

Úvod do organizace počítače

Podpora procedur
& reprezentace čísel

Datové typy a adresování

Pointery & pole

Programy pro MIPS

Přehled

- Mapa paměti
- Funkce jazyka C
- Instrukční podpora procedur (MIPS)
- Stack
- Procedury - konvence
- Manuální kompilace
- Závěr

Přehled (pokrač.)

- Datové typy
 - Aplikace / požadavky HLL (High Level Languages)
 - Podpora HW (data a instrukce)
- Datové typy procesoru MIPS
- Podpora pro operace s byty a řetězci
- Adresní módy (adresní režimy)
 - Data
 - Instrukce
- Velké konstanty a dlouhé adresy
- Kód SPIM (freeware simulátor)

Přehled (pokrač.)

- Pointery (adresy) a hodnoty
- Předávání argumentů
- „Doba života“ paměti (obsahu!) a dosah
- Aritmetika pointerů
- Pointery a pole
- Pointery u procesoru MIPS

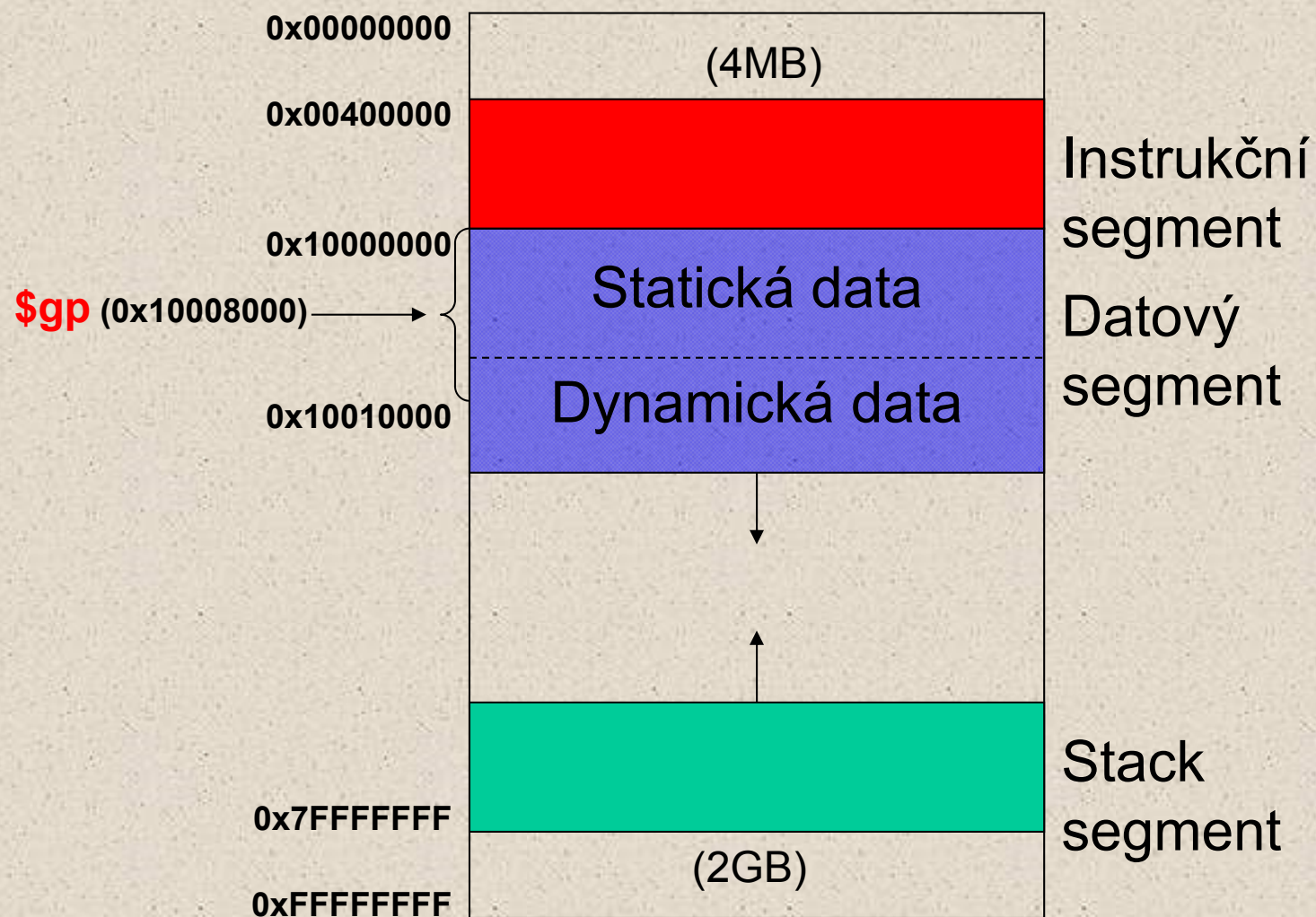
Přehled

- 3 formáty instrukcí MIPS v binárním tvaru:
 - Operační kód (op) určuje formát

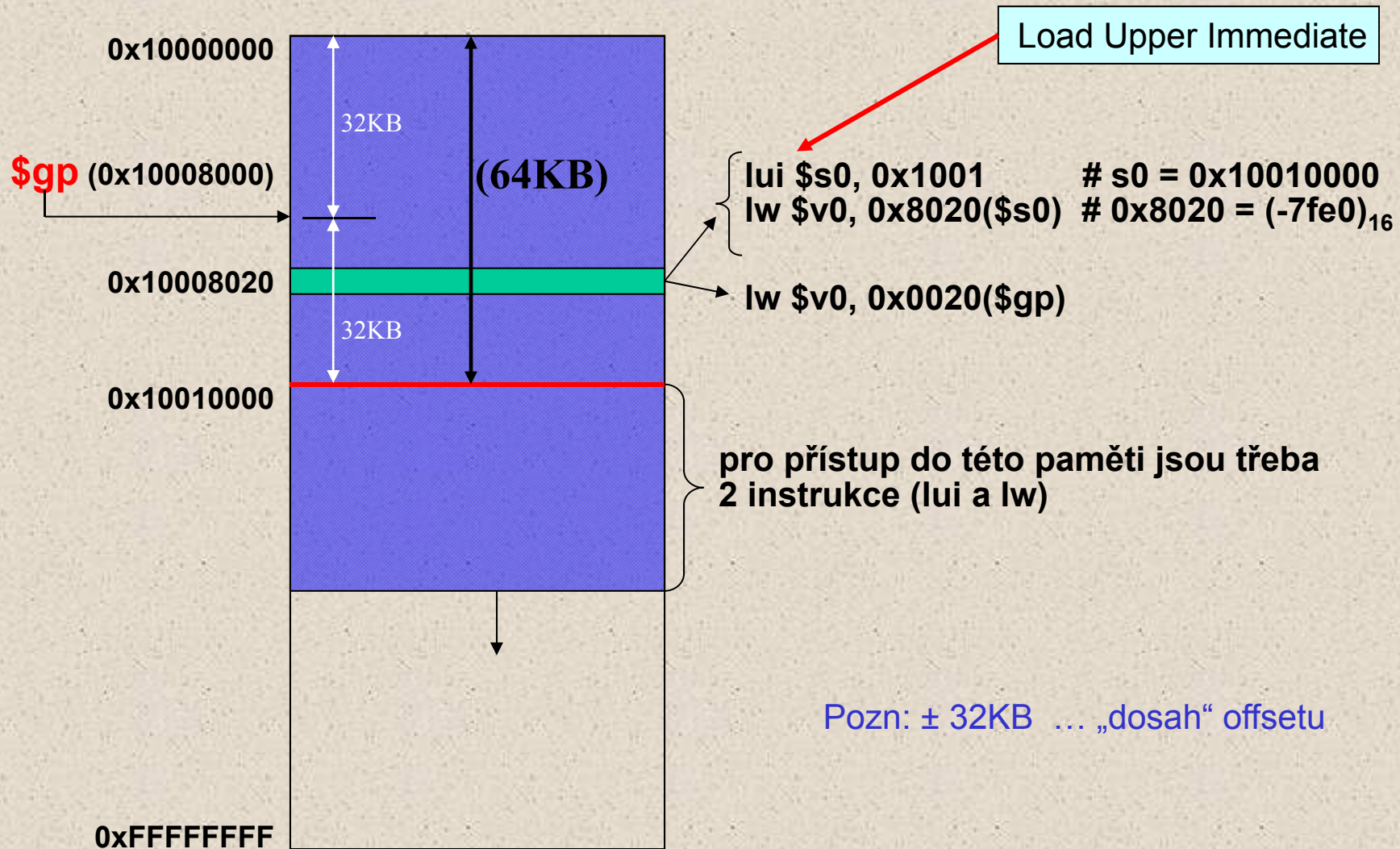
	6 bitů	5 bitů	5 bitů	5 bitů	5 bitů	6 bitů
R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	immediate		
J	op	destination address				

