Principy počítačů a operačních systémů

Instrukce – jazyk počítače

Zimní semestr 2011/2012

Jazyk počítače

Instrukce a instrukční sada

- instrukce – slova jazyka
- instrukční sada – slovník

Jaká “slova” by jazyk měl mít?

- formálně se dá ukázat, že existují instrukce, které postačují k vykonání a řízení libovolného výpočtu
- rozhodují praktické aspekty: jednoduchost realizace, rychlost vykonávání a srozumitelnost

Operace

There must certainly be instructions for performing the fundamental arithmetic operations.

– Burks, Goldstine & von Neumann, 1947

Aritmetické operace

Sčítání (odčítání)

- add a, b, c (sub a, b, c)
- symbolický zápis pro MIPS
- sečti ***b + c*** (odečti ***b - c***) a výsledek ulož do ***a***
- operace pracuje **vždy** se třemi "proměnnými"
  - 2 operandy + místo pro výsledek

Proč vždy tři operandy?

- pevný počet operandů ⇒ zjednodušení HW
- HW má být jednoduchý!

Princip návrhu: jednoduchost ⇔ pravidelnost

Příklad: překlad jednoduchých přiřazení

Fragment programu v jazyce C

```
a = b + c;
d = a – e;
```

Překlad do assembleru pro MIPS

- 5 proměnných **a, b, c, d, e**
- výrazy odpovídají možnostem instrukcí
- jednoduché příkazy překladač přeloží přímo na jednotlivé instrukce

```
add a, b, c
sub d, a, e
```

Příklad: překlad složitějších přiřazení

Fragment programu v jazyce C

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

Překlad do assembleru pro MIPS

- 5 proměnných **f, g, h, i, j**
- instrukce pracují pouze se třemi operandy
- překladač musí rozdělit příkaz do více instrukcí a uložit mezivýsledky do pomocných "proměnných" ***t0*** a ***t1***

```
add t0, g, h    # t0 = g + h
add t1, i, j    # t1 = i + j
sub f, t0, t1   # f = (g + h) - (i + j)
```

Operandy

Registry

Registr = paměťové místo v procesoru

- programově přístupné, omezený počet (1, 8, 16, 32)
- velikost = nativní délka slova (MIPS 32 bitů)

Princip návrhu: menší ⇔ rychlejší

- neplatí absolutně, 31 není nutně rychlejší než 32
- důležité je udržet vysokou rychlost přístupu
- vyšší počet registrů je nutné kódovat více bity

Označení registrů

- čísla v inst. kódu, symbolická jména v assembleru
  - MIPS - $zero, $a0-a3, $s0-$s7, $t0-$t7, $v0-$v1, ..., $gp, $ra, $sp
  - x86 – eax, ebx, ecx, edx, esi, edi, ebp, esp

Příklad: přiřazení s použitím registrů

Fragment programu v jazyce C

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

Překlad do assembleru pro MIPS

- překladač musí “nějak” asociovat proměnné ***f***, ***g***, ***h***, ***i*** a ***j*** s registry **$s0, $s1, $s2, $s3** a **$s4**
- překladač musí rozdělit příkaz do více instrukcí a použít pomocné registry **$t0** a **$t1** pro mezivýsledky

```
add $t0, $s0, $s1   # t0 = g + h
add $t1, $s2, $s3   # t1 = i + j
sub $s4, $t0, $t1   # s4 = (g + h) - (i +j)
```

Paměťové operandy

Práce s daty v programovacích jazycích

- jednoduché datové elementy – proměnné
- složitější datové struktury – pole, struktury, třídy
  - datových elementů je **výrazně** více než registrů v procesoru ⇒ data jsou primárně uložena v operační paměti

… ale aritmetické operace pracují jen s registry

- potřebujeme přesouvat data z paměti do registrů
  - instrukce pro přesun dat (*data transfer instructions*)
- instrukce musí poskytnout adresu **slova** v paměti

Logický model paměti

Velké jednorozměrné pole

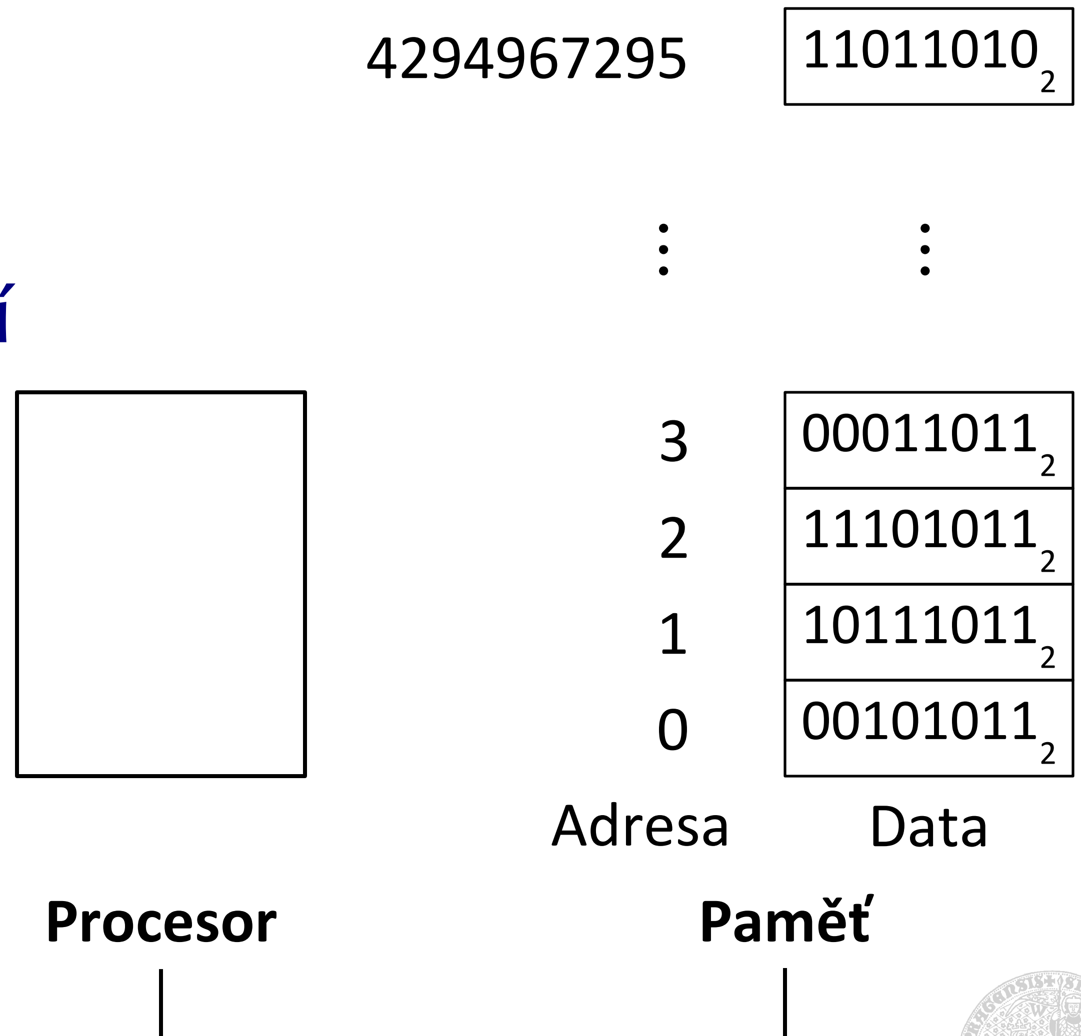
- posloupnost položek stejné velikosti
  - 8-bitů = 1 bajt (*byte*)

