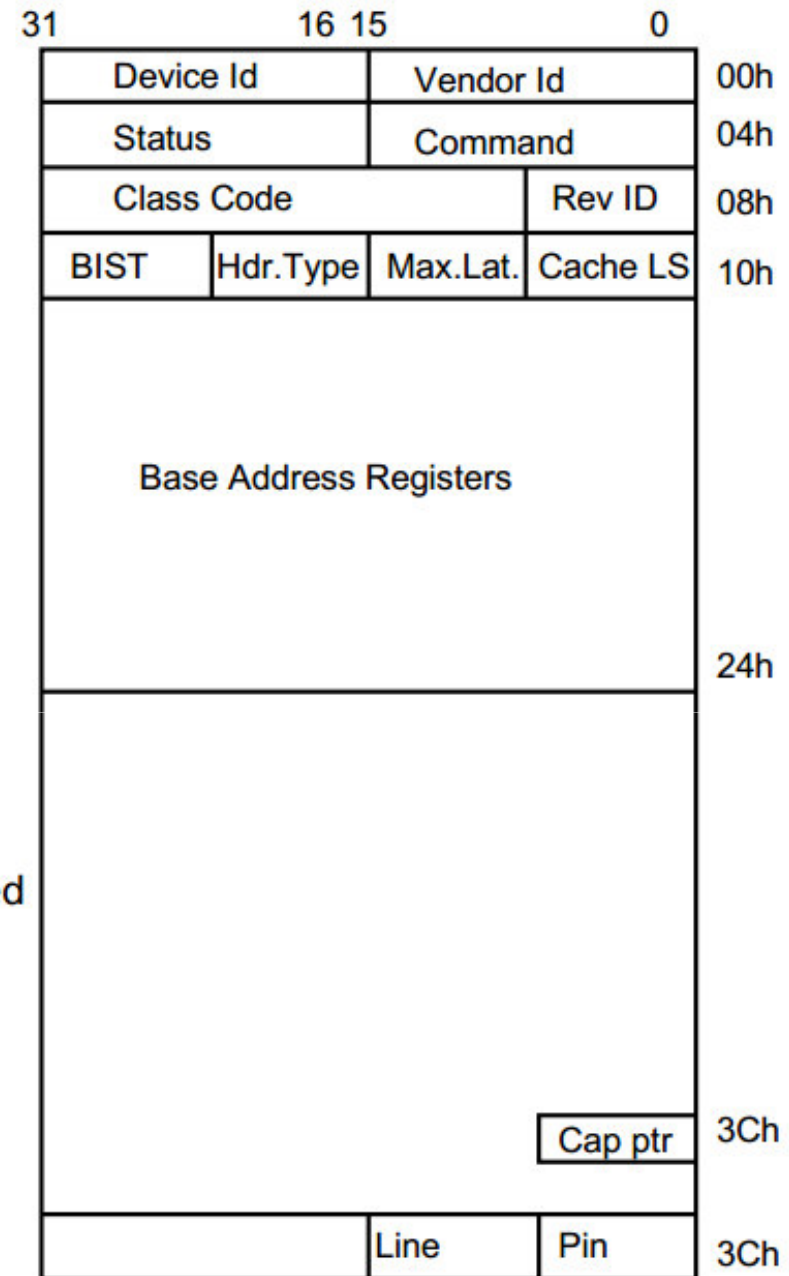


Base Address for PCI Memory Space



Base Address for PCI I/O Space

Hlavička PCI zařízení se nachází v řídicím adresním prostoru PCI sběrnice

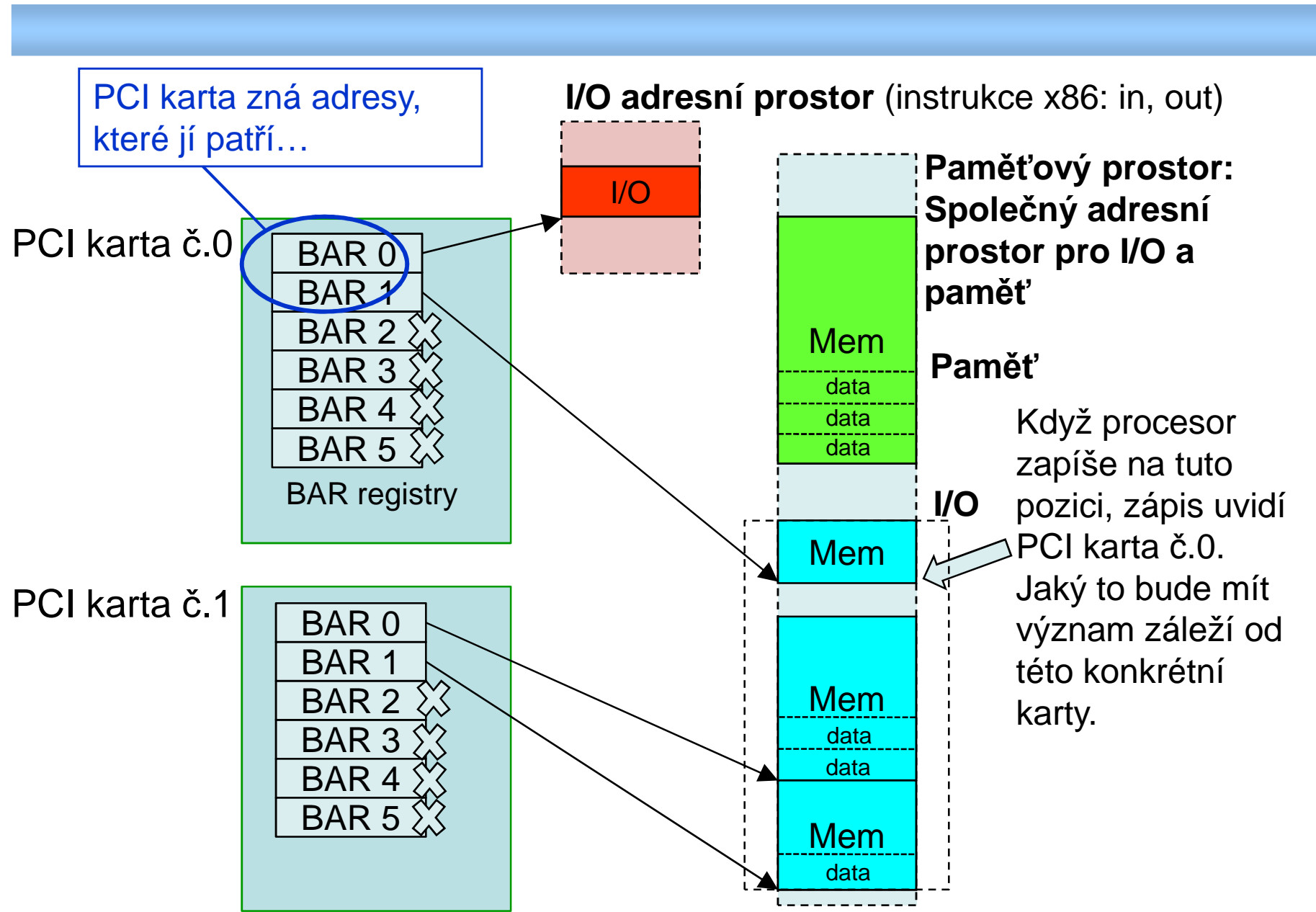


PCI Device Header Type 0 – koncová zařízení

				Byte Offset
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Master Lat. Timer	Cache Line Size	0Ch
Base Address Registers ⁶ max				10h
				14h
				18h
				1Ch
				20h
				24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem vendor ID		2Ch
Expansion ROM Base Address				30h
Reserved			Capabilities Pointer	34h
Reserved				38h
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line	3Ch

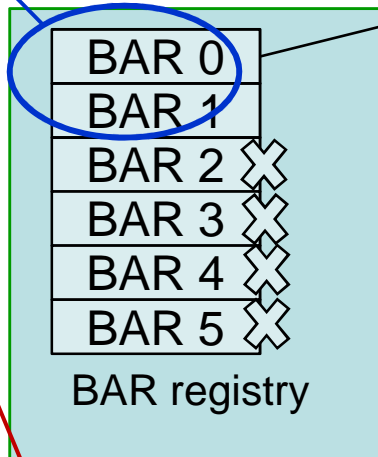
PCI Device Header Type 1 – můstky (bridges)

				Byte Offset