- Operandy
 - Registry: \$0 až \$31 mapovány: \$zero, \$at, \$v_, \$a_, \$s_, \$t_, \$gp, \$sp, \$fp, \$ra
 - Paměť: Mem[0], Mem[4], ... , Mem[4294967292]
 - Index je “adresa” (Array index => Memory index)
- Koncepce programu uloženého v paměti (instrukce jsou čísla!)

Mapa paměti



Datový segment



Funkce / procedury jazyka C

```
main() {  
    int i, j, k;  
  
    ...  
  
    i = mult(j,k);  
  
    ...  
}
```

```
int mult (int mcand, int mlier) {  
    int product;  
  
    product = 0;  
    while (mlier > 0) {  
        product = product + mcand;  
        mlier = mlier - 1; }  
    return product;  
}
```

Jakou informací musí kompilátor sledovat?

Volání procedur

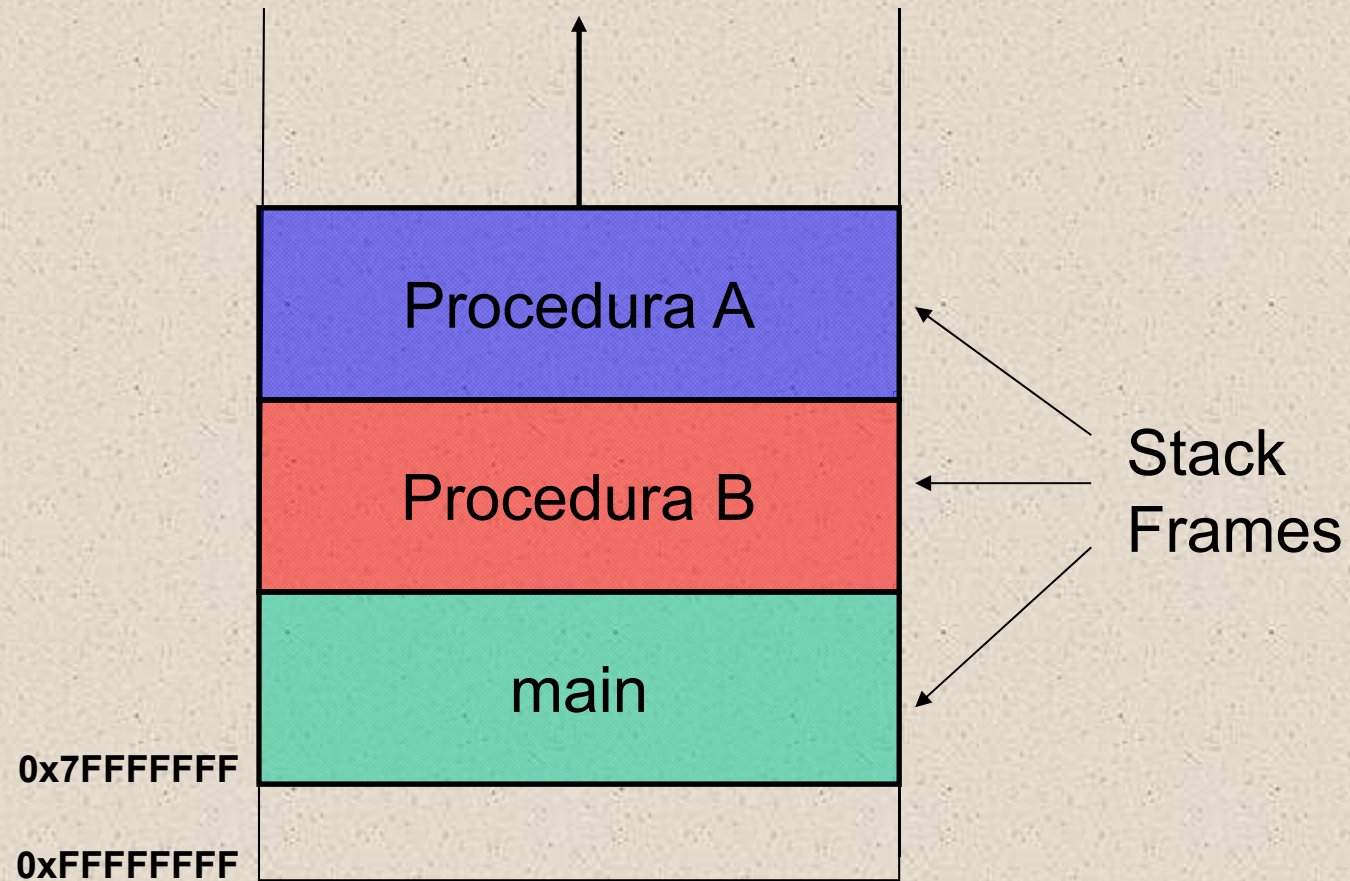
- Problémy
 - Adresa procedury
 - Návratová adresa
 - Argumenty
 - Lokální proměnné
 - Návratová hodnota
- Registry - konvence
 - Labels
 - \$ra
 - \$a0, \$a1, \$a2, \$a3
 - \$s0, \$s1, ..., \$s7
 - \$v0, \$v1
- Dynamická povaha procedur
 - Rámce volání procedur (frames)
 - Argumenty, ukládání registrů, lokální proměnné