Adresa odpovídá pořadí

- adresa = index v poli

Paměť obsahuje čísla

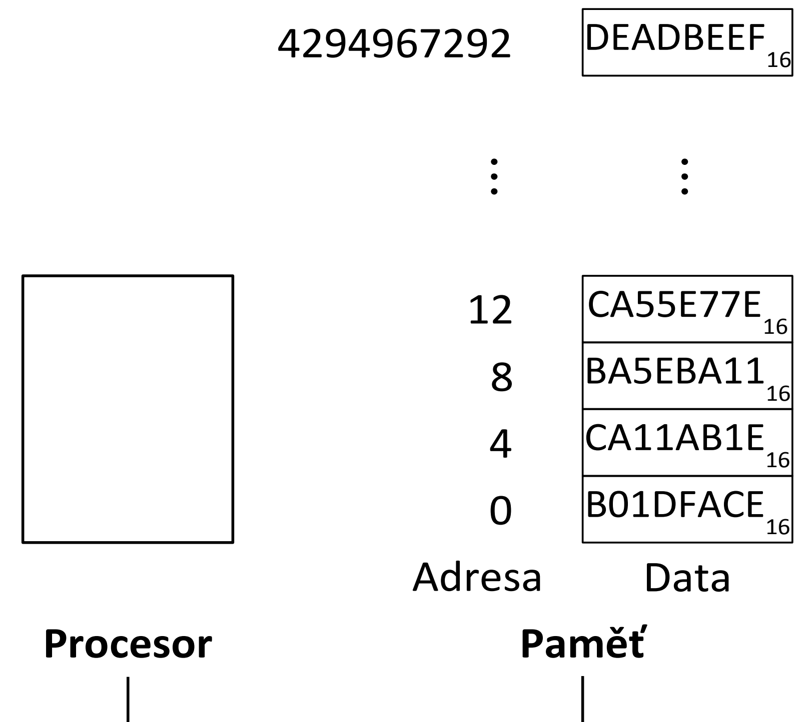
- instrukce, data
- význam určen interpretací



Přesun dat mezi pamětí a registry

Instrukce **load** a **store**

- jméno operace
- cílový/zdrojový registr
- adresa v paměti



Specificky pro MIPS

- **lw** (load word), **sw** (store word)
  - adresa = konstanta + registr
- přečte z/zapíše do paměti slovo (32-bitů)
  - v paměti uloženo ve 4 po sobě jdoucích bajtech
- adresy slov musí být dělitelné 4

Ukládání vícebajtových posloupností

Slova reprezentovaná více bajty

- **DEADBEEF**$_{16}$ ~ 32-bitové slovo ~ 4 bajty
- uloženo po bajtech na adresách ***A*, *A+1*, *A+2*, *A+3***

Big Endian (MIPS, Motorola, ...)

- s rostoucí adresou klesá význam bajtů
- [A]=**DE**$_{16}$, [A+1]=**AD**$_{16}$, [A+2]=**BE**$_{16}$, [A+3]=**EF**$_{16}$

Little Endian (Intel x86, MIPS, ...)

- s rostoucí adresou roste význam bajtů
- [A]=**EF**$_{16}$, [A+1]=**BE**$_{16}$, [A+2]=**AD**$_{16}$, [A+3]=**DE**$_{16}$

Důsledky?

Příklad: přiřazení s paměťovým operandem

Fragment programu v jazyce C

```
int A [100];
g = h + A [7];
```

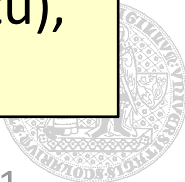
Proměnná **A** reprezentuje pole 100 slov. Prvky pole ***A*** mají indexy od **0** do **99**. **A [7]** reprezentuje 8. prvek pole.