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Master Lat. Timer	Cache Line Size	0Ch
Base Address Register 0				10h
Base Address Register 1				14h
Secondary Latency Timer	Subordinate Bus Number	Secondary Bus Number	Primary Bus Number	18h
Secondary Status		I/O Limit	I/O Base	1Ch
Memory Limit		Memory Base		20h
Prefetchable Memory Limit		Prefetchable Memory Base		24h
Prefetchable Base Upper 32 Bits				28h
Prefetchable Limit Upper 32 Bits				2Ch
I/O Limit Upper 16 Bits		I/O Limit Base Upper 16 Bits		30h
Reserved			Capabilities Pointer	34h
Expansion ROM Base Address				38h
Bridge Control		Interrupt Pin	Interrupt Line	3Ch



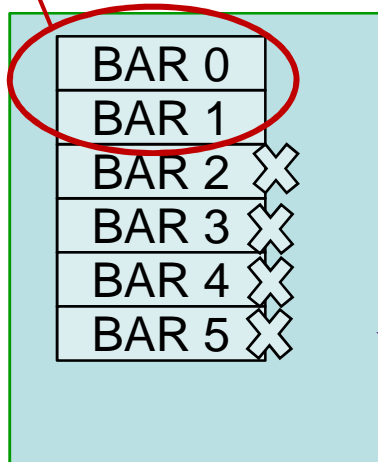
PCI karta zná adresy, které jí patří...

PCI karta č.0



Tohle je ale fyzická adresa !!!

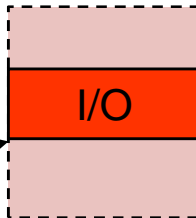
PCI karta č.1



Co vrací funkce mmap()?