Konvence volání procedur

- **Softwarová** pravidla používání registrů

Jméno	Číslo registru	Použití	Rezervován pro call
\$zero	0	the constant value 0	n.a.
\$at	1	reserved for the assembler	n.a.
\$v0-\$v1	2-3	expr. evaluation and function result	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

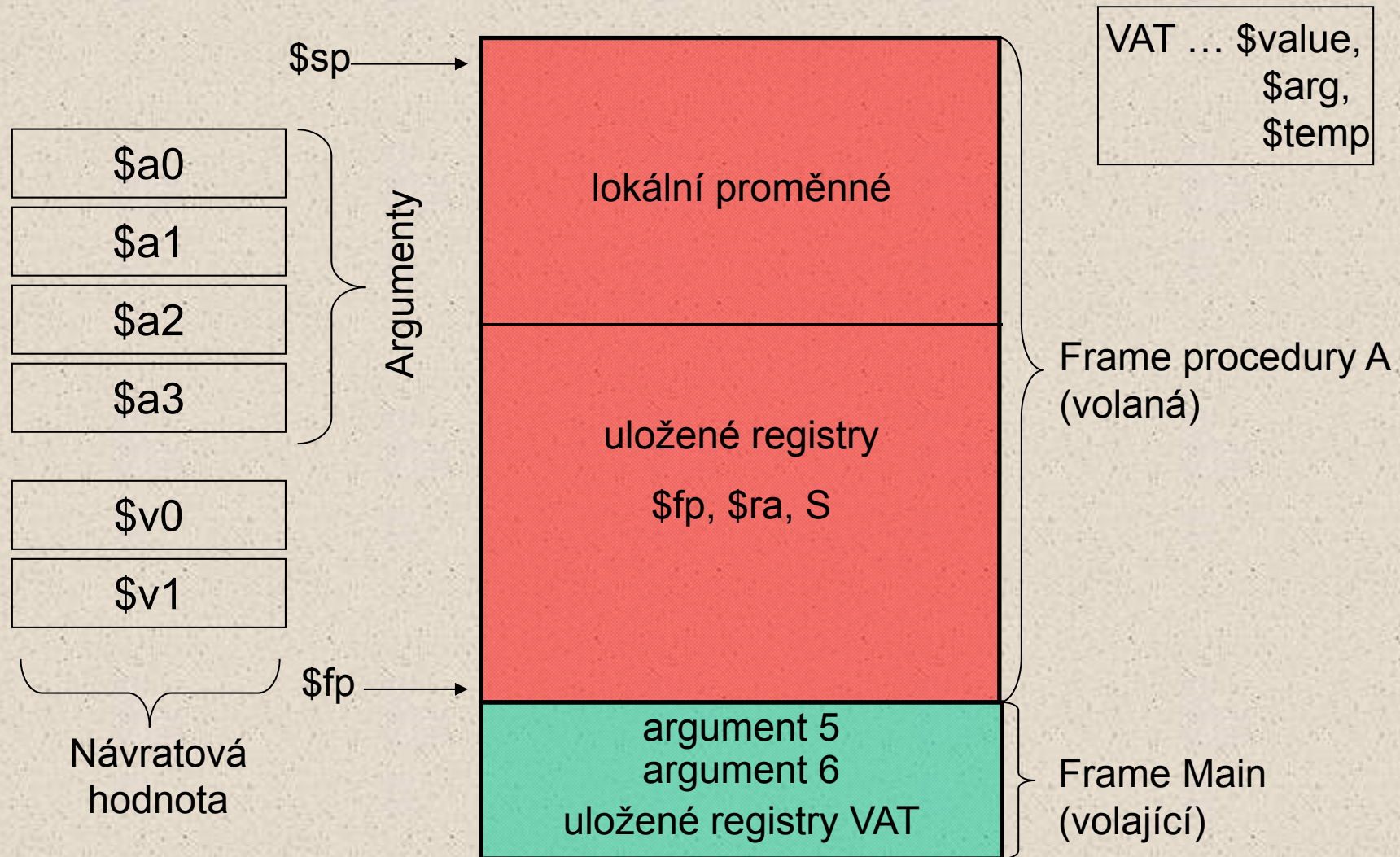
Stack



Registr \$fp

- Konvence MIPS
 - je-li funkci předáváno více parametrů než 4, zapíše se tyto parametry do stacku nad \$fp
 - na tyto extra parametry se dostupuje pomocí pointeru \$fp a příslušného offsetu
- Použití **frame pointeru** je ale nepovinné, některé softwarové produkty jej nevyužívají, na parametry lze dostupovat pomocí \$sp (jako pointer s příslušným offsetem)
- \$fp se během zpracování funkce nemění (představuje pevnou bázi v rámci jednoho provedení funkce)
- \$sp se během zpracování funkce měnit může, na jednotlivé parametry se pak dostupuje s aktuálním offsetem, což je méně přehledné (**překladač určí offsety správně !!**)