Překlad do assembleru pro MIPS

- překladač asocioval proměnné ***g***, a ***h*** s registry **$s1** a **$s2**
- překladač umístil pole **A** do paměti a jeho počáteční (***bázovou***) adresu uložil do registru **$s3**
- obsah ***A*** [7] nutno načíst do registru, adresa dána součtem ***bázové adresy*** **A** a ***offsetu*** 8. prvku pole

```
lw  $t0, 28 ($s3)
add $s1, $s2, $t0
```

$s3 představuje ***bázový registr***, číslo 28 je ***offset*** 8. prvku (slova o šířce 32 bitů), přičemž ***offset = index × velikost***



Příklad: přiřazení se dvěma paměťovými operandy

Fragment programu v jazyce C

```
int A [100];
A [12] = h + A [7];
```

Překlad do assembleru pro MIPS

- překladač asocioval proměnnou ***h*** s registrem **$s2** a počáteční adresu ***A*** do uložil do registru **$s3**
- obsah ***A*** [7] nutno načíst do registru, výsledek operace nutno uložit z registru do ***A*** [12]

```
lw  $t0, 28 ($s3)   # t0 = A [7]
add $t0, $s2, $t0   # t0 = h + A [7]
sw  $t0, 48 ($s3)   # A [12] = h + A [7]
```

Jak používá překladač registry?

Práce s registry je rychlejší než práce s pamětí

- menší = rychlejší

Data v registrech jsou užitečnější než v paměti

- aritmetické operace pracují s dvěma registry
- instrukce load/store pouze čtou/zapisují

Proměnných je výrazně více než registrů

- programátor/překladač musí registry využívat efektivně
  - nejčastěji používané proměnné v registrech
  - ostatní proměnné přesouvány mezi pamětí a registry podle potřeby – ***register spilling***
  - “nejčastěji” platí pro omezený kontext (funkce, cyklus, ...)

Přímé operandy (konstanty)

K čemu se hodí konstanty?

- zvýšení indexu do pole aby odkazoval na následující prvek, řídící proměnné cyklu, ...
- stávající instrukce umožňují pouze použití konstant uložených v paměti – např. přičtení 4 k registru $s3

```
lw  $t0, OffsetConst4 ($s1)
add $s3, $s3, $t0
```

$s1 obsahuje adresu tabulky (pole) konstant

- abychom se vyhnuli čtení z paměti, potřebujeme možnost zadat konstanty přímo

```
addi $s3, $s3, 4
```

Pozn.: instrukce ***lw*** a ***sw*** již přímý operand podporují.

Princip návrhu: nejčastější ⇔ nejrychlejší

- častý výskyt ⇒ podpora v instrukcích (MIPS $zero = 0)

Reprezentace čísel v počítači

Bez znaménka (přirozená čísla) a se znaménkem (celá čísla)

Jak počítač reprezentuje čísla?

Člověk vs. počítač

- číslům samotným je to úplně jedno, ta prostě jsou
- člověk má 10 prstů ⇒ desítková soustava
  - číslo tvořeno posloupností číslic **0, 1, …, 9**
- počítač rozlišuje 2 hodnoty ⇒ dvojková soustava
  - číslo tvořeno posloupností číslic **0** a **1**

HW reprezentace čísla

- 1 číslice = 1 bit (*binary digit*)
- rozdílné úrovně napětí pro **2** hodnoty (**0** a **1**)
  - nejmenší jednotka informace (true/false, on/off, high/low, ...)
- veškeré informace složeny z bitů
  - posloupnost číslic = posloupnost elektronických signálů

Hodnota čísla v pozičním systému o základu β

Hodnota *n*-ciferného celého čísla *A*

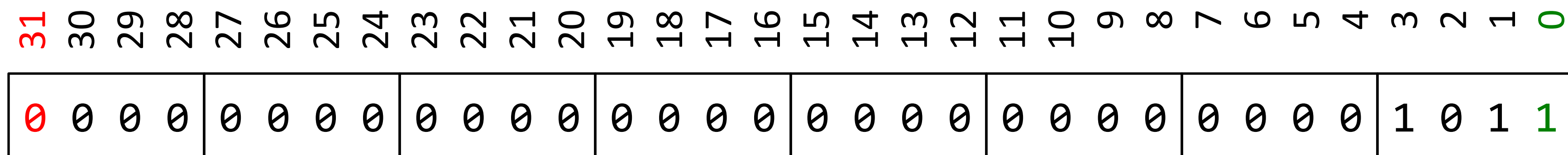
- číslice na ***i***-té pozici reprezentována koeficientem a_i

$$A = \sum_{i=0}^{i=n-1} a_i \beta^i$$

$$\begin{aligned} 1011_2 &= (1\times 2^3) + (0\times 2^2) + (1\times 2^1) + (1\times 2^0)_{10} \\ &= (1\times 8) + (0\times 4) + (1\times 2) + (1\times 1)_{10} \\ &= 8 + 0 + 2 + 1_{10} = 11_{10} \end{aligned}$$

Zápis čísla v *n*-bitovém slově (β=2)

- bity očíslovány zprava doleva
  - bit na pozici ***i=0*** nejméně významný (least significant bit, LSB)
  - bit na pozici ***i=n-1*** nejvýznamnější (most significant bit, MSB)



Co lze vyjádřit n-bitovým slovem?

