

Architektura počítače

Instruction Set Architectures

Formáty instrukcí MIPS

Reprezentace instrukcí

Definice

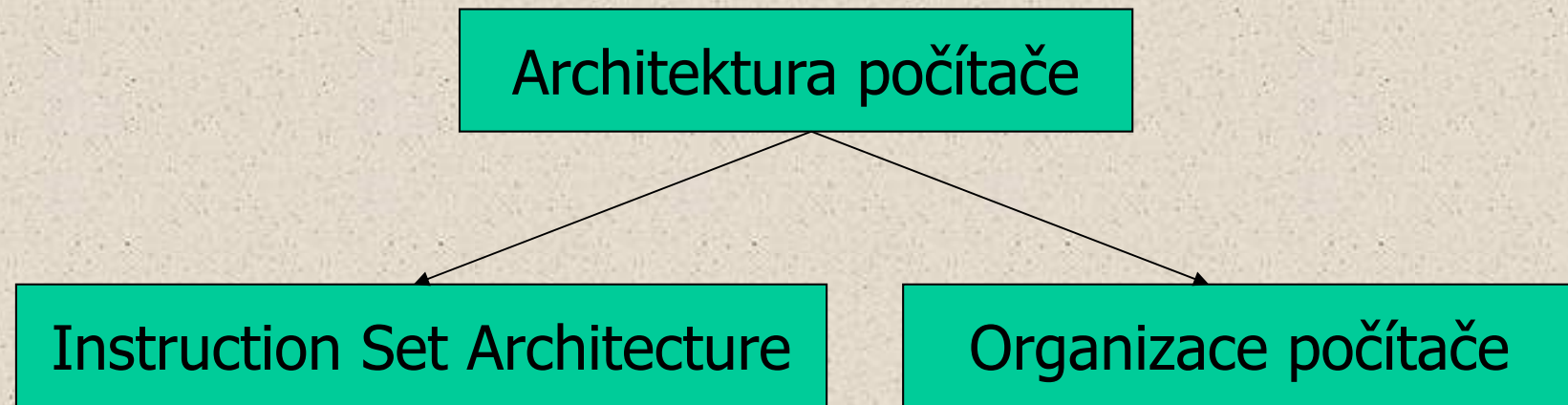
- Co je “architektura?”
 - “The art or science of building...the art or practice of designing and building structures...” (Webster Dictionary)
 - “včetně plánu, návrhu, konstrukce a dekorace...” (American College Dictionary)
- Co je “architektura počítače?”
 - “... architekturou rozumíme strukturu modulů, jak jsou organizovány v počítačovém systému ...” (H. Stone, 1987)
 - “Architektura počítače je interface mezi strojem a softwarem” (Andris Padges, architekt IBM 360/370)

Definice

- Co je “architektura počítače?”
 - Amdahl, Blaauw, Brooks (IBM 1964):

“... the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine”.

Dvě hlavní oblasti



- Architekturu počítače můžeme rozdělit do dvou oblastí:
 - ISA (Instruction Set Architecture)
 - Organizace počítače

ISA (Instruction Set Architecture)

- ISA (Instruction Set Architecture) je definována jako:

Atributy [počítačového] systému z hlediska programátora, to znamená strukturu a funkční chování, na rozdíl od organizace a řízení datových toků, logického návrhu a fyzické implementace. (Amdahl, Blaaw a Brooks 1964)

- ISA zahrnuje takové volby a rozhodnutí, jako například:
 - Datové typy & struktury (kódování & reprezentace)
 - Instrukční soubor
 - Instrukční formáty
 - Adresní módy a přístup k datům a instrukce
 - Podmínky výjimek

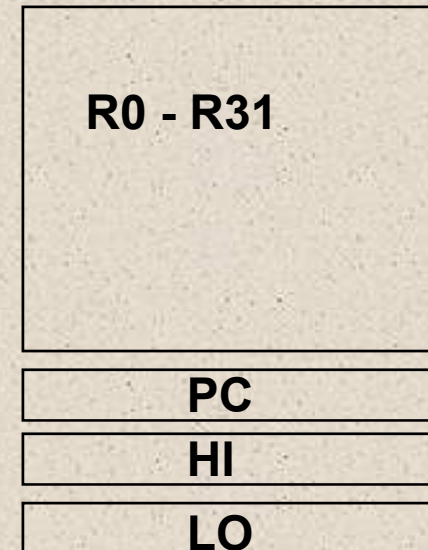
ISA

- Instrukční soubor lze chápat jako interface mezi softwarem a hardwarem – představuje abstraktní formu hardwaru. Odděluje celou složitost implementačních detailů od software.
- Příklady různých ISA zahrnují...
 - Intel IA-32 (x86)
 - Intel IA-64
 - DEC Alpha
 - MIPS
 - Sparc (a UltraSparc)
- Například softwarový vývojář vyvíjí software pro ISA, např. IA-32. Aktuální hardwarová implementace není rozhodující a může se lišit systém od systému, pokud je zachována ISA.
(Z toho důvodu může tentýž program být zpracováván na Pentiu i na Pentiu 4, což jsou z hlediska stavby velmi rozdílné procesory)

Příklad: ISA procesoru MIPS R3000

- Šest hlavních typů instrukcí
 - Čtení/zápis (Load/Store)
 - Aritmetické
 - Skoky a větvení
 - Floating Point operace
 - Správa paměti (Memory Management)
 - Speciální
- Tři formáty instrukcí – všechny o šíři 32 bitů ...

Registry (zde 35)