Rámce stacku (frames)



Volající/volaný - konvence

- Těsně před vyvoláním funkce volající
 - Předá argumenty (\$a0 - \$a3). Další arg.: uloží do stacku
 - Uloží ukládané registry volajícího (\$a0 - \$a3; \$t0 - \$t9)
 - Proveďte instrukci **jal** (skok na volanou proceduru a uložení návratové adresy)
- Těsně před zahájením výpočtu volané funkce se
 - Alokují paměť pro frame ($\$sp = \$sp - \text{fsize}$)
 - Uloží ukládané registry volaného (\$s0-\$s7; \$fp; \$ra)
 - $\$fp = \$sp + (\text{fsize} - 4)$
- Těsně před návratem do volajícího:
 - Uložení funkční hodnoty do registru \$v0
 - Obnovení všech registrů volané funkce
 - Pop stack frame ($\$sp = \$sp + \text{fsize}$); obnova \$fp
 - Návrat provedením skoku na adresu uloženou v \$ra

Podpora procedur

```
main( ) {  
    ...  
    s = sum (a, b);  
    ...  
}
```

```
int sum(int x, int y) {  
    return x + y;  
}
```

address

```
1000 add    $a0,$s0,$zero    # $a0 = x  
1004 add    $a1,$s1,$zero    # $a1 = y  
1008 addi   $ra,$zero,1016    # $ra=1016  
1012 j      sum              # jump to sum  
1016 ...  
  
2000 sum:   add    $v0,$a0,$a1  
2004 jr     $ra              # jump to 1016
```

Jak by bylo možno řešit volání procedury bez podpory

Instrukce skoku a link

- Jednoduchá instrukce pro skok a uložení návratové adresy

jal: jump & link (*Společné části stavět rychlé*)

- Formát J: **jal label**
- Měla by se nazývat *laj*
 1. (link): uložení adresy příští instrukce do \$ra
 2. (jump): skok na návěští

```
1000 add    $a0,$s0,$zero    # $a0 = x
1004 add    $a1,$s1,$zero    # $a1 = y
1008 jal    sum    # $ra = 1012; jump to sum
1012 ...
```

```
2000 sum:  add    $v0,$a0,$a1
2004 jr     $ra                # jump to 1012
```

Podpora – instrukce jal

Vnořené procedury

```
int sumSquare(int x, int y) {  
    return mult(x,x) + y;  
}
```

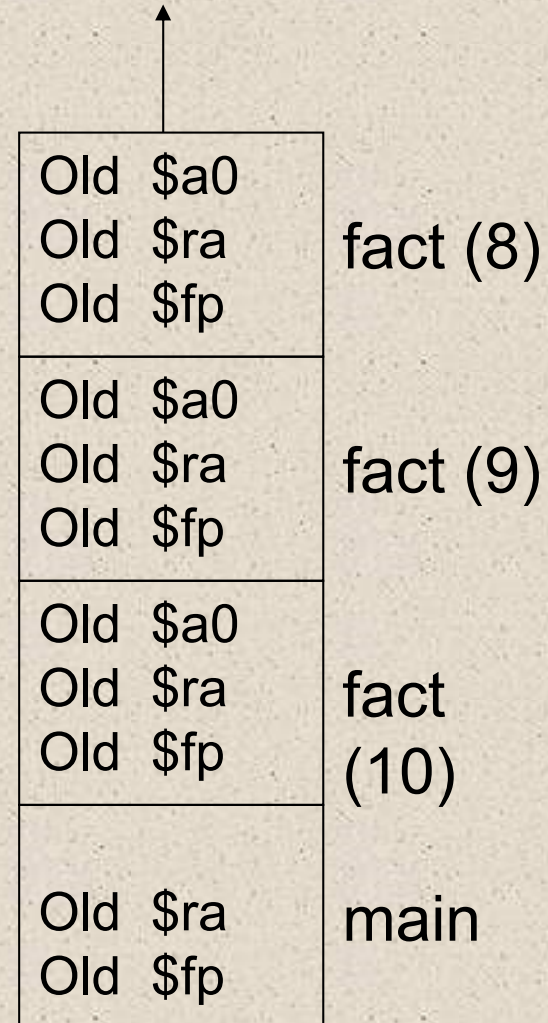
sumSquare:

```
subi $sp,$sp,12    # space on stack  
sw $ra,$ 8($sp)    # save ret addr  
sw $a0,$ 0($sp)    # save x  
sw $a1,$ 4($sp)    # save y  
addi $a1,$a0,$zero # mult(x,x)  
jal mult           # call mult  
lw $ra,$ 8($sp)    # get ret addr  
lw $a0,$ 0($sp)    # restore x  
lw $a1,$ 4($sp)    # restore y  
add $vo,$v0,$a1    # mult()+y  
addi $sp,$sp,12    # free stack space  
jr $ra
```

Příklad(1/2)

```
main( ) {  
    int f;  
    f = fact (10);  
    printf ("Fact(10) = %d\n",  
    f);  
}
```

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



Příklad (2/2)