2^n různých kombinací jedniček a nul

- reprezentace přirozených čísel od ***0*** do ***2^n-1***

```
0000 0000 0000 0000 0000 0000 0000 0000₂ = 0₁₀
0000 0000 0000 0000 0000 0000 0000 0001₂ = 1₁₀
0000 0000 0000 0000 0000 0000 0000 0010₂ = 2₁₀
                    ...
1111 1111 1111 1111 1111 1111 1111 1101₂ = 4 294 967 293₁₀
1111 1111 1111 1111 1111 1111 1111 1110₂ = 4 294 967 294₁₀
1111 1111 1111 1111 1111 1111 1111 1111₂ = 4 294 967 295₁₀
```

$$A=(a_{31}\times 2^{31})+(a_{30}\times 2^{30})+(a_{29}\times 2^{29})+\ldots+(a_1\times 2^1)+(a_0\times 2^0)$$

- kombinace bitů čísla pouze ***reprezentují***
  - čísla mají nekonečně mnoho číslic (počáteční nuly)
  - není-li možné reprezentovat výsledek ⇒ ***přetečení***

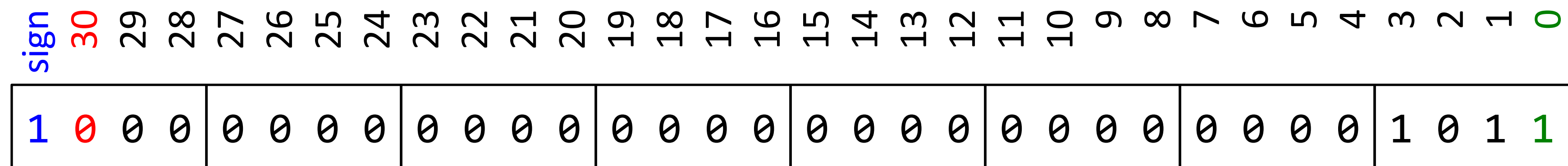
Jak reprezentovat kladná a záporná čísla?

Co třeba přidat znaménko?

- na reprezentaci znaménka stačí 1 bit

Reprezentace se znam. bitem (*sign & magnitude*)

- reprezentace celých čísel od ***-(2^n-1^-1)*** do ***2^n-1^-1***
- nejvyšší bit ***reprezentuje*** znaménko (znaménkový bit)
  - inverze znaménkového bitu mění znaménko čísla
- přímočaré, ale problematické řešení
  - HW operace musí nastavovat znaménko výsledku
  - dvojí (kladná a záporná) reprezentace nuly



$$A=(-1^{a_{31}})\times((a_{30}\times 2^{30})+(a_{29}\times 2^{29})+\ldots+(a_1\times 2^1)+(a_0\times 2^0))$$

Co by se líbilo hardwaru?

Chceme jednoduchý HW...

- Co se stane, pokud od menšího přirozeného čísla odečteme větší přirozené číslo?

$1_{10} - 3_{10} = 1_2 - 11_2 = ?$

- dojde k výpůjčce z vyšších řádů (nul před číslem), takže výsledek bude začínat řetězcem jedniček...

$1_2 - 11_2 = 000...0001_2 - 000...00011_2 = 111...1110_2$

Co na to HW?

- kladné číslo začíná nulami, záporné jedničkami
  - počet bitů reprezentace je omezen šířkou slova

$0001_2 - 0011_2 = 1110_2$

- HW operace s čísly se nemusí ohlížet na znaménko

Nejpoužívanější reprezentace celých čísel

Reprez. ve dvojkovém doplňku (*two's complement*)

- reprezentace celých čísel od -2^{n-1} do $2^{n-1}-1$
- nejvyšší (znaménkový) bit ***indikuje*** znaménko (*sign bit*)

```
0000 0000 0000 0000 0000 0000 0000 0000₂ = 0₁₀
0000 0000 0000 0000 0000 0000 0000 0001₂ = 1₁₀
                  ...
0111 1111 1111 1111 1111 1111 1111 1111₂ = 2 147 483 647₁₀
1000 0000 0000 0000 0000 0000 0000 0000₂ = -2 147 483 648₁₀
1000 0000 0000 0000 0000 0000 0000 0001₂ = -2 147 483 647₁₀
                  ...
1111 1111 1111 1111 1111 1111 1111 1110₂ = -2₁₀
1111 1111 1111 1111 1111 1111 1111 1111₂ = -1₁₀
```

$$A=(a_{31}\times -2^{31})+(a_{30}\times 2^{30})+(a_{29}\times 2^{29})+\ldots+(a_1\times 2^1)+(a_0\times 2^0)$$

- znaménkový bit ≠ bity vlevo od znaménka ⇒ přetečení

$011_2 + 001_2 = 000...000100_2$

Záporná čísla ve dvojkovém doplňku

Příklad: zjistěte hodnotu čísla

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2 = ?_{10}$

$$(1\times -2^{31})+(1\times 2^{30})+(1\times 2^{29})+\ldots+(1\times 2^{2})+(0\times 2^{1})+(0\times 2^{0})=$$
$$=-2^{31}+2^{30}+2^{29}+\ldots+2^{2}+0+0=-2147483648_{10}+2147483644_{10}=-4_{10}$$

Negace *n*-bitového čísla (doplněk do 2^n)

- rychlá negace: inverze bitů (0 → 1, 1 → 0) a přičtení 1
  - $A + \overline{A} = 111\ldots111_2 = -1_{10}$, tedy $A + \overline{A} + 1 = 0$, resp. $\overline{A} + 1 = -A$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2 = ?_{10}$

$-(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0011_2 + 1_2) = -(?)_{10}$

$-(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0100_2) = -(4)_{10}$

- převod z desítkové soustavy do dvojkové analogicky
- Proč doplněk? Pro *n*-bitové číslo ***A*** platí $-A + A = 2^n$
  - tedy $-A = 2^n - A$ (číslo opačné k ***A*** odpovídá jeho doplňku do 2^n)

Znaménkové rozšíření ve dvojkovém doplňku

Převod *n*-bitového na *m*-bitové číslo, *n* < *m*

- registry mají pevnou šířku, data v paměti mohou být "užší" ⇒ při načtení do registru nutno rozšířit

mem: $0000\ 0000\ 0000\ 1000_2 = 8_{10}$

reg: $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1000_2 = 8_{10}$

mem: $1111\ 1111\ 1111\ 1000_2 = -8_{10}$

reg: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1000_2 = -8_{10}$

- zkopírovat znaménkový bit do chybějících bitů vlevo
  - kladné (záporné) číslo ve dvojkovém doplňku má nekonečně mnoho nul (jedniček) nalevo od znaménkového bitu
  - *n*-bitové slovo je omezená reprezentace čísla, při rozšíření na *m* bitů je nutné bity nalevo od znaménkového bitu obnovit