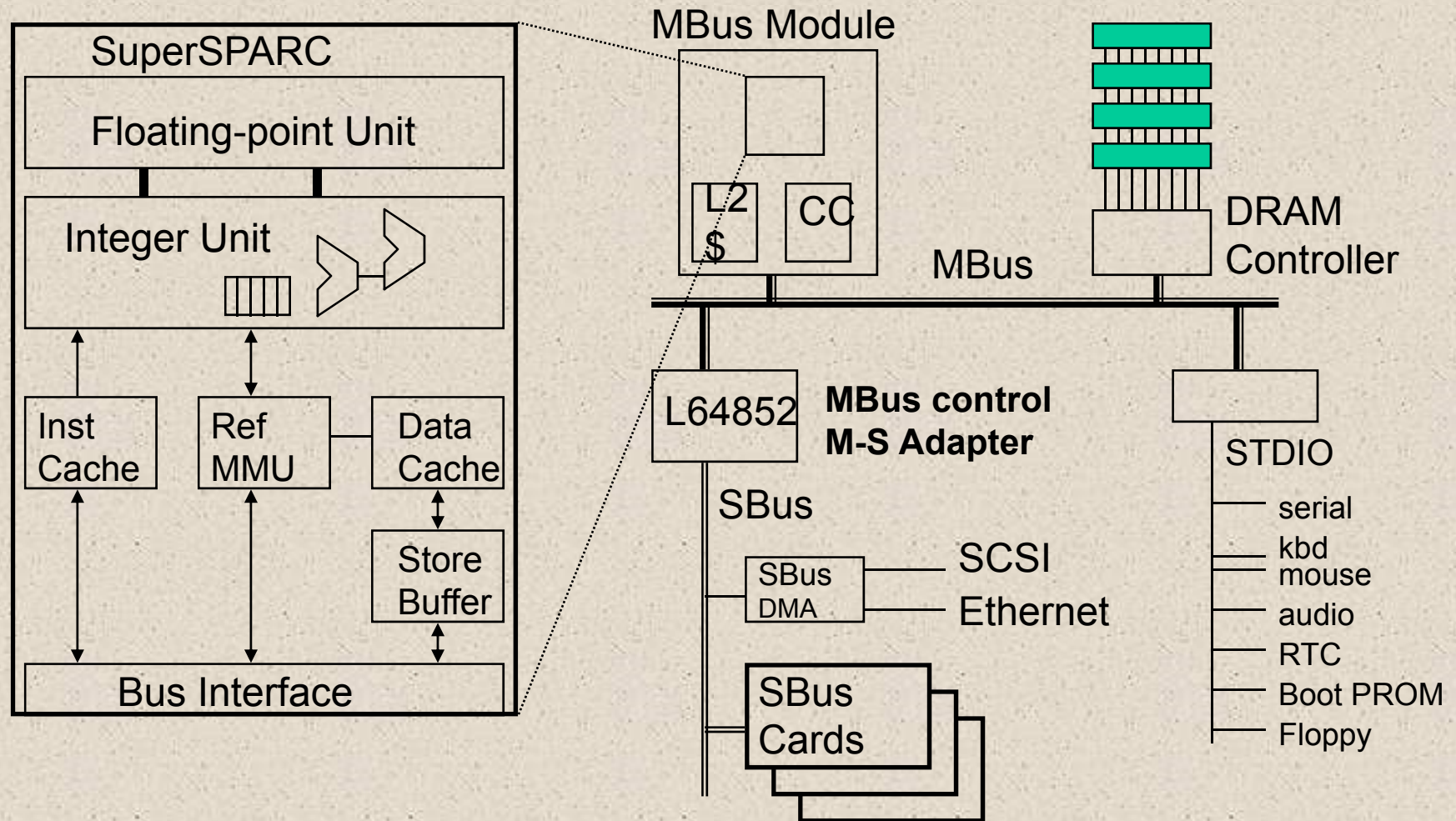


opcode	rs	rt	rd	sa	funct
opcode	rs	rt	immediate value		
opcode	jump or branch target				

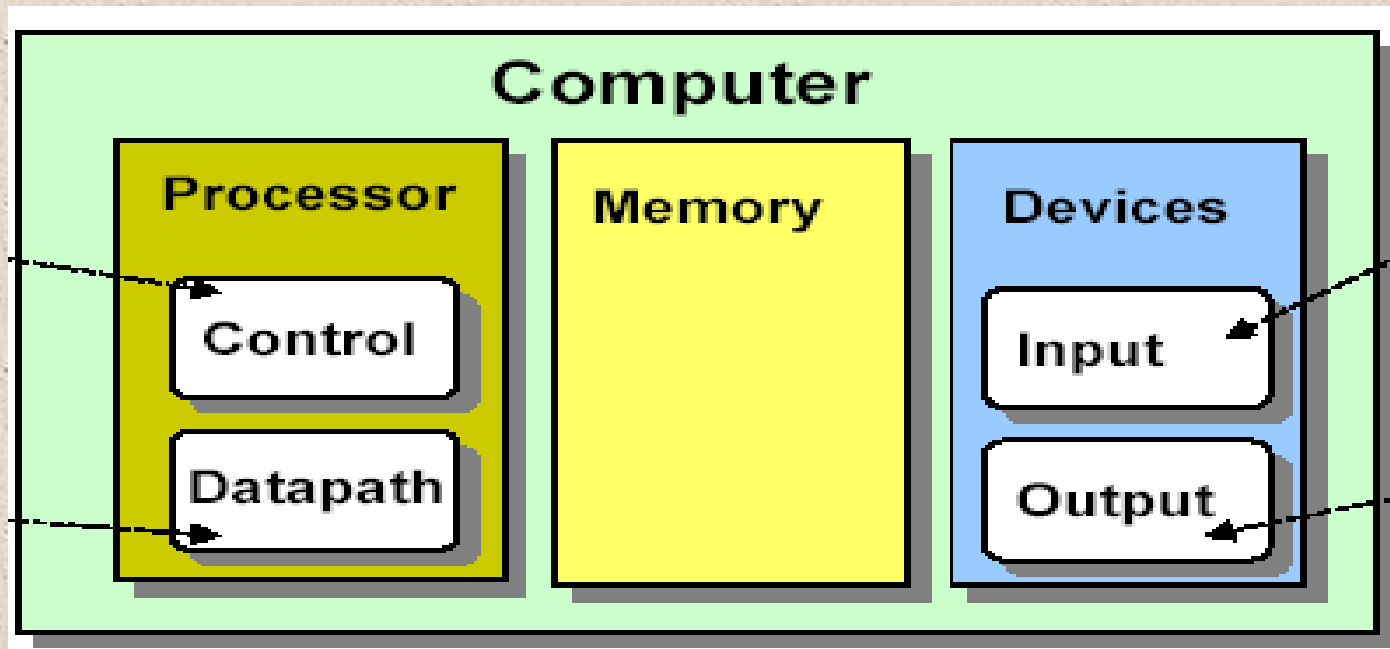
Organizace počítače

- Organizace počítače zahrnuje implementační detaily – jak vlastně hardware doopravdy pracuje...
 - Vlastnosti a výkonové charakteristiky funkčních jednotek (jako např. registrů, ALU, posuv. jednotek, logických jednotek atd.)
 - Propojení těchto komponent
 - Informační toky mezi komponentami
 - Řízení toku informací
 - Kombinace funkčních jednotek pro realizaci ISA
 - Popis RTL (Register Transfer Level)