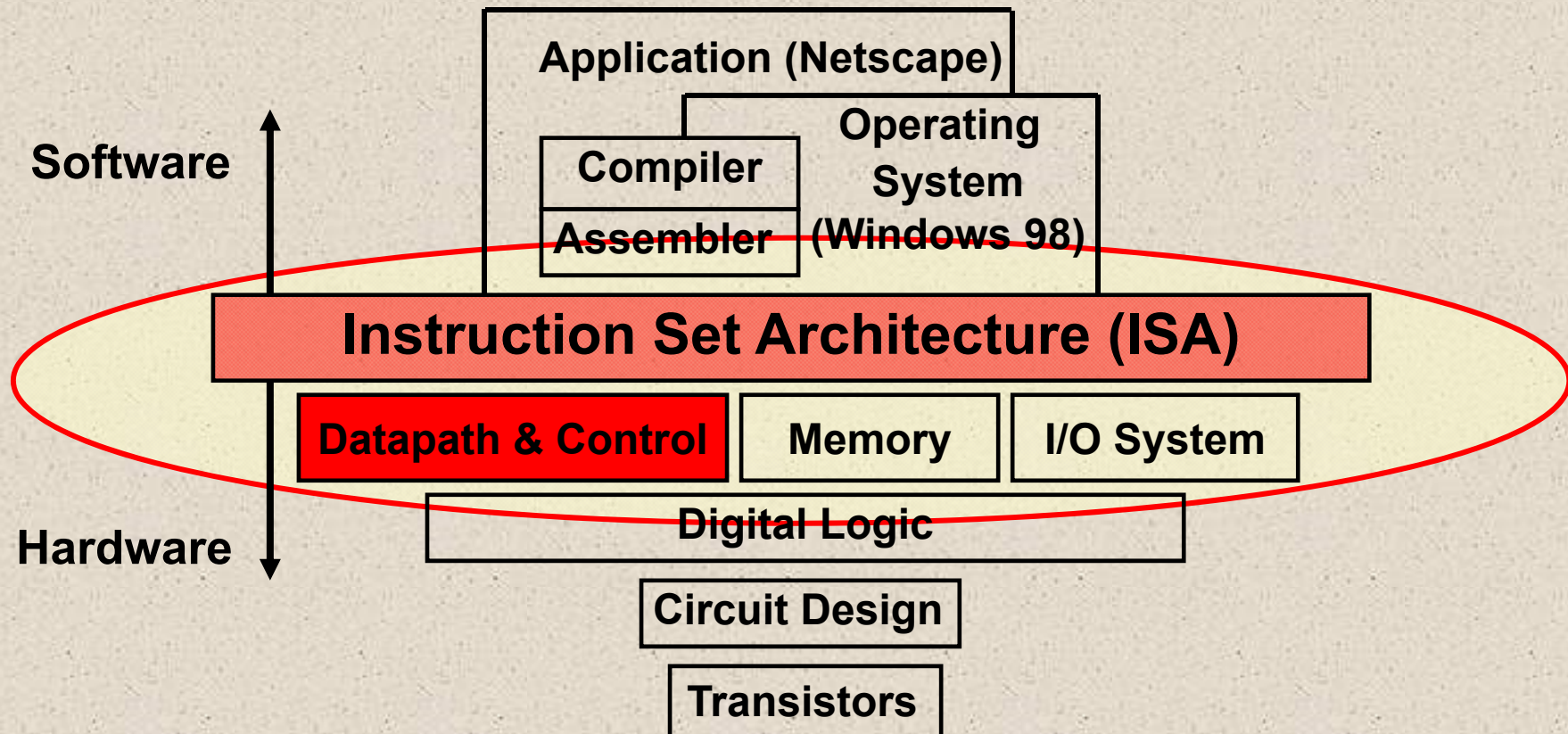
main: subu \$sp, \$sp, 32
 sw \$ra, 20(\$sp)
 sw \$fp, 16(\$sp)
 addiu \$fp, \$sp, 28
 li \$v0, 4
 la \$a0, str
 syscall
 li \$a0, 10
 jal fact
 addu \$a0, \$v0, \$zero
 li \$v0, 1
 syscall
 lw \$ra, 20(\$sp)
 lw \$fp, 16(\$sp)
 addiu \$sp, \$sp, 32
 jr \$ra

fact: subu \$sp, \$sp, 32
 sw \$ra, 20(\$sp)
 sw \$fp, 16(\$sp)
 addiu \$fp, \$sp, 28
 sw \$a0, 0(\$fp)
 lw \$v0, 0(\$fp)
 bgtz \$v0, L2
 li \$v0, 1
 j L1
L2: lw \$v1, 0(\$fp)
 subu \$v0, \$v1, 1
 move \$a0, \$v0
 jal fact
 lw \$v1, 0(\$fp)
 mul \$v0, \$v0, \$v1
L1: lw \$ra, 20(\$sp)
 lw \$fp, 16(\$sp)
 addiu \$sp, \$sp, 32
 jr \$ra