Jiné reprezentace

Reprezentace s posunutím nuly (*biased notation*)

- nula je reprezentována konstantou
  - min záporné číslo $00\ldots00_2$, max kladné číslo $11\ldots11_2$
- reprezentace celých čísel (typicky) od -2^{n-1} do $2^{n-1}-1$
- používá se u čísel v plovoucí řádové čárce (později)

Reprez. v jedničkovém doplňku (*one's complement*)

- reprezentace celých čísel od $-(2^{n-1}-1)$ do $2^{n-1}-1$
- nejvyšší (znaménkový) bit ***indikuje*** znaménko (*sign bit*)
- negace čísla je prostá inverze bitů, tedy $-A = \overline{A}$
- předchůdce dvojkového doplňku, nepoužívá se
  - operace odčítání vyžaduje krok navíc

Reprezentace instrukcí v počítači

Reprezentace instrukcí v počítači

Instrukce = posloupnost 0 a 1

- číselná reprezentace – strojový kód
  - složen z několika čísel odpovídajících různým částem instrukce (specifikace operace a operandů)
- operace a registry reprezentovány čísly
  - symbolická jména operací pouze v assembleru
  - $s0-s7 ⇒ 16-23, $t0-t7 ⇒ 8-15, $a0-a4 ⇒ … (MIPS)
  - add ⇒ 0, 32 (MIPS)

Příklad: překlad instrukce do strojového kódu (MIPS)

```
add $t0, $s1, $s2
0 | 17 | 18 | 8 | 0 | 32
000000 | 10001 | 10010 | 01000 | 00000 | 100000
```

Jak si zjednodušit práci s dvojkovými čísly?

Potřebujeme kratší zápis a rychlý převod...

- osmičková/oktalová soustava (β = 8)

000_2	001_2	010_2	011_2	100_2	101_2	110_2	111_2
0_8	1_8	2_8	3_8	4_8	5_8	6_8	7_8

$00\ 010\ 011\ 010\ 101\ 111\ 001\ 101\ 111\ 011\ 111_2 = 02\ 325\ 715\ 737_8$

- šestnáctková/hexadecimální soustava (β = 16)

0000_2	0001_2	0010_2	0011_2	0100_2	0101_2	0110_2	0111_2
0_{16}	1_{16}	2_{16}	3_{16}	4_{16}	5_{16}	6_{16}	7_{16}
1000_2	1001_2	1010_2	1011_2	1100_2	1101_2	1110_2	1111_2
8_{16}	9_{16}	A_{16}	B_{16}	C_{16}	D_{16}	E_{16}	F_{16}

$0001\ 0011\ 0101\ 0111\ 1001\ 1011\ 1101\ 1111_2 = 1357\ 9BDF_{16}$

Základní formát instrukčního kódu (MIPS)

Instrukce registrového typu (*r-type*)

op	rs	rt	rd	shamt	funct
6 bitů	5 bitů	5 bitů	5 bitů	5 bitů	6 bitů

- ***op***: operační kód – základní operace (*opcode*)
- ***rs***: první zdrojový registrový operand
- ***rt***: druhý zdrojový registrový operand
- ***rd***: cílový registrový operand
- ***shamt***: posun v logických operacích (*shift amount*)
- ***funct***: funkce – varianta operace (*function code*)

Co když potřebujeme v instrukci delší pole?

- konstanta v load/store a add immediate instrukcích

Formát pro přímé operandy (MIPS)

Instrukce s přímým operandem (*i-type*)

op	rs	rt	address or constant
6 bitů	5 bitů	5 bitů	16 bitů

- ***op***: operační kód – základní operace (*opcode*)
- ***rs***: zdrojový registrový operand
- ***rt***: cílový registrový operand
- do zbytku je možné zapsat 16b konstantu

Princip návrhu: dobrý návrh ⇔ dobré kompromisy

- konstantní délka instrukce vs. stejný formát instrukce
  - stejná délka ⇒ různý formát pro různé typy operací
  - snažíme se o co nejpodobnější formáty

Příklad: přiřazení se dvěma paměťovými operandy

Fragment programu v jazyce C

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

Překlad do assembleru pro MIPS

```
lw  $t0, 1200 ($t1)  # t0 = A [300]
add $t0, $s2, $t0    # t0 = h + A [300]
sw  $t0, 1200 ($t1)  # A [300] = h + A [300]
```