Příklad: TI SuperSPARC



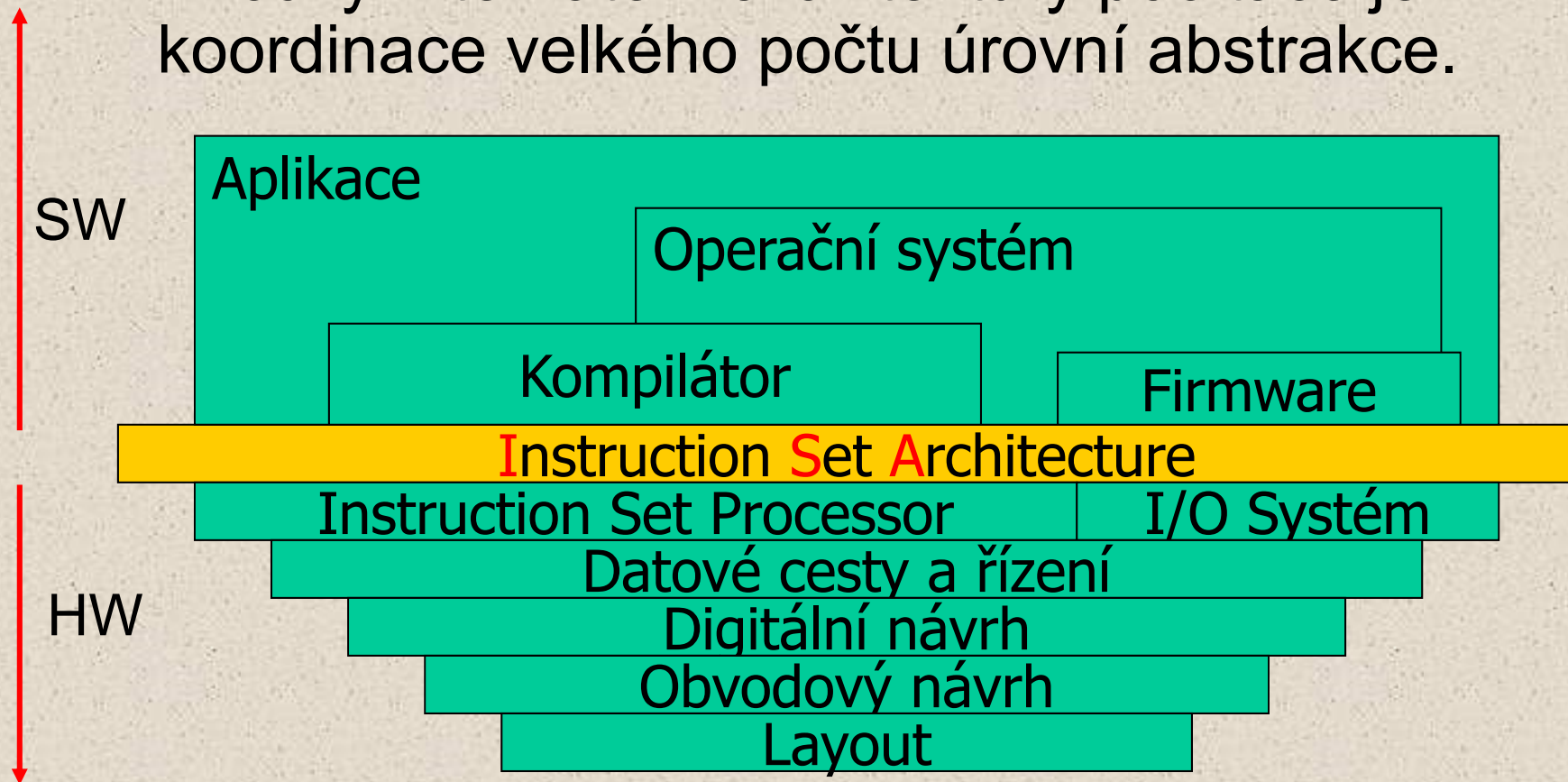
Organizace počítače - obecně



- Každá část každého počítače může být zařazena do jedné z pěti skupin.
- Návrh každého systému je podřízen požadavkům na cenu, velikost a schopnosti každé komponenty. Typický návrh uvažuje: procesor 25% celkových nákladů, paměť 25% celkových nákladů a 50% celkových nákladů ostatní zařízení.

Klíčové téma: abstrakce

- Klíčovým tématem architektury počítače je koordinace velkého počtu úrovní abstrakce.



Jak se budeme architekturou zabývat?

- Architekturou se budeme zabývat od základů, směrem zdola nahoru.
- Budeme se zabývat návrhem hardware, včetně návrhu některých komponent (použití HDL).
- Cílem je studie architektury, jako směs teorie a také praktických příkladů. Budeme se zabývat návrhem, simulací a verifikací.

Zvolený přístup

- Klasický návrh mikroprocesoru, který je použit k ilustraci je MIPS (navržen autory učebnice - D. A. Patterson and J. L. Hennessy: Computer Organization and Design: The Hardware Software Interface, **4rd Edition**, 2009).
- Vedle této základní architektury se také částečně zmíníme o dalších typech architektur jako např. PowerPC a Intel IA-32/x86.
- MIPS je relativně stará architektura – je jednoduchá a vyhýbá se mnoha složitostem, které se v současných systémech používají (**což je velmi vhodné pro výuku**).
- Tento přístup dovolí přiblížit různé přístupy k návrhu a řešení problémů.

Přehled úrovní ISA

- *ISA: jak se počítač jeví programátoru na úrovni strojního jazyka (kompilátor)*
 - Paměťový model
 - Instrukce
 - Formáty
 - Typy (aritmetické, logické, přesuny dat a řízení výpočtu)
 - Režimy (jádro a uživatel)
 - Operandy
 - Registry
 - Datové typy
 - Adresování
- Dokumenty s formální definicí ISA
 - V9 SPARC a JVM

Programovací jazyky

Jazyky vyšší úrovně

```
temp  = v[k];  
v[k]  = v[k+1];  
v[k+1] = temp;
```

```
TEMP = V(K)  
V(K)  = V(K+1)  
V(K+1) = TEMP
```

Kompilátor C/Java

Kompilátor Fortranu

Asembler

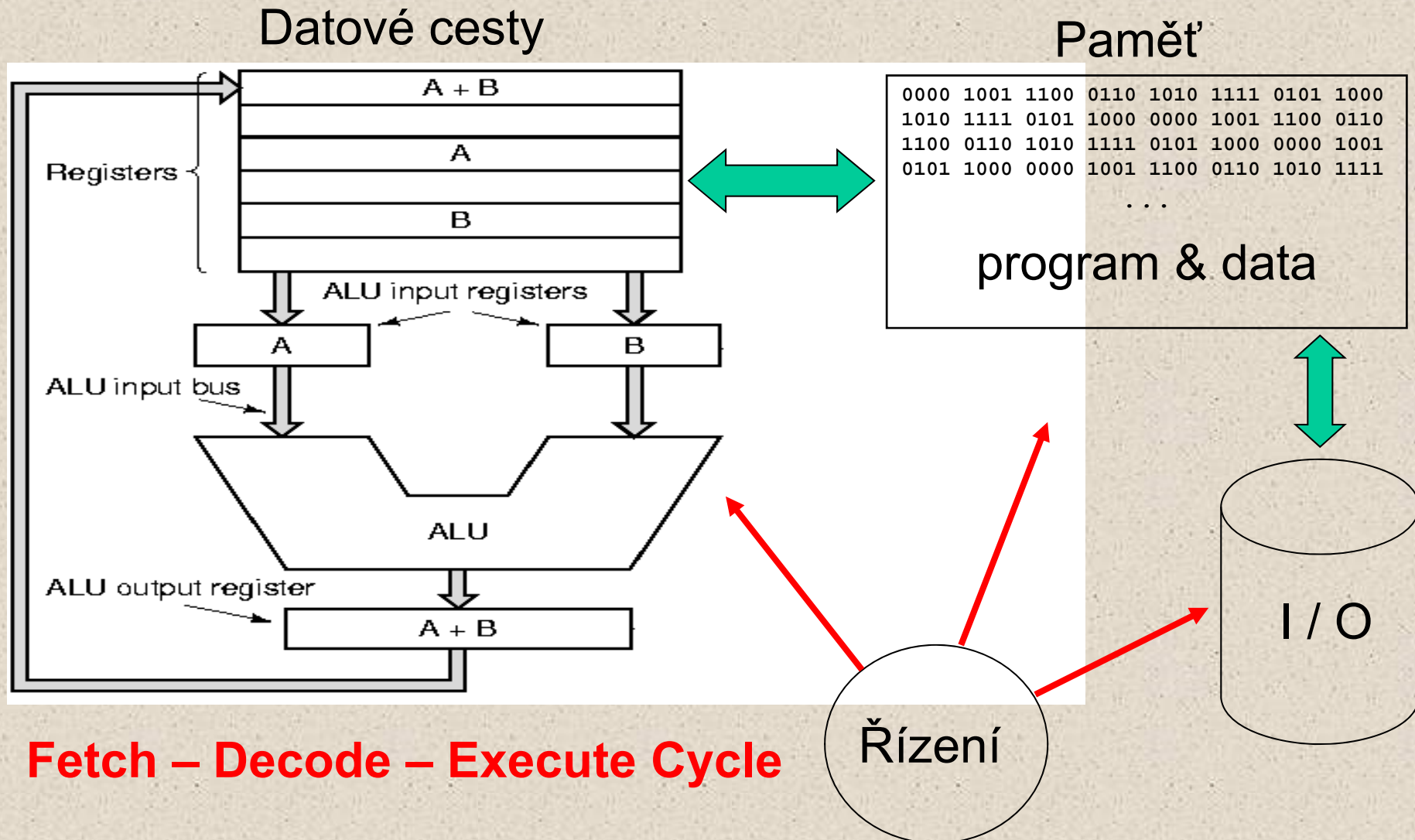
```
lw $t0, 0($2)  
lw $t1, 4($2)  
sw $t1, 0($2)  
sw $t0, 4($2)
```