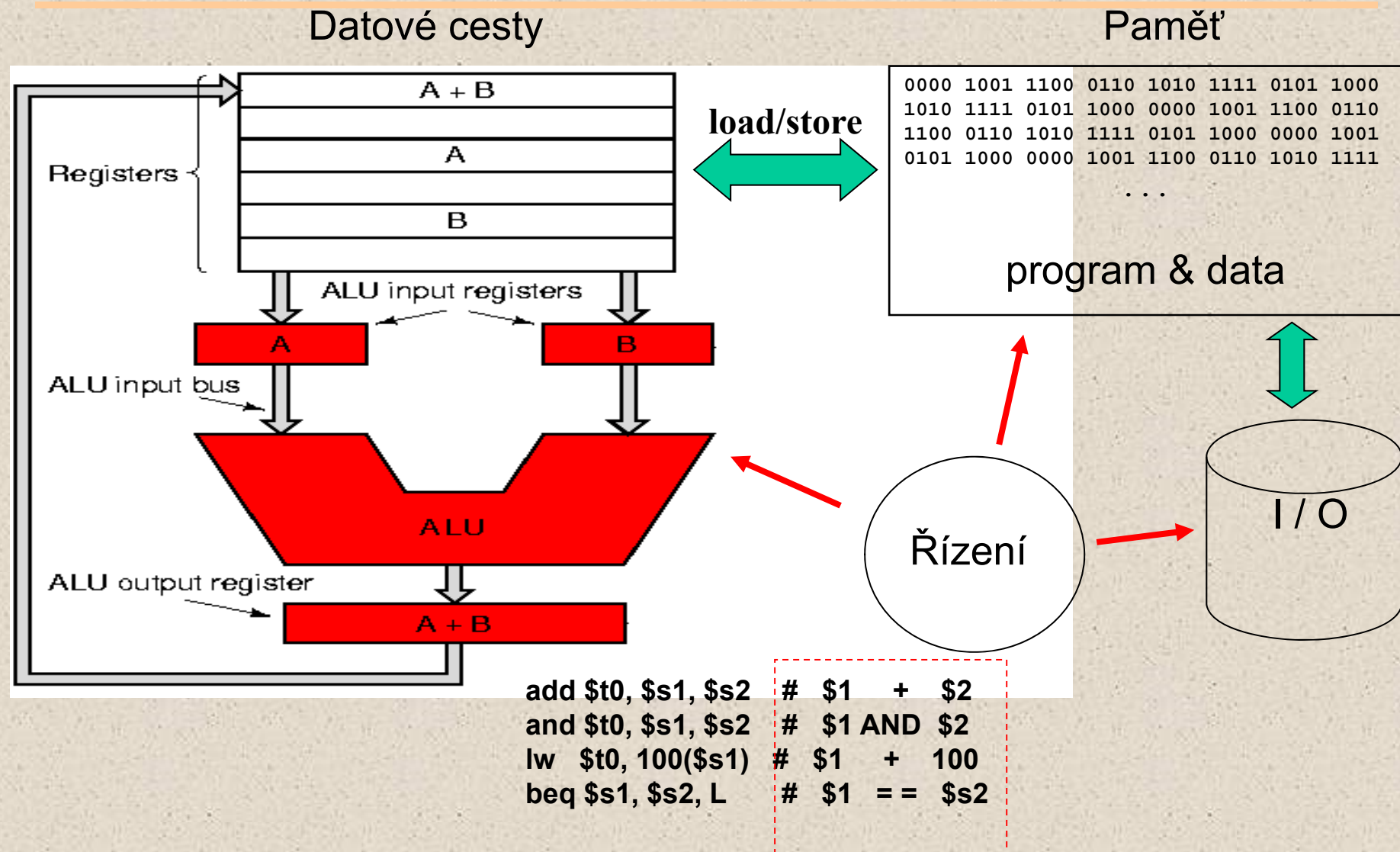
Souhrn

- Konvence volající / volaný
 - Práva a úkoly
 - Volaný používá volně registry VAT
 - Volající používá registry S bez obav z přepsání
- Podpora instrukcí: `jal label` a `jr $ra`
 - `volání` `návrat`
- Stack lze použít kdykoliv je třeba něco uložit.
Nezapomeňte jej opustit ve stejném stavu !!!
- Konvence použití registrů
 - Účel a limity použití
 - Dodržujte pravidla i v případě, že celý program píšete vy !

Nová látka – Číselné systémy



Arithmeticko-logická jednotka



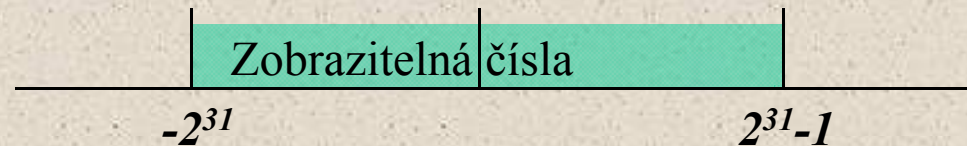
Počítačová aritmetika

- Počítačová aritmetika vs. matematická teorie

- Čísla s limitovanou přesností

- Množina není uzavřená vzhledem k operacím $+$, $-$, $*$, $/$
 \Rightarrow

- Přetečení
- Podtečení
- Nedefinováno



- **Zákony „obyčejné“ algebry vždy v počítačích neplatí (zaokrouhlovací procesy !!!)**

- $a + (b - c) = (a + b) - c$
- $a * (b - c) = a * b - a * c$

- Binární čísla

- Základ: 2
- Číslice: 0 a 1

Dekadická čísla = $\sum_{i=0}^{k-1} d_i \cdot 10^i$

$$\begin{array}{c} (\underbrace{1101}_{B} \underbrace{1010}_{A} \underbrace{1101}_{D})_2 \\ (\quad B \quad \quad A \quad \quad D \quad)_{16} \end{array}$$

$$(d_{k-1} \dots d_2 d_1 d_0) \quad d_i \in \{0, 1, \dots, 9\}$$

Reprezentace dat

Bity mohou reprezentovat cokoliv:

- Znaky
 - 26 písmen => 5 bitů
 - Velká/malá + diakritika => 7 bitů (z 8)
 - „Zbytek“ světových jazyků => 16 bitů (Unicode)
- Čísla bez znaménka ($0, 1, \dots, 2^{n-1}$)
- Logické hodnoty
 - 0 -> False, 1 => True
- Barvy
- Polohu / adresy / příkazy
- **n-bitů může reprezentovat pouze 2^n různých objektů**