Strojový kód pro MIPS

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

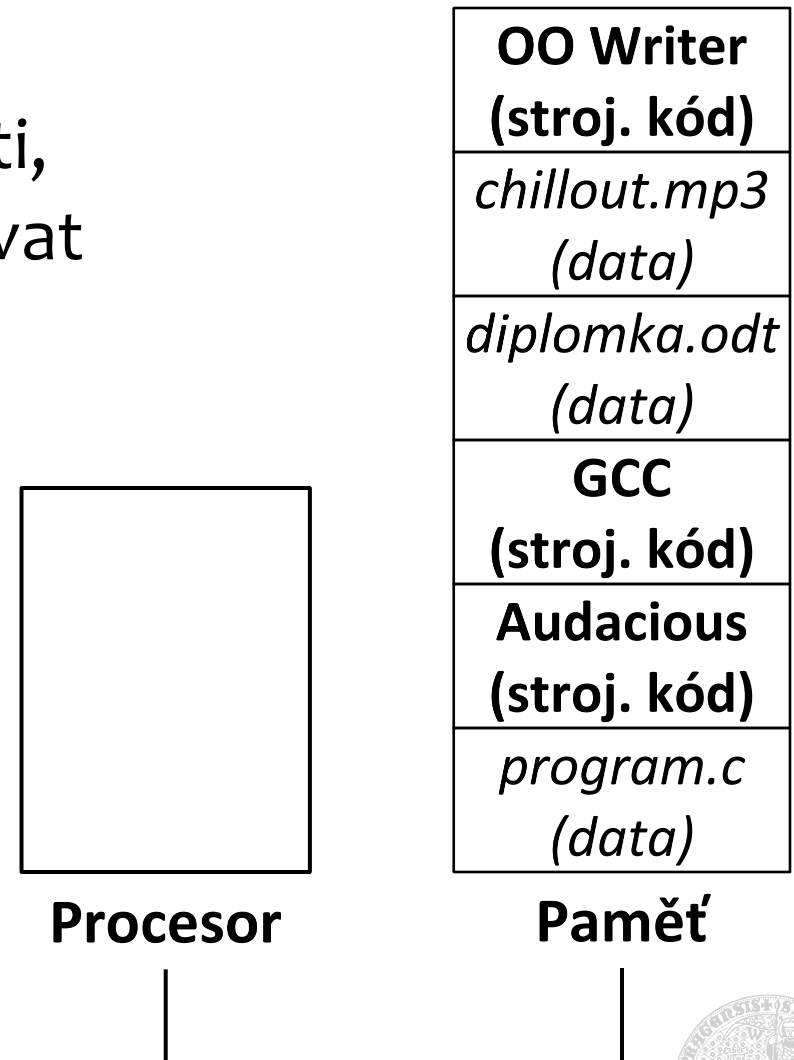
Počítač s vnitřním řízením (*stored-program computer*)

Hlavní principy

1. instrukce reprezentovány čísly
2. programy jsou uloženy v paměti, odkud/kam se dají číst a zapisovat jako čísla

Důsledek

- software je možné šířit ve formě binárních souborů
- funkce počítače závisí pouze na tom, jaký program zrovna vykonává



Logické operace

K čemu jsou potřeba logické operace?

Manipulace s bity v rámci slov

- procesor pracuje s celými slovy, data mohou mít granularitu bajtů, či (skupin) bitů ve slovech
  - pole v instrukčním kódu, znaky textového dokumentu, stavové slovo z V/V zařízení, ...
- instrukce pro logické operace usnadňují mj. "balení" a "vybalování" bitů ze slov

Typické operace

- logické posuny vlevo/vpravo
  - operátory **<<** a **>>** v jazycích C a Java
- logický součin/součet/negace/... po bitech (*bitwise*)
  - operátory **&**, **|**, **~**, ... v jazycích C a Java

Logické posuny

Posouvají všechny bity registru vlevo/vpravo

- "uvolněné" bity jsou nahrazeny 0
- instrukce **sll**, **srl** (*shift left/right logical*)
  - délka posunu určena polem ***shamt*** v r-type instrukci
- posun o ***i*** vlevo/vpravo odpovídá násobení/dělení $\mathbf{2}^{i}$
  - pouze pro přirozená čísla

Příklad: posun vlevo o 4

```
s0 = 0000 0000 0000 0000 0000 0000 0000 1001₂ = 9₁₀
sll $t2, $s0, 4   # t2 = s0 << 4
t2 = 0000 0000 0000 0000 0000 0000 1001 0000₂ = 144₁₀
```

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

Logické operace po bitech

Logický AND, OR, XOR, NOR, NOT, … po bitech

- logická funkce postupně aplikována na jednotlivé bity z operandů a výsledek uložen do cílového registru
- umožňují izolovat (AND), vynulovat (AND, NOR), nastavit (OR), invertovat (XOR) vybrané bity slova, případně invertovat celé slovo (NOT)

Příklad: izolace, nastavení a inverze bitů

```
t2 = 0000 0000 0000 0000 0000 1101 1100 0000₂
t1 = 0000 0000 0000 0000 0011 1100 0000 0000₂
and $t0, $t1, $t2     # t0 = t1 & t2
t0 = 0000 0000 0000 0000 0000 1100 0000 0000₂
or $t0, $t1, $t2      # t0 = t1 | t2
t0 = 0000 0000 0000 0000 0011 1101 1100 0000₂
nor $t0, $t1, $zero   # t0 = ~(t1 | $zero) = ~t1
t0 = 1111 1111 1111 1111 1100 0010 0011 1111₂
```

Příklad: zjištění hodnoty bitového pole

Fragment programu v jazyce C

```
shamt = (rtype & 000007C0₁₆) >> 6;
```

Překlad do assembleru pro MIPS

```
andi $t0, $s1, 000007C0₁₆
srl  $s0, $t0, 6
```

Strojový kód pro MIPS

<table>
<tr><td>001100</td><td>10001</td><td>01000</td><td colspan="3">0000 0111 1100 0000</td></tr>
<tr><td>000000</td><td>00000</td><td>01000</td><td>10000</td><td>00110</td><td>000010</td></tr>
</table>

Rozhodovací operace

Jak se liší počítač od kalkulátoru?