Assembler MIPS

Strojní jazyk

```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```

Výpočet programu



Strojový jazyk MIPS

- Aritmetické instrukce
 - add, sub, mult, div
- Logické instrukce
 - and, or, ssl (shift left), srl (shift right)
- Přesuny dat
 - lw (load), sw (store), lui (load upper immediate)
- Větvení
 - Podmíněné skoky: beq, bne, slt (set on less than)
 - Nepodmíněné skoky: j, jr (jump register), jal (jump and link)

Instrukce MIPS

- Jednoduché instrukce
- **Zásada_1: Jednoduchost podporuje regularitu**

add a, b, c

Právě 3 operandy (registry)

a = b + c;

poznámka

a = b + c + d + e;

add a, b, c

add a, a, d

add a, a, e

- Tyto instrukce jsou **symbolickou** reprezentací toho, čemu MIPS „rozumí“

Příklad

Kompilace přiřazení v C do MIPS

$f = (g + h) - (i + j);$

Add	t0, g, h	# dočasná proměnná t0 obsahuje g+h
Add	t1, i, j	# dočasná proměnná t1 obsahuje i+j
Sub	f, t0, t1	# do f je uložen výsledek

Operandy

- Operand nemůže být libovolná proměnná (jako v C)
 - princip KISS – odstraňuje nedostatky CISC
- Registry (Keep It Small and Simple)
 - Omezený počet (32 32-bitových registrů u MIPS)
 - **Zásada_2: Menší je rychlejší**
 - Pojmenování: čísla nebo *jména*
 - \$8 - \$15 => \$t0 - \$t7 (vztahuje se na dočasnou proměnnou)
 - \$16 - \$22 => \$s0 - \$s8 (vztahuje se na proměnnou z C)
 - Jména zvýší srozumitelnost vašeho kódu

$f = (g + h) - (i + j);$
↑ ↑ ↑ ↑ ↑
\$s0 \$s1 \$s2 \$s3 \$s4



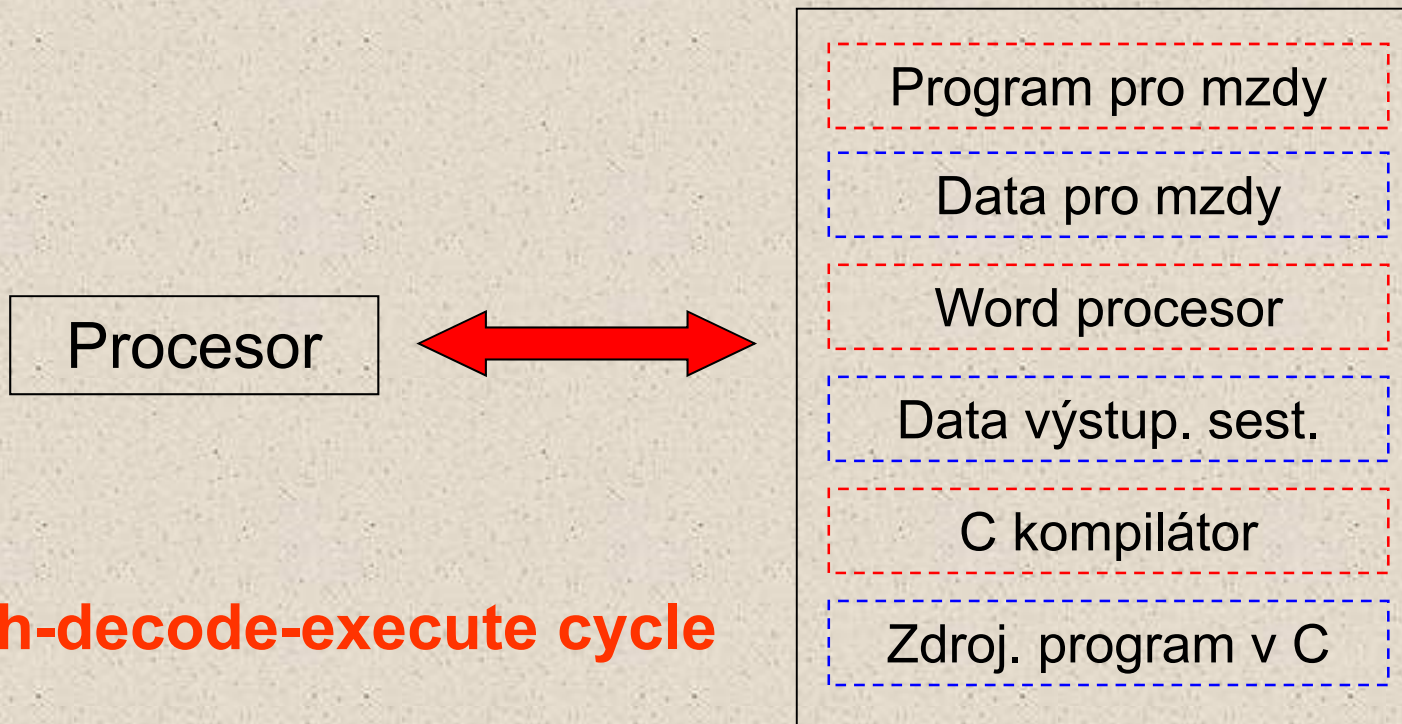
```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```


Registry - konvence

Jméno	Číslo registru	Použití	Vyhrazen při call
\$zero	0	the constant value 0	n.a.
\$at	1	reserved for the assembler	n.a.
\$v0-\$v1	2-3	value for results and expressions	no
\$a0-\$a3	4-7	arguments (procedures/functions)	yes
\$t0-\$t7	8-15	temporaries	no
\$s0-\$s7	16-23	saved	yes
\$t8-\$t9	24-25	more temporaries	no
\$k0-\$k1	26-27	reserved for the operating system	n.a.
\$gp	28	global pointer	yes
\$sp	29	stack pointer	yes
\$fp	30	frame pointer	yes
\$ra	31	return address	yes