Nezapomeňte na munmap()...

I/O adresní prostor (instrukce x86: in, out)



Paměťový prostor: Společný adresní prostor pro I/O a paměť

Paměť

Když procesor zapíše na tuto pozici, zápis uvidí PCI karta č.0. Jaký to bude mít význam záleží od této konkrétní karty.



mmap(BAR1)

předch. adr. +4
mmap(BAR0)

mmap(BAR1)

Tohle musí být (jsou) virtuální adresy !!!

Obsah adresáře /proc/bus/pci

The image shows a directory listing of /proc/bus/pci. At the top, there are folders 00, 01, 03, 04, 05, 06 and a file named devices. A box highlights the contents of folder 00, which includes files 00.0, 01.0, 1a.0, 1a.1, 1a.7, 1b.0, 1c.0, 1c.3, 1c.4, 1c.5, 1d.0, 1d.1, 1d.2, 1d.3, 1d.7, 1e.0, 1f.0, 1f.2, 1f.3, and 1f.5. To the right, a hex dump shows the binary data of these files. The hex dump has columns for each file and rows for memory addresses from 00000000 to 00000050. The data shows a repeating pattern of hex values, with some variations in the 00000020 and 00000030 rows.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00000000	72	11	32	1f	07	00	10	00	01	00	00	ff	08	00	00	00
00000010	00	00	8f	fe	00	00	00	00	00	00	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	72	11	32	1f
00000030	00	00	00	00	50	00	00	00	00	00	00	00	0b	01	00	00
00000040
00000050

- Jednotlivé soubory v těchto adresářích představují jednotlivé hlavičky PCI zařízení (64 Bajtů každý soubor)
- **Úkol na doma: Napište program v jazyku C/C++, který bude procházet soubory v daném adresáři a jeho podadresářích a bude hledat výskyt posloupnosti znaků (4B) na začátku každého souboru. V případě úspěchu vytiskněte cestu k souboru.**

Obsah adresáře `/proc/bus/pci` Příkaz: `lspci -vb`

00:00.0 Host bridge: Intel Corporation 82X38/X48 Express DRAM Controller

Subsystem: Hewlett-Packard Company Device 1308

Flags: bus master, fast devsel, latency 0

Capabilities: [e0] Vendor Specific Information <?>

Kernel driver in use: x38_edac

Kernel modules: x38_edac

00:01.0 PCI bridge: Intel Corporation 82X38/X48 Express Host-Primary PCI Express Bridge

Flags: bus master, fast devsel, latency 0

Bus: primary=00, secondary=01, subordinate=01, sec-latency=0

I/O behind bridge: 00001000-00001fff

Memory behind bridge: f0000000-f2ffffff

Kernel driver in use: pcieport

Kernel modules: shpchp

00:1a.0 USB Controller: Intel Corporation 82801I (ICH9 Family) USB UHCI Controller #4 (rev 02)