Může rozhodovat na základě dat

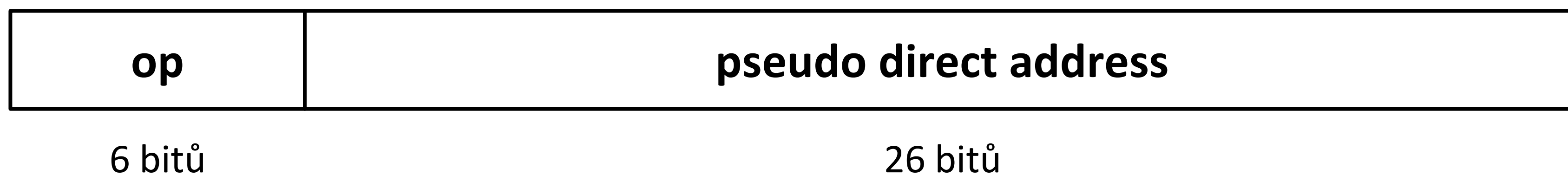
- vstupní a dosud vypočtené hodnoty ovlivňují, které instrukce se budou provádět v dalších krocích
- řídící příkazy v programovacích jazycích
  - ***if-then-else***, ***goto***, ***switch-case***, ***for*** a ***while***

Instrukce pro podporu rozhodování

- podmíněné a nepodmíněné skoky
  - beq, bne (branch if [not] equal)
  - j, jr (jump [register])
- porovnávání
  - slt, slti (set on less than [immediate])

Formát pro přímé nepodmíněné skoky (MIPS)

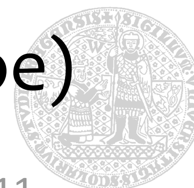
Instrukce přímých nepodmíněných skoků (*j-type*)



- ***op***: operační kód – základní operace (*opcode*)
- přímá adresa skoku
  - adresa slova (bez spodních 2 bitů), horní 4 bity z PC
  - nelze přímo skákat přes hranici 256MiB

Ostatní typy skoků

- nepřímý skok na adresu v registru (formát r-type)
- větvení při rovnosti/nerovnosti registrů (formát i-type)



Příklad: překlad větvení if-then-else

Fragment programu v jazyce C

```
if (i == j) {
     f = g + h;
} else {
    f = g – h;
}
```

překladač asocioval proměnné ***f***, ***g***, ***h*** a ***i***, ***j*** s registry ***$s0***, ***$s1***, ***$s2*** a ***$s3***, ***$s4***

Překlad do assembleru pro MIPS

- větve programu oddělil překladač návěštími

```
     bne  $s3, $s4, Else   # i ≠ j ⇒ goto Else
     add  $s0, $s1, $s2    # f = g + h
     j    Exit             # goto Exit
Else:
     sub  $s0, $s1, $s2    # f = g – h
Exit:
```

Příklad: cyklus while

Fragment programu v jazyce C

```
while (save [i] == k) {
    i += 1;
}
```

překladač asocioval proměnné ***save*** a ***i***, ***k*** s registry ***$s0*** (báze) a ***$s1***, ***$s2***

Překlad do assembleru pro MIPS

```
Loop:
    sll  $t1, $s1, 2      # t1 = i << 2 = i * 4
    add  $t1, $t1, $s0    # t1 = & save [i]
    lw   $t0, 0 ($t1)     # t0 = save [i]
    bne  $t0, $s2, Exit   # t0 ≠ k ⇒ goto Exit
    addi $s1, $s1, 1      # i = i + 1
    j    Loop             # goto Loop
Exit:
```

Příklad: větvení switch-case

Více větví v jednom místě

- přeloženo jako kaskáda ***if-then-else***
- rozskok podle tabulky
  - načtení cílové adresy do registru + instrukce **jr**

Překlad do assembleru pro MIPS

```
sll  $t1, $s1, 2     # t1 = i * 4
add  $t2, $s0, $s1   # t2 = & tabulka [i]
lw   $t0, 0 ($t2)    # t0 = tabulka [i]
jr   $t0             # skok na adresu v t0
```

Volání procedur a funkcí

Volání procedur a funkcí

Procedury a funkce

- možno volat odkudkoliv, návrat do místa volání
- lokální kontext – vstupní a výstupní parametry

Průběh volání

1. uložení vstupních parametrů pro funkci
2. předání řízení do kódu funkce – skok
3. získání prostředků pro vykonání funkce
4. provedení požadované funkce
5. uložení výsledků pro volajícího
6. návrat do místa volání

Volání procedur a funkcí

Kam uložit data?

- pokud možno do registrů

Podpora pro volání na MIPS

- registry pro data
  - \$a0, \$a1, \$a2, \$a3 – vstupní parametry funkce
  - \$v0, \$v1 – návratová hodnota funkce
  - \$ra – registr pro návratovou adresu
- skok s uložením návratové adresy do \$ra
  - adresa instrukce následující po instrukci volání (PC + 4)
  - jal (jump and link)

Volání procedur a funkcí

Co když potřebujeme místo v registrech?

- rekurzivní volání, mnoho parametrů
- registry nutno “vylít” do paměti – zásobník

Zásobník – Last In First Out

- ukazatel na vrchol zásobníku, operace push a pop
- zásobník roste směrem k nižším adresám
  - snížení/zvýšení vrcholu vytvoří/ubere místo na zásobníku

Podpora pro zásobník na MIPS

- $sp – ukazatel vrcholu zásobníku
- přístup do paměti relativně k vrcholu zásobníku

Příklad: překlad funkce (1)

Fragment programu v jazyce C

- funkce příjmá 4 parametry
- funkce má 1 lokální proměnnou
- funkce vrací 1 hodnotu

```
int leaf (int g, int h, int i, int j) {
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

Příklad: překlad funkce (2)

Vstup do funkce

- rezervace místa na zásobníku
- uložení obsahu potřebných registrů na zásobník

Překlad do assembleru pro MIPS

```
leaf:
    addi $sp, $sp, -12
    sw   $t1, 8 ($sp)
    sw   $t0, 4 ($sp)
    sw   $s0, 0 ($sp)
```

Zásobník při volání procedury/funkce

$sp

$sp

obsah $t1

obsah $t0

obsah $s0

$sp

před
voláním

během
volání

po
volání

Příklad: překlad funkce (3)

Tělo funkce

- výpočet hodnoty funkce ze zadaných parametrů
- uložení výsledku do registru pro volajícího

Překlad do assembleru pro MIPS