Koncepce „programu v paměti“

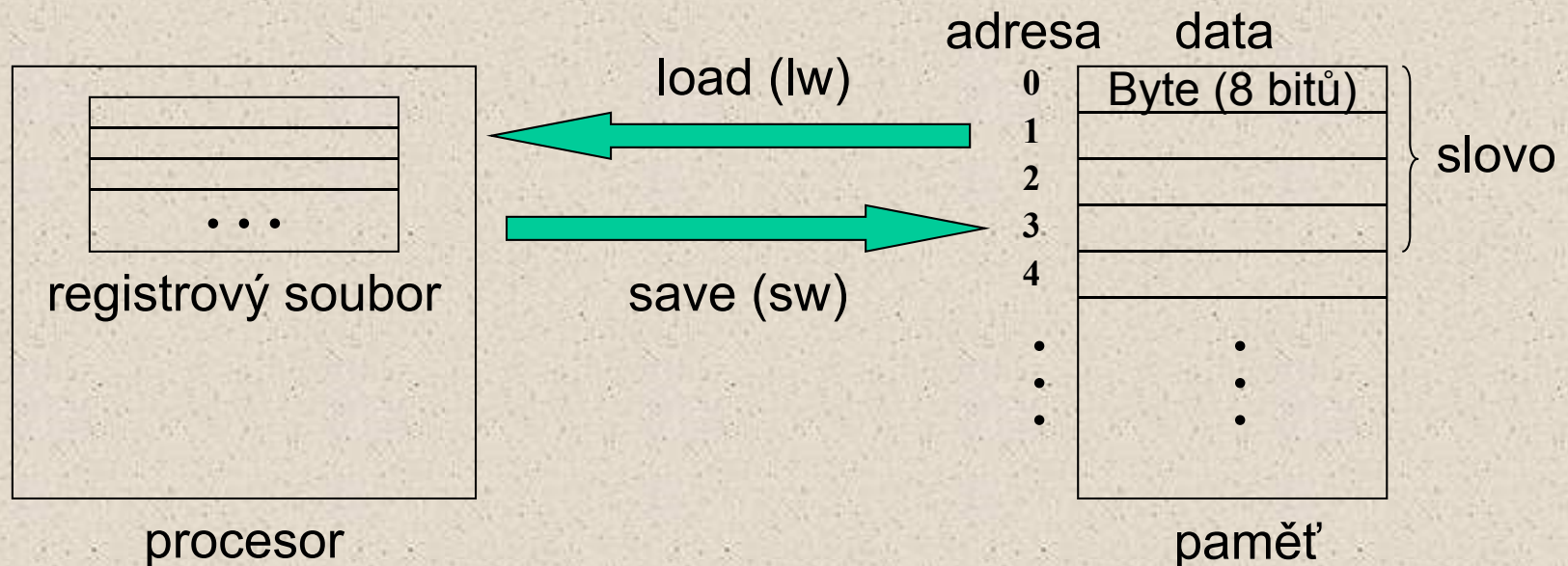
- Instrukce reprezentují 32-bitová čísla
- Programy jsou uloženy v paměti
 - Lze je číst i zapisovat stejně jako čísla



Fetch-decode-execute cycle

Paměť

- Obsahuje instrukce a data programu
 - Instrukce jsou čteny *automaticky* řadičem
 - Data jsou přesouvána explicitně tam a zpět mezi pamětí a procesorem
- Instrukce přesunu dat



Paměťový model

- Paměť je adresovatelná po bytech (1 byte = 8 bitů)
- Load/Store - jediný přístup k datům v paměti
- Jednotka pro přenos: *word* (4 byty)
 - $M[0], M[4], M[4n], \dots, M[4,294,967,292]$
- **Slova musí být zarovnána !!!**
 - Slovo začíná na adresách 0, 4, ... 4n
- Adresa je dlouhá 32 bitů
 - 2^{32} bytů nebo 2^{30} slov

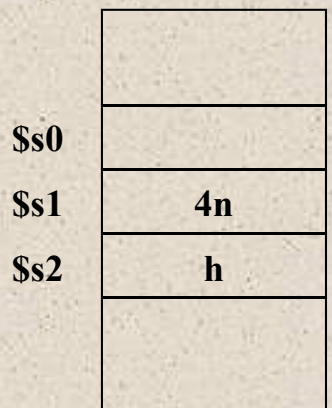
Load / Store

$A[0] = h + A[2];$

lw \$t0, 8(\$s1)

add \$t0, \$s2, \$t0

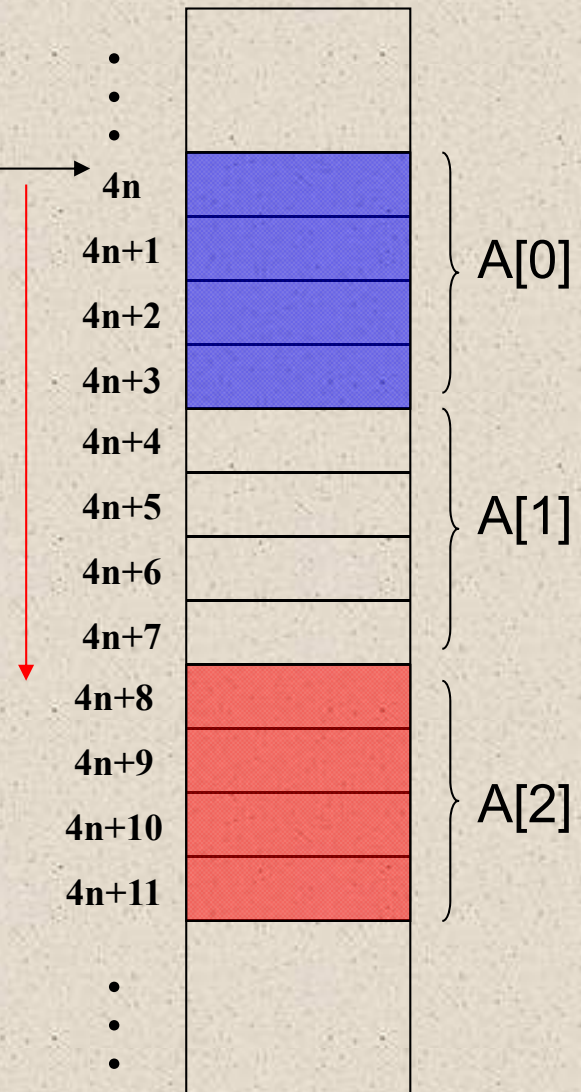
sw \$t0, 0(\$s1)



registry

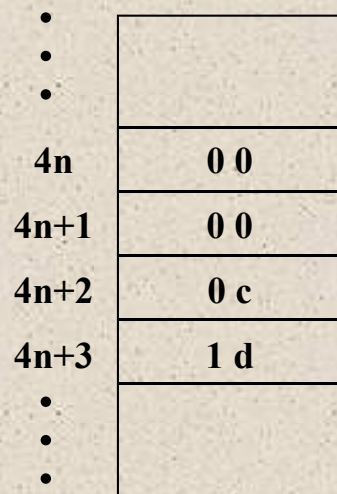
bázová adresa
(\$s1)

Offset (8)