Subsystem: Hewlett-Packard Company Device 1308

Flags: bus master, medium devsel, latency 0, IRQ 5

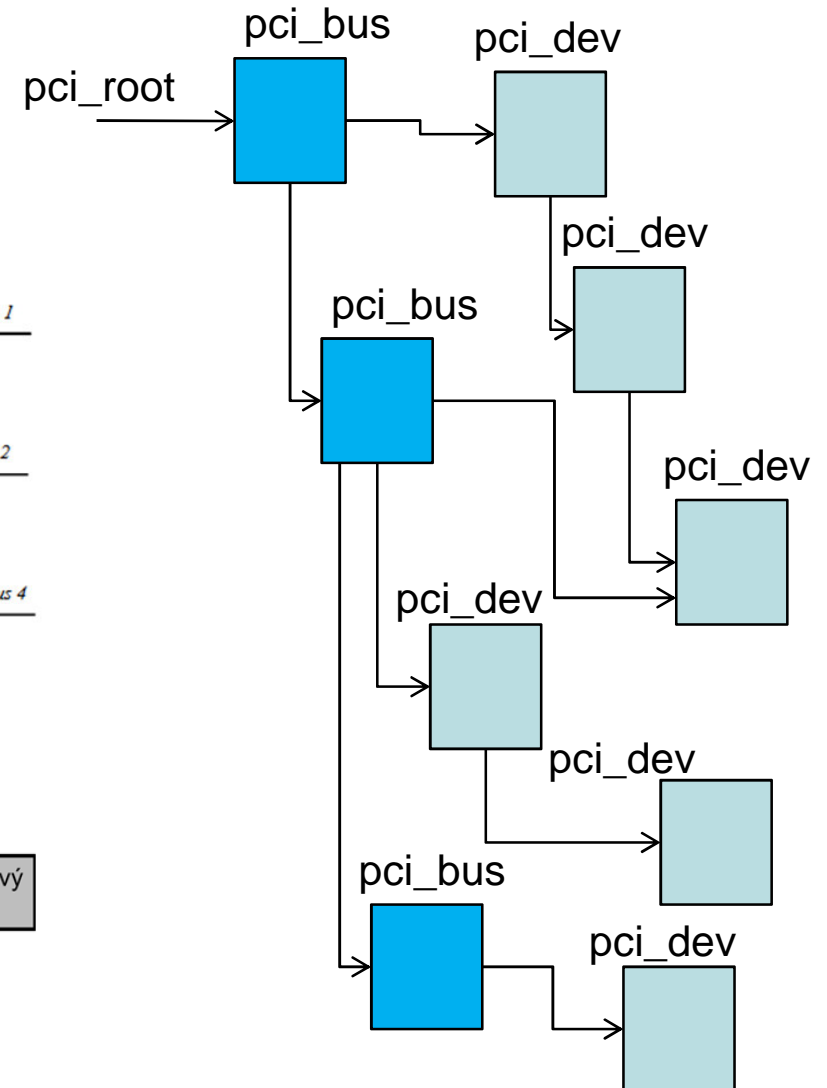
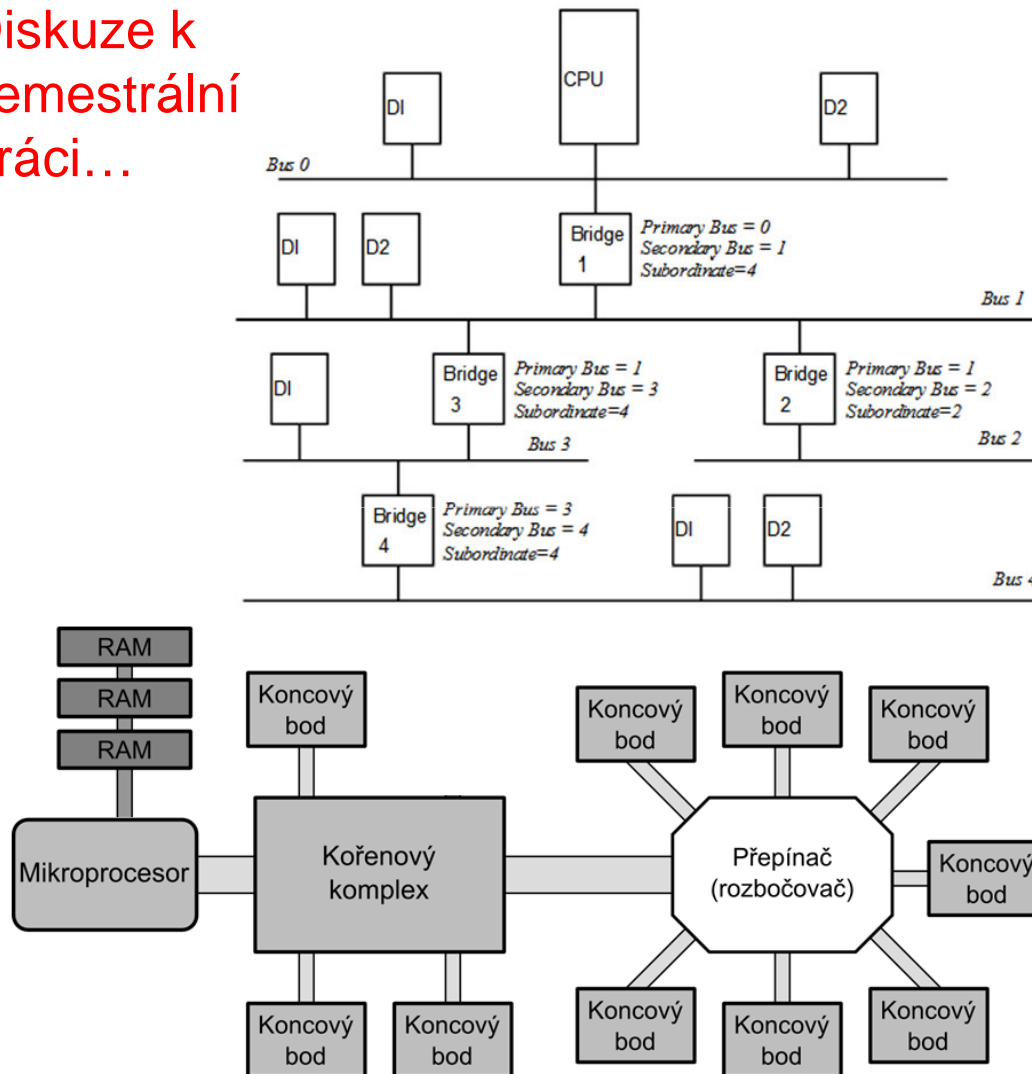
I/O ports at 2100

Capabilities: [50] PCI Advanced Features

Kernel driver in use: uhci_hcd

PCI vs. PCI Express - Umíte si již představit jak to funguje?

Diskuze k
semestrální
práci...



Volání funkcí a předávání parametrů – Úvod

Další podrobnosti v přednášce č.9

Rozdělení paměti

Text segment:

program (256MB): 4 nejvýzn. bity vždy 0...