Záporná čísla

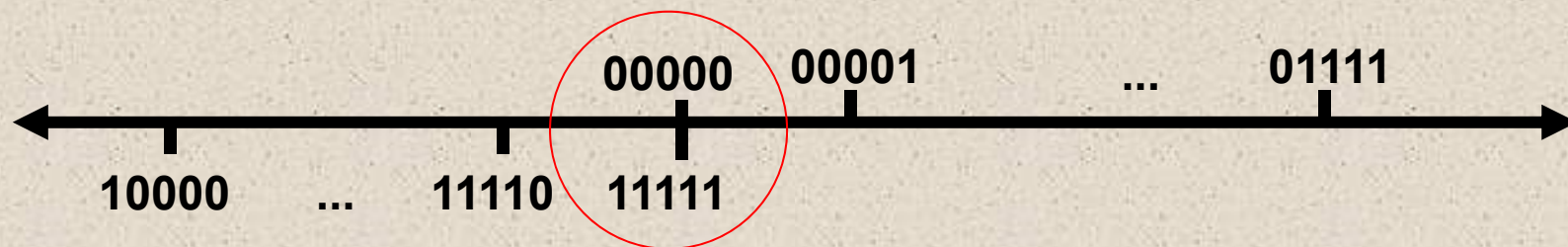
- Dosud **čísla bez znaménka** – *Jak je to se znaménkem?*
- Naivní řešení: bit na levém okraji slova definujeme jako znaménko
 - 0 značí **+**, 1 značí **-** => *znaménkový bit*
 - Ostatní bity tvoří numerickou hodnotu čísla
 - Této reprezentaci se říká **znaménko a amplituda**
- MIPS používá 32-bitová čísla integer (16-bitů immediate/displacement)
 - +1 se zobrazí:
0000 0000 0000 0000 0000 0000 0000 0001
 - 1 se zobrazí :
1000 0000 0000 0000 0000 0000 0000 0001

Problémy se znaménkem a amplitudou

1. Komplikovanější aritmetické obvody
Jsou třeba speciální kroky v závislosti na tom, zda jsou znaménka shodná či nikoliv
(např., $-x \cdot -y = xy = x * y$)
2. Dvojí reprezentace nuly
 - $0x00000000 = +0$
 - $0x80000000 = -0$
 - Komplikace při porovnávání ($+0 == -0$)
- Vzhledem k „zmatku“ kolem nuly se tato reprezentace („přímý kód“) běžně nepoužívá (vyjma fp)

Vyzkoušejme: Jednotkový doplněk

- Získání záporného čísla \Rightarrow inverze bitů
- Příklad: $7_{10} = 00111_2$ $-7_{10} = 11000_2$
- Kladná čísla začínají 0, záporná čísla mají na počátku (vlevo) 1.



- Stále existují dvě nuly (operace.. 😞)
 - $0x00000000 = +0$
 - $0xFFFFFFFF = -0$
- Aritmetika není složitá 😊

Dvojkový doplněk

- Kladná čísla začínají 0
- Záporná čísla ==> inverze kladného + jedna
- Příklad

$$1_{10} = 00000001_2$$

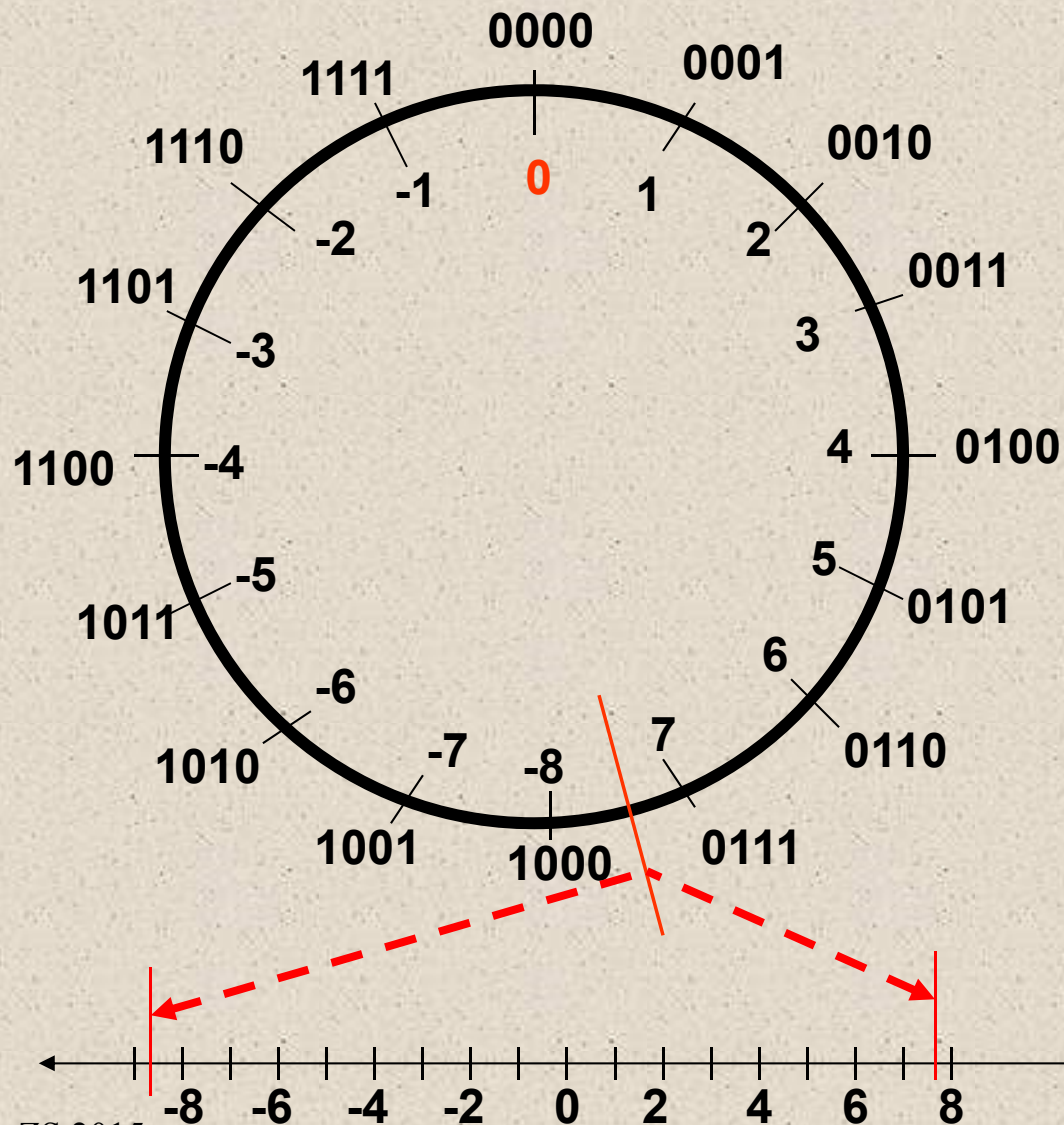
$$-1_{10} = 11111110_2 + 1_2 = 11111111_2$$

$$7_{10} = 00000111_2$$

$$-7_{10} = 11111000_2 + 1_2 = 11111001_2$$

- Kladná čísla mají nekonečně mnoho úvodních 0
- Záporná čísla také mají nekonečně úvodních 1