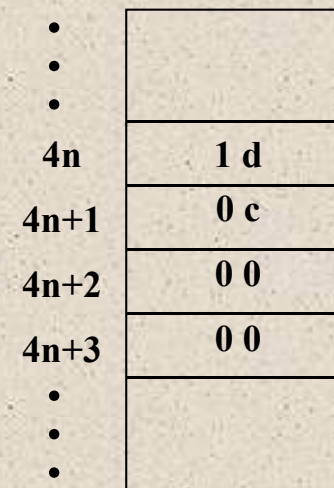
„Big“ a „Little“ Endian

$$(3101)_{10} = 12 * 16^2 + 1 * 16^1 + 13 * 16^0$$
$$= (00 \ 00 \ 0c \ 1d)_{16}$$



big endian

(MIPS R3000)



little endian

(DEC, Intel)

Přehled

- ISA: představuje interface pro hardware / software
 - Principy návrhu, využití
- Instrukce MIPS
 - Aritmetické: `add/sub $t0, $s0, $s1`
 - Přenos dat: `lw/sw $t1, 8($s1)` (znamená load/store-word)
- Operandy musí být registry
 - 32 32-bitových registrů
 - `$t0 - $t7` („dočasné“) mají adresy `$8 - $15`
 - `$s0 - $s7` („ukládané“) mají adresy `$16 - $23`
- Paměť: velké, jednorozměrné pole bytů $M[2^{32}]$
 - Adresa v paměti je index do toho pole bytů
 - Zarovnaná slova: `M[0], M[4], M[8], ..., M[4,294,967,292]`
 - Big/little endian – pořádek bytů ve slově

Strojový jazyk

- Všechny instrukce mají stejnou délku (32 bitů)
- **Zásada_3: Dobrý návrh vyžaduje dobré kompromisy**
 - Stejná délka nebo stejný formát
- Tři různé formáty instrukcí
 - R: formát aritmetických instrukcí
 - I: formát pro přesuny dat, větvení, immediate
 - J: formát skokové instrukce
- `add $t0, $s1, $s2`
 - 32 bitů ve strojní instrukci
 - Pole pro:
 - Operaci (add)
 - Operandy (\$s1, \$s2, \$t0)

```
10101101001010000000010010110000
00000010010010000100000000100000
10001101001010000000010010110000
```

```
lw    $t0, 1200($t1)
add   $t0, $s2, $t0
sw    $t0, 1200($t1)
```

```
A[300] = h + A[300];
```


Formáty instrukcí

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R:	op	rs	rt	rd	shamt	funct
I:	op	rs	rt	address / immediate		
J:	op	target address				

op: základní operace (opcode)

rs: první operand - registr

rt: druhý operand - registr

rd: registr pro uložení výsledku

shamt: délka posuvu

funct: vybírá specifickou variantu operačního kódu (funkční kód)

address: offset pro instrukce typu load/store ($\pm 2^{15}$)

immediate: konstanty pro instrukce s přímými operandy („immediate op.“)

Formát R

add \$t0, \$s1, \$s2 (add \$8, \$17, \$18 # \$8 = \$17 + \$18)

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

sub \$t1, \$s1, \$s2 (sub \$9, \$17, \$18 # \$9 = \$17 - \$18)

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
0	17	18	9	0	34
000000	10001	10010	01001	00000	100010

Formát I

lw \$t0, 52(\$s3)

lw \$8, 52(\$19)

6 bits	5 bits	5 bits	16 bits
35	19	8	52
100011	10011	01000	0000 0000 0011 0100

sw \$t0, 52(\$s3)

sw \$8, 52(\$19)

6 bits	5 bits	5 bits	16 bits
43	19	8	52
101011	10011	01000	0000 0000 0011 0100

Příklad

`A[300] = h + A[300];` `/* $t1 <= base of array A; $s2 <= h */`



Kompilátor

```
lw    $t0, 1200($t1)    # dočasný registr $t0 naplněn A[300]
add   $t0, $s2, $t0     # dočasný registr $t0 naplní se h + A[300]
sw    $t0, 1200($t1)     # uloží h + A[300] zpět do A[300]
```




Assembler

35	9	8	1200
0	18	8	8 0 32
43	9	8	1200

100011	01001	01000	0000 0100 1011 0000
000000	10010	01000	01000 00000 100000
101011	01001	01000	0000 0100 1011 0000

Immediates (numerické konstanty)

- Často se používají malé konstanty (50% všech operandů)
 - $A = A + 5;$
 - $C = C - 1;$
- Řešení
 - Uložíme typické konstanty do paměti a používáme je
 - Vytvoření HW registrů (např. **\$0** nebo **\$zero**)
- **Zásada_4: postavit sdílené sekce co nejrychlejší**
- Instrukce MIPS pro konstanty (formát I)
 - `addi $t0, $s7, 4` # \$t0 = \$s7 + 4



8	23	8	4
001000	10111	01000	0000 0000 0000 0100

Aritmetické přetečení

- Počítače mají omezenou délku slova (např. 32 bitů) a tím také omezenou přesnost

$$\begin{array}{r} 15 \\ + 3 \\ \hline 18 \end{array}$$
$$\begin{array}{r} 1111 \\ 0011 \\ \hline 10010 \end{array}$$


- Některé jazyky detekují přetečení (Ada), jiné nikoliv (C)
- MIPS implementuje 2 typy aritmetických instrukcí:
 - Add, sub, and addi: způsobují přetečení
 - Addu, subu, and addiu: nezpůsobují přetečení
- Kompilátory MIPS C generují implicitně instrukce addu, subu, addiu

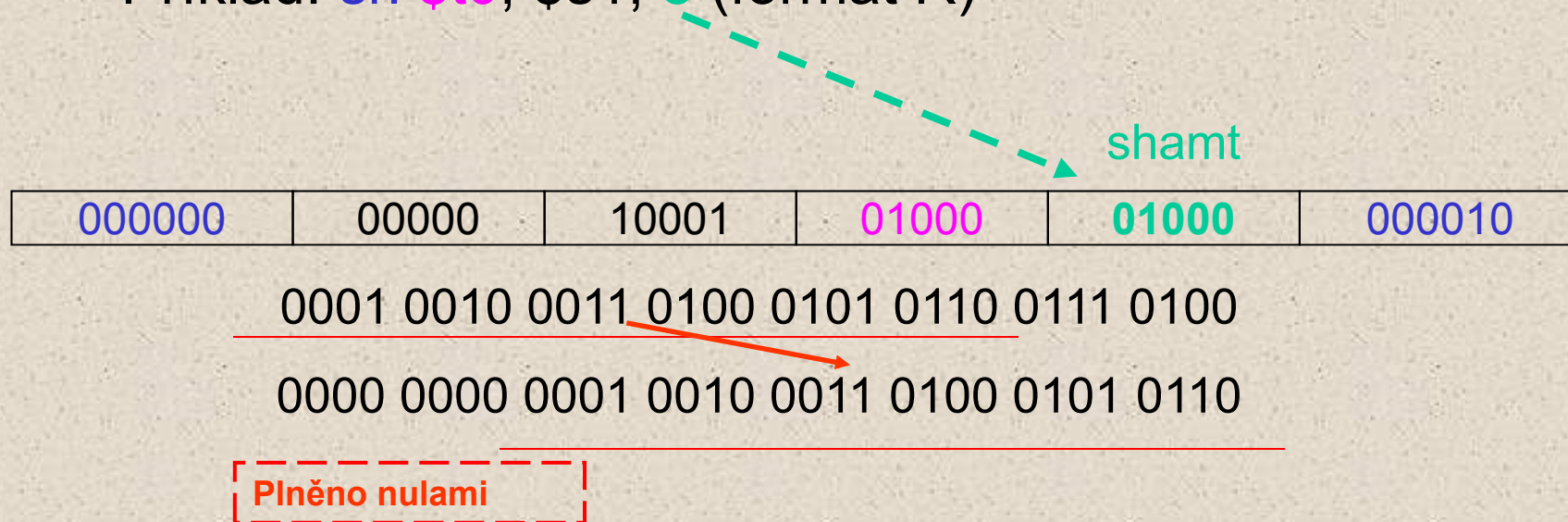
Logické instrukce

- Bitové operace
 - Obsah registrů je považován za pole o délce 32 bitů a nikoliv za jedno 32-bitové číslo
- Instrukce
 - and, or: 3 operandy jsou registry (formát R)
 - andi, ori: 3. argument je typu *immediate* (formát I)
- Příklad: maskování (andi \$t0, \$t0, 0xFFF)