Global data segment:

globální proměnné (64 KB)
přístupné přes \$gp (inicializován na 0x10008000) a 16-bitový offset

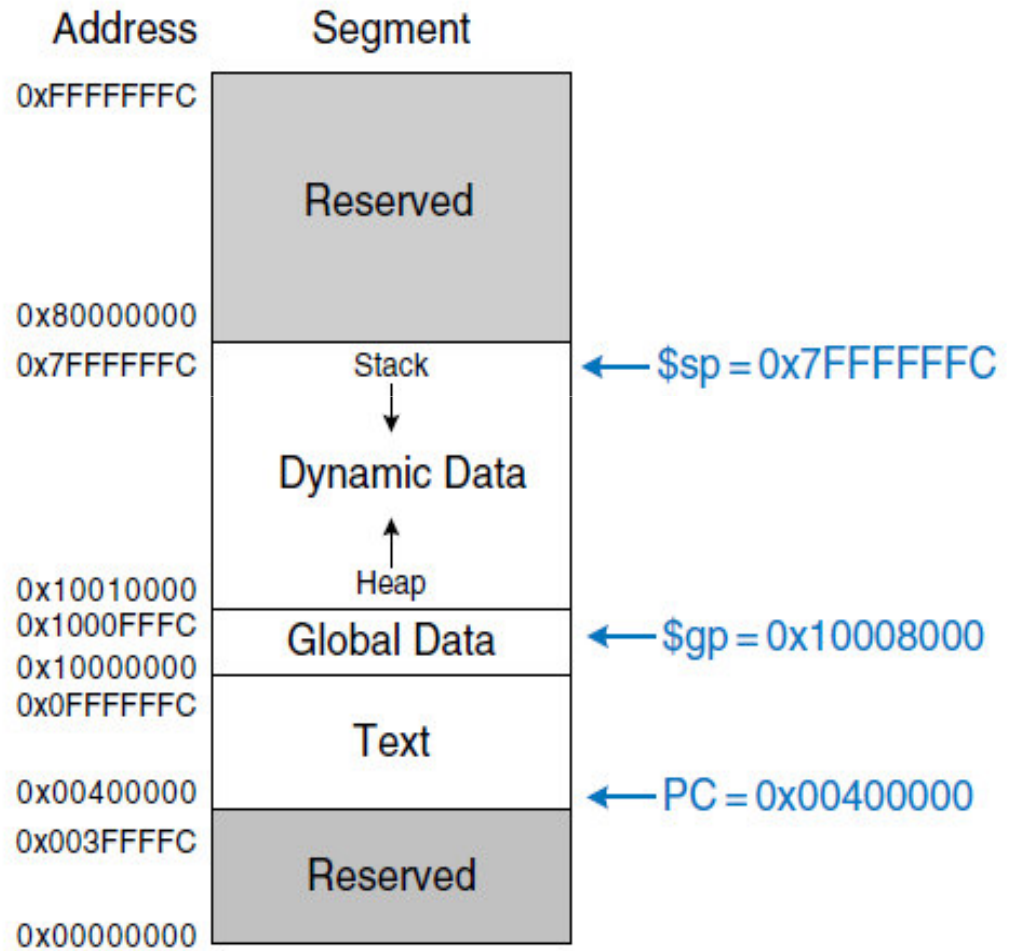
Dynamic Data Segment:

Uchovává Stack a Heap (2 GB).
Dynamicky alokováno. Stack –
lokální proměnné, záloha registrů;
Heap - malloc

Reserved Segments:

Rezervováno OS. Nepřístupné
přímo. Část prostoru pro přerušení,
část pro paměťově mapované I/O.

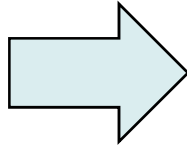
MIPS32: Adresa 32 bitů => 4GB



Jednoduchý příklad

C/C++

```
int main() {  
    simple();  
    ...  
}  
  
void simple(){  
    return;  
}
```



MIPS

```
0x00400200 main: jal simple  
0x00400204 ...  
  
0x00401020 simple: jr $ra
```