```
add  $t0, $a0, $a1      # t0 = g + h
add  $t1, $a2, $a3      # t1 = i + j
sub  $s0, $t0, $t1      # s0 = (g + h) – (i + j)
add  $v0, $s0, $zero    # v0 = s0 = f
```

Příklad: překlad funkce (4)

Návrat z funkce

- obnova obsahu registrů ze zásobníku
- zrušení místa vyhrazeného na zásobníku
- návrat do místa volání

Překlad do assembleru pro MIPS

```
lw   $s0, 0 ($sp)
lw   $t0, 4 ($sp)
lw   $t1, 8 ($sp)
addi $sp, $sp, 12
jr   $ra
```

Příklad: volání funkce

Fragment programu v jazyce C

```
int a, b, c, d, e;
a = leaf (b, c, d, e);
```

Překlad do assembleru pro MIPS

```
lw   $a0, 4 ($sp)    # načtení parametrů z proměnných
lw   $a1, 8 ($sp)    # b, c, d, a e na zásobníku do
lw   $a2, 12 ($sp)   # registrů a0, a1, a2 a a3
lw   $a3, 16 ($sp)
jal  leaf            # skok s uložením návratové adresy
sw   $v0, 0 ($sp)    # uložení výsledku z registru v0
                     # do proměnné a na zásobníku
```

Struktura dat na zásobníku

$fp
$sp

před
voláním

$fp
$sp

uschované
reg. $a0-3

návratová
adresa, $ra

uschované
registry

lokální
proměnné

během
volání

$fp
$sp

po
volání

Komunikace s lidmi

Čísla nestačí, potřebujeme (alespoň) text...

Jak reprezentovat text v počítači?

- text je posloupnost znaků
- znaky zakódujeme do čísel (omezený počet)
- posloupnost čísel uložíme do paměti

Kolik čísel potřebujeme?

- záleží na tom, kolik znaků potřebujeme reprezentovat
  - pro “běžné” jazyky stačí 8-bitů na znak - 1 bajt
  - pro jiné jazyky potřebujeme 16 i více bitů

Jaká čísla si máme zvolit?

- měla by jim rozumět většina programů
  - nutná shoda ⇒ standardizace

Kódování znaků podle ASCII

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Zdroj: P&H

Kódování znaků pomocí Unicode

Národní abecedy

- Latin, Greek, Cyrillic, …
- Arabic, Hebrew, …
- Katakana, Hiragana, …
- Cherokee, Phoenician, …

	xx0	xx1	xx2	xx3	xx4	xx5	xx6	xx7
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								
A								
B								
C								
D								
E								
F								

Zdroj: dkuug.dk

Reprezentace

- UTF-8, UTF-16
  - proměnný počet bajtů/znak
- UTF-32
  - 4 bajty/znak

Jak se pracuje s textem?

Znaky sdruženy do řetězců (*strings*)

- posloupnost znaků v paměti (v určitém kódování)

  ```
  “pocitac” = 70 6F 63 69 74 61 63 (ASCII)
  “počítač” = 70 6F C4 8D C3 AD 74 61 C4 8D (UTF-8)
  ```

- koncept poskytovaný programovacím jazykem
  - liší se v implicitním kódování a uchování délky řetězce

Hlavní způsoby reprezentace řetězců

(1) první pozice v řetězci vyhrazena pro délku

(2) délka je součástí přidružená proměnné/struktury

(3) řetězec je ukončen speciálním znakem (kód 0, *nul*)

Jsou potřeba speciální instrukce?

V principu ne, ale práce s textem je velmi častá...

- místo se slovy se často pracuje s bajty – při použití běžných instrukcí může být práce málo efektivní

Instrukce pro usnadnění/zefektivnění práce

- práce s paměti s menší granularitou
  - znaménkové rozšíření načtených dat

Specificky pro MIPS

- load byte/byte unsigned/half word/upper half word
- store byte/half word/upper half word

Bludy a pasti

Bludy

Mocnější instrukce vedou k vyššímu výkonu

- viz případ Intelu a jeho řetězcových instrukcí na kopírování bloků paměti

Nejvyšší výkon získáme jen psaním v assembleru

- problémy s přenositelností, srozumitelností, udržovatelností, produktivitou, ...

Kvůli důležitosti binární kompatibility se úspěšné instrukční sady nemění

- zpětná kompatibility je sice svatá, ale nové instrukce se objevují
- viz Intel, 400 nových instrukcí od r. 2005

Pasti(čky)

Adresy sousedních slov v paměti se neliší o 1

- paměti jsou většinou adresovány po bajtech

Bity nejsou bajty

- počet bitů na reprezentaci čísla není počet bajtů, které číslo zabere v paměti

Shrnutí

Počítač s vnitřním řízením

Základní princip

- Instrukce jsou zakódované čísly
  - v principu nedají se rozlišit od dat
- program je uložen v modifikovatelné paměti
  - účel počítače určuje právě vykonávaný program

Návrh instrukční sady

- rovnováha mezi počtem instrukcí na program, počtem taktů na instrukci a taktovací frekvencí
  - Jednoduchost podporuje pravidelnost
  - Menší znamená rychlejší
  - Optimalizace pro běžné situace
  - Dobrý návrh vyžaduje dobré kompromisy

Instrukce poskytují podporu pro vyšší jazyky

Různé třídy instrukcí

- aritmetické instrukce realizují výrazy v podmínkách a přiřazovacích příkazech
- instrukce pro přenos dat se objeví při práci s datovými strukturami a poli
- podmíněné skoky jsou použity v řídících příkazech pro větvení a cykly
- nepodmíněné skoky se používají k volání a návratu z procedur a funkcí a pro příkazy typu switch/case

Četnost použití různých tříd instrukcí se liší

- ovlivňuje návrh datové cesty a řízení