1011	0110	1010	0100	0011	1101	1001	1010
0000	0000	0000	0000	0000	1111	1111	1111
<hr/>							
0000	0000	0000	0000	0000	1101	1001	1010

Instrukce pro posuv

- Posouvá všechny bity registru vlevo/vpravo
 - sll (shift left logical): uprázdněné pozice plní 0
 - srl (shift right logical): uprázdněné pozice plní 0
 - sra (shift right arithmetic): uprázdněné pozice plní znaménkem
- Příklad: srl \$t0, \$s1, 8 (formát R)



Násobení a dělení

- Používají se speciální registry (hi, lo)

- 32-bitů x 32-bitů = 64-bitů

- Mult \$s0, \$s1

000000	10000	10001	00000	00000	011000
--------	-------	-------	-------	-------	--------

- hi: horní polovina součinu

- lo: dolní polovina součinu

- Div \$s0, \$s1

000000	10000	10001	00000	00000	011010
--------	-------	-------	-------	-------	--------

- hi: zbytek (\$s0 % \$s1)

- lo: podíl (\$s0 / \$s1)

- Přesun výsledku do obecných registrů:

- mfhi \$s0

- mflo \$s1

000000	00000	00000	10000	00000	010000
--------	-------	-------	-------	-------	--------

000000	00000	00000	10001	00000	010010
--------	-------	-------	-------	-------	--------

Assembler vs. strojový jazyk

- Assembler umožňuje přehledný symbolický zápis
 - Mnohem snazší než psát samotná čísla
 - Prvý operand určen k uložení výsledku
 - Makroinstrukce
 - Návěští k identifikaci a pojmenování slov, která obsahují instrukce/data
- Strojní jazyk je základ
 - Operand pro výsledek nemusí být na prvním místě
 - Úsporný formát
- Assembler nahrazuje makroinstrukce (strojní move není!)
 - Move \$t0, \$t1 (add \$t0, \$t1, \$zero)
- Pro určení výkonu je třeba započítávat reálné instrukce

Nové – instrukce větvení programu

- Podmíněné skoky
 - If-then
 - If-then-else
- Smyčky
 - While
 - Do while
 - For
- Nerovnosti
- Přepínače

Podmíněné skoky

- Instrukce rozhodování
- Skok při rovnosti
 - **beq** register1, register2, *destination_address*
- Skok při nerovnosti
 - **bne** register1, register2, *destination_address*
- Příklad: beq \$s3, \$s4, 20

6 bitů	5 bitů	5 bitů	16 bitů
4	19	20	5
000100	10011	10100	0000 0000 0000 0101

Návěští

- Není třeba vypočítávat adresu cíle skoku

```
    if (i == j) go to L1;  
    f = g + h;  
L1:  f = f - i;
```

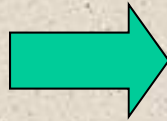
```
f => $s0  
g => $s1  
h => $s2  
i => $s3  
j => $s4
```

```
(4000) beq $s3, $s4, L1    # if i equals j go to L1  
(4004) add $s0, $s1, $s2  # f = g + h  
L1: (4008) sub $s0, $s0, $s3 # f = f - i
```

L1 odpovídá adrese instrukce odčítání

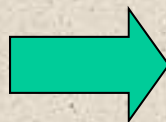
Příkaz if

```
if (condition)
    clause1;
else
    clause2;
```



```
if (condition) goto L1;
    clause2;
    goto L2;
L1: clause1;
L2:
```

```
if (i == j)
    f = g + h;
else
    f = g - h;
```



```
    beq $3, $4, True
    sub $0, $s1, $s2
    j False
True: add $s0, $s1, $s2
False:
```

Smyčky

```
Loop: g = g + A[i];  
      i = i + j;  
      if (i != h) goto Loop;
```

```
g: $s1  
h: $s2  
i: $s3  
j: $s4  
Base of A: $s5
```

```
Loop: add $t1, $s3, $s3    # $t1 = 2 * i  
      add $t1, $t1, $t1    # $t1 = 4 * i  
      add $t1, $t1, $5     # $t1=address of A[i]  
      lw  $t0, 0($t1)      # $t0 = A[i]  
      add $s1, $s1, $t0    # g = g + A[i]  
      add $s3, $s3, $s4    # i = i + j  
      bne $s3, $s2, Loop   # go to Loop if i != h
```

Základní blok

Smyčka while

```
while (save[i] == k)
    i = i + j;
```

i: \$s3; j: \$s4; k: \$s5; base of save: \$s6

```
Loop: add $t1, $s3, $s3    # $t1 = 2 * i
      add $t1, $t1, $t1    # $t1 = 4 * i
      add $t1, $t1, $s6    # $t1 = address of save[i]
      lw  $t0, 0($t1)      # $t0 = save[i]
      bne $t0, $s5, Exit   # go to Exit if save[i] != k
      add $s3, $s3, $s4    # i = i + j
      j    Loop           # go to Loop
```

Exit:

Počet provedených instrukcí if $\text{save}[i + m * j]$ does not equal k pro $m = 10$ a does equal k for $0 \leq m \leq 9$ je roven $10 * 7 + 5 = 75$

Optimalizace

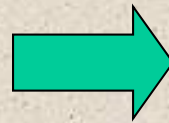
6 Instrukcí

	add \$t1, \$s3, \$s3	# Temp reg \$t1 = 2 * i
	add \$t1, \$t1, \$t1	# Temp reg \$t1 = 4 * i
	add \$t1, \$t1, \$s6	# \$t1 = address of save[i]
	lw \$t0, 0(\$t1)	# Temp reg \$t0 = save[i]
	bne \$t0, \$s5, Exit	# go to Exit if save[i] ≠ k
Loop:	add \$s3, \$s3, \$s4	# i = i + j
	add \$t1, \$s3, \$s3	# Temp reg \$t1 = 2 * i
	add \$t1, \$t1, \$t1	# Temp reg \$t1 = 4 * i
	add \$t1, \$t1, \$s6	# \$t1 = address of save[i]
	lw \$t0, 0(\$t1)	# Temp reg \$t0 = save[i]
	beq \$t0, \$s5, Loop	# go to Loop if save[i] = k
Exit:		

Počet instrukcí provedených touto novou formou smyčky je roven $5 + 10 * 6 = 65 \rightarrow \text{Úspora} = 1.15 = 75/65$. Jestliže $4 * i$ je vypočítáno před smyčkou, lze dosáhnout další úspory.