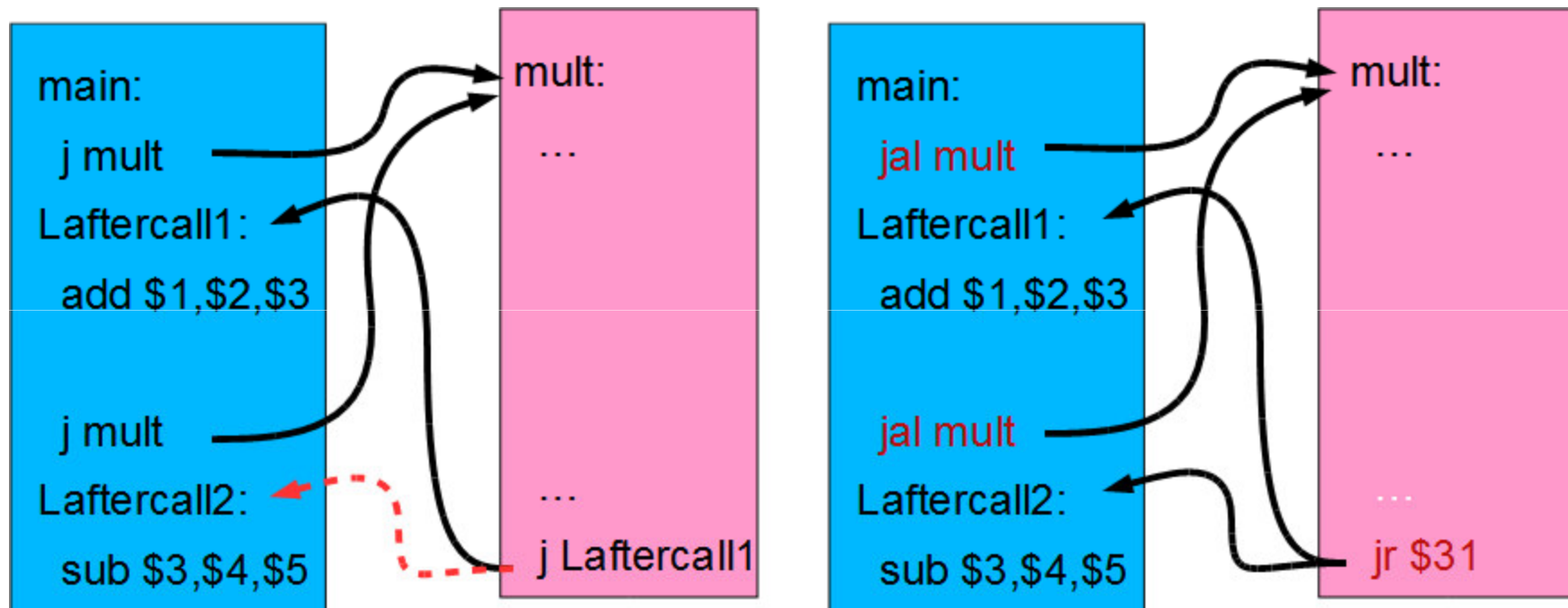
JAL -- Jump and link

Description:	For procedure call.
Operation:	$\$31 = PC + 8; PC = ((PC+4) \& 0xf0000000) (target \ll 2)$
Syntax:	jal target
Encoding:	0000 11ii iiiiiiii iiiiiiii iiiiiiii iiiiiiii

Volající (caller) uloží návratovou adresu do **\$ra** (č.31) a skočí na návěstí volaného (calle) – to vše zabezpečí instrukce **jal**. Volaný nesmí přepsat registry (architectural state) ani paměť, kterou používá volající. Volaný se vrátí pomocí instrukce **jr**.

Rozdíl mezi voláním a skokem

Skok neuloží návratovou hodnotu
kód tedy nejde využít z více míst



naopak volání s využitím registru ra
**umožňuje volat podprogram tehdy,
kdy je potřeba**

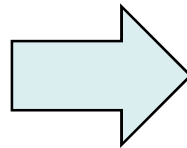
Autor příkladu Prof. Sireer
Cornell University

Co když má funkce něco vrátet nebo má argumenty?

- \$a0–\$a3 pro argumenty
- \$v0–\$v1 pro návratovou hodnotu

C/C++

```
int main() {  
    int y;  
    y=fun(2,3,4,5)  
    ...  
}  
  
int fun(int a, int b,  
int c, int d)  
{  
    int res;  
    res = a+b - (c+d);  
    return res;  
}
```



MIPS

```
main:  
addi $a0, $0, 2  
addi $a1, $0, 3  
addi $a2, $0, 4  
addi $a3, $0, 5  
jal fun  
add $s0, $v0, $0  
  
fun:  
add $t0, $a0, $a1  
add $t1, $a2, $a3  
sub $s0, $t0, $t1  
add $v0, $s0, $0  
jr $ra
```

Volaný ale nesmí měnit mezivýsledky volajícího...!!!

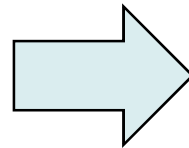
- tzn. volaný musí zachovat obsahy registrů, které volající používá!
Které registry to ale jsou?
- Volaný proto uloží registry, které chce použít na zásobník (stack) před tím než je modifikuje a obnoví je před tím než se vrátí. Podrobněji:
 1. vytvoří prostor na zásobníku pro zálohování registrů (nebo lokálních proměnných)
 2. zálohuje hodnoty pracovních registrů
 3. vykoná vlastní tělo funkce
 4. obnoví registry použitím zásobníku
 5. dealokuje prostor na zásobníku
- Prostrou na zásobníku, který vytvoří procedura hovoříme **Stack frame**
- Na vrchol zásobníku ukazuje **Stack pointer** – registr **\$sp**

Volaný ale nesmí měnit mezivýsledky volajícího...!!!

- *Pozn: Ne všechny registry je nutno zálohovat – dáno volací konvencí.*

C/C++

```
int fun(int a, int b,  
int c, int d)  
{  
    register int res;  
    res = a+b - (c+d);  
    return res;  
}
```



MIPS

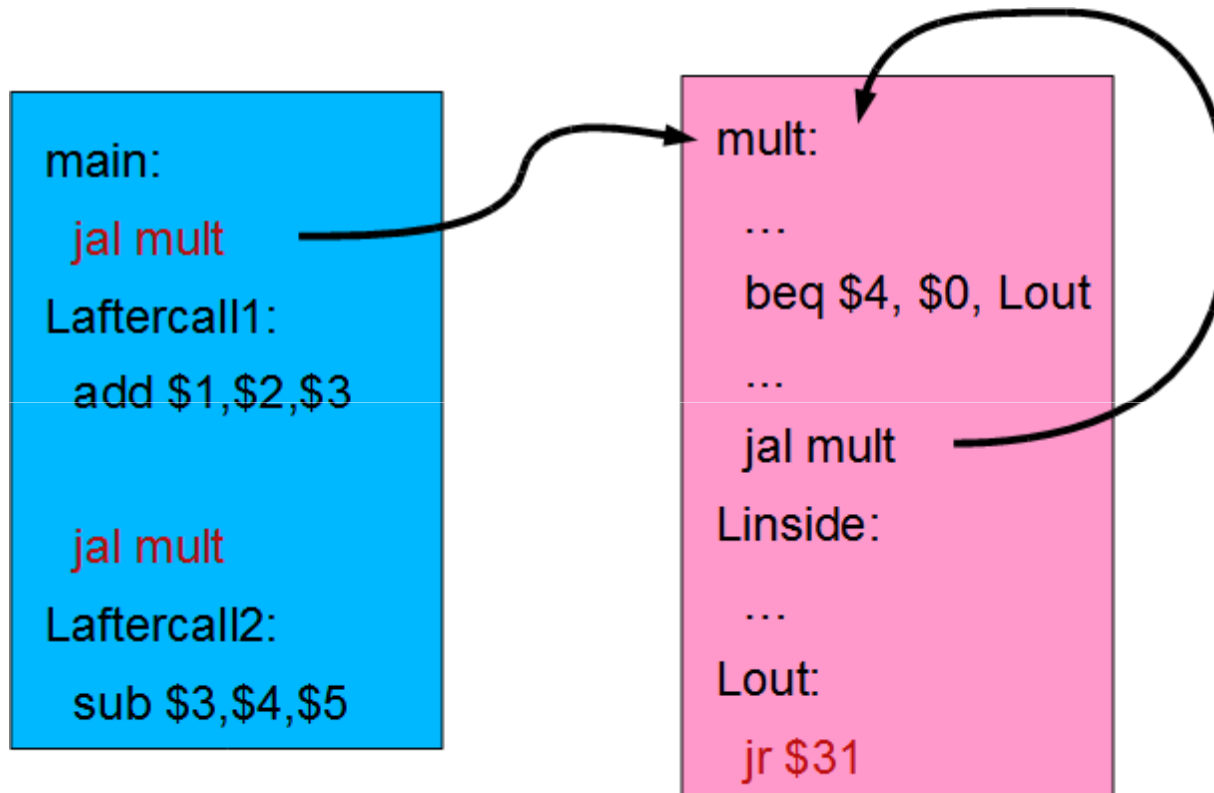
```
fun:  
addi $sp, $sp, -4    //alokace  
sw $s0, 0($sp)      //záloha  
add $t0, $a0, $a1  
add $t1, $a2, $a3  
sub $s0, $t0, $t1  
add $v0, $s0, $0  
lw $s0, 0($sp)      //obnovení  
addi $sp, $sp, 4    //dealokace  
jr $ra
```

Které registry mám tedy zálohovat?

Odověď: Všechny (vyjma návratové hodnoty). Některé zálohuje volající (pokud je používá), jiné volaný (když je modifikuje).

Preserved	Nonpreserved
Saved registers: $\$s0-\$s7$	Temporary registers: $\$t0-\$t9$
Return address: $\$ra$	Argument registers: $\$a0-\$a3$
Stack pointer: $\$sp$	Return value registers: $\$v0-\$v1$
Stack above the stack pointer	Stack below the stack pointer
<i>callee-save</i>	<i>caller-save</i>

Problém vícenásobného volání s link-registrem

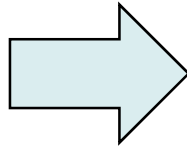


Registr **ra** je potřeba někam uložit stejně jako ukládané registry sX

Rekurze?