Smyčka Do-While

```
do {  
    g = g + A[i];  
    i = i + j;  
} while (i != h);
```



```
L1:  g = g + A[i];  
     i = i + j;  
     if (i != h) goto L1
```



```
g: $s1  
h: $s2  
i: $s3  
j: $s4  
Base of A: $s5
```

```
L1: sll $t1, $s3, 2      # $t1 = 4*i  
    add $t1, $t1, $s5    # $t1 = addr of A  
    lw  $t1, 0($t1)      # $t1 = A[i]  
    add $s1, $s1, $t1    # g = g + A[i]  
    add $s3, $s3, $s4    # i = i + j  
    bne $s3, $s2, L1     # go to L1 if i != h
```

- Podmíněný skok je klíčová instrukce rozhodování

Porovnání

- Programy potřebují často testovat $<$ and $>$
- Instrukce ***Set on less than***
- **slt** register1, register2, register3
 - register1 = (register2 < register3)? 1 : 0;
- Příklad: if (g < h) goto Less;

g: \$s0
h: \$s1

slt	\$t0, \$s0, \$s1
bne	\$t0, \$0, Less

- **slti**: vhodné pro smyčky if (g >= 1) goto Loop

slti	\$t0, \$s0, 1	# \$t0 = 1 if g < 1
beq	\$t0, \$0, Loop	# goto Loop if g >= 1

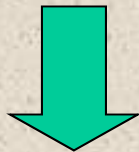
- Verze bez znaménka: **sltu** a **sltiu**

Relativní podmínky

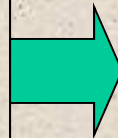
- == != < <= > >=
- MIPS přímo nepodporuje poslední čtyři
- Kompilátory používají **slt**, **beq**, **bne**, **\$zero**, and **\$at**
- Pseudoinstrukce
 - blt \$t1, \$t2, L # if (\$t1 < \$t2) go to L { slt \$at, \$t1, \$t2
bne \$at, \$zero, L
 - ble \$t1, \$t2, L # if (\$t1 <= \$t2) go to L { slt \$at, \$t2, \$t1
beq \$at, \$zero, L
 - bgt \$t1, \$t2, L # if (\$t1 > \$t2) go to L { slt \$at, \$t2, \$t1
bne \$at, \$zero, L
 - bge \$t1, \$t2, L # if (\$t1 >= \$t2) go to L { slt \$at, \$t1, \$t2
beq \$at, \$zero, L

Přepínač jazyka C

```
switch (k) {  
  case 0: f = i + j; break;  
  case 1: f = g + h; break;  
  case 2: f = g - h; break;  
  case 3: f = i - j; break;  
}
```



```
if (k==0) f = i + j;  
else if (k==1) f = g + h;  
else if (k==2) f = g - h;  
else if (k==3) f = i - j;
```



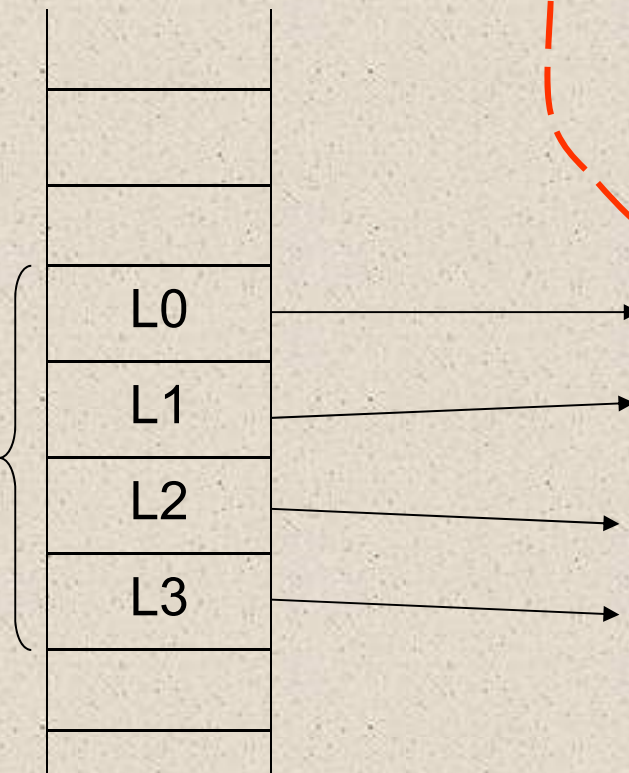
```
# f: $s0; g: $s1; h: $s2; i: $s3; j: $s4; k:$s5  
  
    bne $s5, $0, L1      # branch k != 0  
    add $s0, $s3, $s4    # f = i + j  
    j    Exit            # end of case  
L1: addi $t0, $s5, -1    # $t0 = k - 1  
    bne $t0, $0, L2      # branch k != 1  
    add $s0, $s1, $s2    # f = g + h  
    j    Exit            # end of case  
L2: addi $t0, $s5, -2    # $t0 = k - 2  
    bne $t0, $0, L3      # branch k != 2  
    sub $s0, $s1, $s2    # f = g - h  
    j    Exit            # end of case  
L3: addi $t0, $s5, -3    # $t0 = k - 3  
    bne $t0, $0, Exit    # branch k != 3  
    sub $s0, $s3, $s4    # f = i - j  
Exit:
```

Tabulky skoků

- Jump register instruction

- **jr** <register>
- nepodmíněný skok na adresu obsaženou v registru

Tabulka
skoků



```
# f: $s0; g: $s1; h: $s2; i: $s3; j: $s4; k:$s5
```

```
# $t2 = 4; $t4 = base address of JT
```

```
slt    $t3, $s5, $zero    # test k < 0
```

```
bne    $t3, $zero, Exit   # if so, exit
```

```
slt    $t3, $s5, $t2      # test k < 4
```

```
beq    $t3, $zero, Exit   # if so, exit
```

```
add    $t1, $s5, $5       # $t1 = 2*k
```

```
add    $t1, $t1, $t1      # $t1 = 4*k
```

```
add    $t1, $t1, $t4      # $t1 = &JT[k]
```

```
lw     $t0, 0($t1)        # $t0 = JT[k]
```

```
jr     $t0               # jump register
```

```
L0: add $s0, $s3, $s4     # k == 0
```

```
j      Exit              # break
```

```
L1: add $s0, $1, $s2      # k == 1
```

```
j      Exit              # break
```

```
L2: sub $s0, $s1, $s2     # k == 2
```

```
j      Exit              # break
```

```
L3: sub $s0, $s3, $s4     # k == 3
```

```
Exit:
```

Závěr

- ISA se navrhuje tak, aby dlouho přežila trendy technologie
- *V jednoduchosti je síla* (*Simpler is better* - KISS)
 - Jednoduché instrukce (omezují nedostatky CISC)
- *Malé je rychlejší*
 - Malý počet registrů nahrazuje proměnné v C
- Paměťový model
 - Adresace po bytech, zarovnání, lineární pole
- Instrukční formát MIPS – 32 bitů

Závěr (pokrač.)

- Assembler: výsledek = první operand (vlevo)
- Strojový jazyk: výsledek = poslední operand
- Tři formáty instrukcí MIPS :
 - R (aritmetické)
 - I (immediate)
 - J (skoky)
- Rozhodovací instrukce – používají *jump* (goto)
- Zlepšení výkonu – „loop unrolling“

rozvinutí smyček