C/C++

```
int fac(int n) {  
    if (n<=1)  
        return 1;  
    else  
        return n*fac(n-1);  
}
```

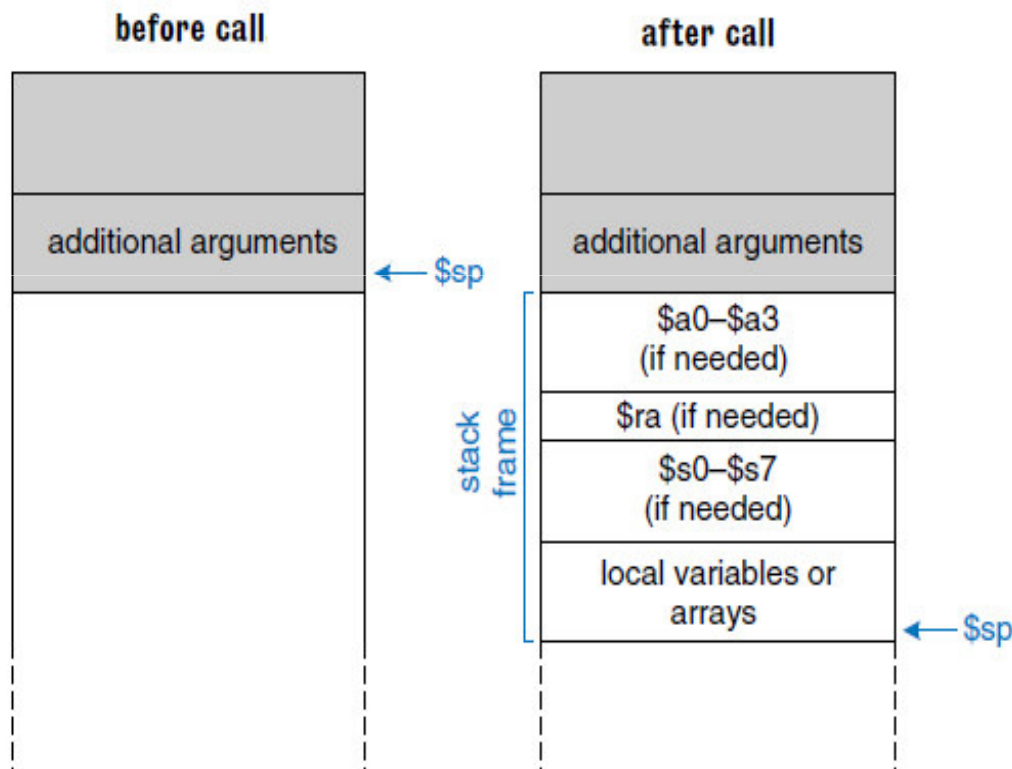


MIPS

```
fac:  addi $sp, $sp, -8  
      sw $a0, 4($sp)  
      sw $ra, 0($sp)  
      addi $t0, $0, 2  
      slt $t0, $a0, $t0  
      beq $t0, $0, else  
      addi $v0, $0, 1  
      addi $sp, $sp, 8  
      jr $ra  
else: addi $a0, $a0, -1  
      jal fac  
      mul $v0, $a0, $v0  
      lw $ra, 0($sp)  
      lw $a0, 4($sp)  
      addi $sp, $sp, 8  
      jr $ra
```

Co když je parametrů víc než 4?

- MIPS konvence: První čtyři před registry, další na stack hned nad stack pointer. Volající vytváří prostor pro tyto parametry..



```
int main(){
    simple(1,2,3,4,5,6);
    return 0;
}

addiu    $sp,$sp,-8
addi     $2,$0,5      # 0x5
sw       $2,4($sp)
addi     $2,$0,6      # 0x6
sw       $2,0($sp)
addi     $a0,$0,1     # 0x1
addi     $a1,$0,2     # 0x2
addi     $a2,$0,3     # 0x3
addi     $a3,$0,4     # 0x4
jal      simple
```

x86

```
int simple(int a, int b, int c, int d, int e, int f){
    return a-f;
}
int main(){
    int x;
    x=simple(1, 2, 3, 4, 5, 6);
    return 0;
}
```

<pre>_simple: pushl %ebp movl %esp, %ebp movl 28(%ebp),%eax movl 8(%ebp),%edx movl %edx, %ecx subl %eax, %ecx movl %ecx, %eax popl %ebp ret</pre>	<pre>_main: pushl %ebp movl %esp, %ebp andl \$-16, %esp subl \$48, %esp call ___main movl \$6, 20(%esp) movl \$5, 16(%esp) movl \$4, 12(%esp) movl \$3, 8(%esp) movl \$2, 4(%esp) movl \$1, 0(%esp) call _simple movl %eax, 44(%esp) movl \$0, %eax leave ret</pre>	<p>ebp na stack, pozor: push meni esp...</p> <p>esp do ebp</p> <p>zarovnani na 16-byte</p> <p>esp = esp - 48</p> <p>posledni argument</p> <p>predposledni</p> <p>...</p> <p>...</p> <p>...</p> <p>prvni argument</p> <p>volani funkce</p> <p>prirazeni vysledku x = simple(...);</p> <p>return 0;</p>
--	--	---

Literatura

- http://en.wikipedia.org/wiki/PCI_configuration_space
- **David Money Harris and Sarah L. Harris:** Digital Design and Computer Architecture, Morgan Kaufmann (2007)
- **John L. Hennessy and David A. Patterson:** Computer Architecture: A Quantitative Approach, 3rd Edition, Morgan Kaufmann (2003)