Grafické znázornění čísel dvojkového doplňku



- 2^{n-1} nezáporných
- 2^{n-1} záporných
- **jediná nula**
- $2^{n-1}-1$ kladných
- porovnání
- přetečení

Příklady: Dvojkový doplněk

$$\begin{aligned} 0000 \dots 0000 \ 0000 \ 0000 \ 0000_2 &= 0_{10} \\ 0000 \dots 0000 \ 0000 \ 0000 \ 0001_2 &= 1_{10} \\ 0000 \dots 0000 \ 0000 \ 0000 \ 0010_2 &= 2_{10} \end{aligned}$$

...

$$\begin{aligned} 0111 \dots 1111 \ 1111 \ 1111 \ 1101_2 &= 2,147,483,645_{10} \\ 0111 \dots 1111 \ 1111 \ 1111 \ 1110_2 &= 2,147,483,646_{10} \\ 0111 \dots 1111 \ 1111 \ 1111 \ 1111_2 &= 2,147,483,647_{10} \\ 1000 \dots 0000 \ 0000 \ 0000 \ 0000_2 &= -2,147,483,648_{10} \\ 1000 \dots 0000 \ 0000 \ 0000 \ 0001_2 &= -2,147,483,647_{10} \\ 1000 \dots 0000 \ 0000 \ 0000 \ 0010_2 &= -2,147,483,646_{10} \end{aligned}$$

...

$$\begin{aligned} 1111 \dots 1111 \ 1111 \ 1111 \ 1101_2 &= -3_{10} \\ 1111 \dots 1111 \ 1111 \ 1111 \ 1110_2 &= -2_{10} \\ 1111 \dots 1111 \ 1111 \ 1111 \ 1111_2 &= -1_{10} \end{aligned}$$

Dvojkový doplněk

- Může reprezentovat kladná i záporná čísla – znaménkový bit (MSB). Zápis:

$$d_{31} \times (-2^{31}) + d_{30} \times 2^{30} + \dots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$$

- Příklad

$$\begin{aligned} & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2 \\ &= 1 \times (-2^{31}) + 1 \times 2^{30} + 1 \times 2^{29} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= -2^{31} + 2^{30} + 2^{29} + \dots + 2^2 + 0 + 0 \\ &= -2,147,483,648_{10} + 2,147,483,644_{10} \\ &= -4_{10} \end{aligned}$$

- **Pozn.!** Musí být známa délka zobrazení => poloha MSB
=> MIPS používá 32 bitů, takže MSB je d_{31}

Dvojkový komplement - algoritmus

- Invertovat (každou 0 na 1 a každou 1 na 0),
potom přičíst 1 k výsledku
 - Součet čísla a jeho 1 doplňku musí být $111\dots111_2 = -1_{10}$
 - Nechť x' značí invertovanou reprezentaci x
 - Potom $x + x' = -1 \Rightarrow x + x' + 1 = 0 \Rightarrow x' + 1 = -x$
- Příklad:
 $x = -4$: 1111 1111 1111 1111 1111 1111 1111 1100₂
 x' : 0000 0000 0000 0000 0000 0000 0000 0011₂
 $x' + 1$: 0000 0000 0000 0000 0000 0000 0000 0100₂
invert: 1111 1111 1111 1111 1111 1111 1111 1011₂
add 1 : 1111 1111 1111 1111 1111 1111 1111 1100₂

Porovnání se zn. a bez zn.

- $X = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
- $Y = 0011\ 1011\ 1001\ 1010\ 1000\ 1010\ 0000\ 0000_2$

Nejednoznačnost:

- Je $X > Y$?
 - Bez znaménka: **ANO**
 - Se znaménkem: **NE**
- Konverze na dekadický tvar (pro kontrolu)
 - Porovnání se znaménkem:
 $-4_{10} < 1,000,000,000_{10}?$
 - Porovnání bez znaménka:
 $-4,294,967,292_{10} < 1,000,000,000_{10}$

Dvojkový doplněk – rozšíření znaménka

- *Problém:* Konvertovat číslo na větší počet bitů
- *Řešení:* MSB rozšířit na nové pozice vlevo
 - 2 doplněk - **kladné** číslo má **nekonečně 0 vlevo**
 - 2 doplněk **záporné číslo má nekonečně 1 vlevo**
 - Bitová reprezentace maskuje tyto bity; rozšířením znaménka se jich část obnoví
 - 16-bitů -4_{10} konvertovat na délku 32-bitů:

16-bitu

32-bitu

MSB

1111 1111 1111 1100₂

1111 1111 1111 1111 1111 1111 1111 1100₂

Definice kódů pro zobrazení záporných čísel

- Přímý kód:

$$N_p = (1 - 2 \cdot x_{n-1}) \cdot \sum_{i=0}^{n-2} x_i \cdot 2^i$$

- Inverzní kód (jedničkový doplněk)

$$N_i = - (2^{n-1} - 1) \cdot x_{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i$$

- Doplněkový kód (dvojkový doplněk)

$$N_d = - 2^{n-1} \cdot x_{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i$$



Datové typy

- Aplikace / HLL

- Integer
- Floating point
- Character
- String
- Date
- Currency
- Text
- Objects (ADT- Abstract Data Types)
- Blob (Binary large object)
- Double precision
- Signed, unsigned

- Podpora hardware

- Numerické datové typy
 - Integer
 - 8 / 16 / 32 / 64 bitů
 - Se znaménkem, bez znaménka
 - BCD čísla (COBOL, Y2K!)
 - Floating point
 - 32 / 64 / 128 bitů
- Nenumerické datové typy
 - Znaky
 - Řetězce
 - Boolean (bitové mapy)
 - Pointery

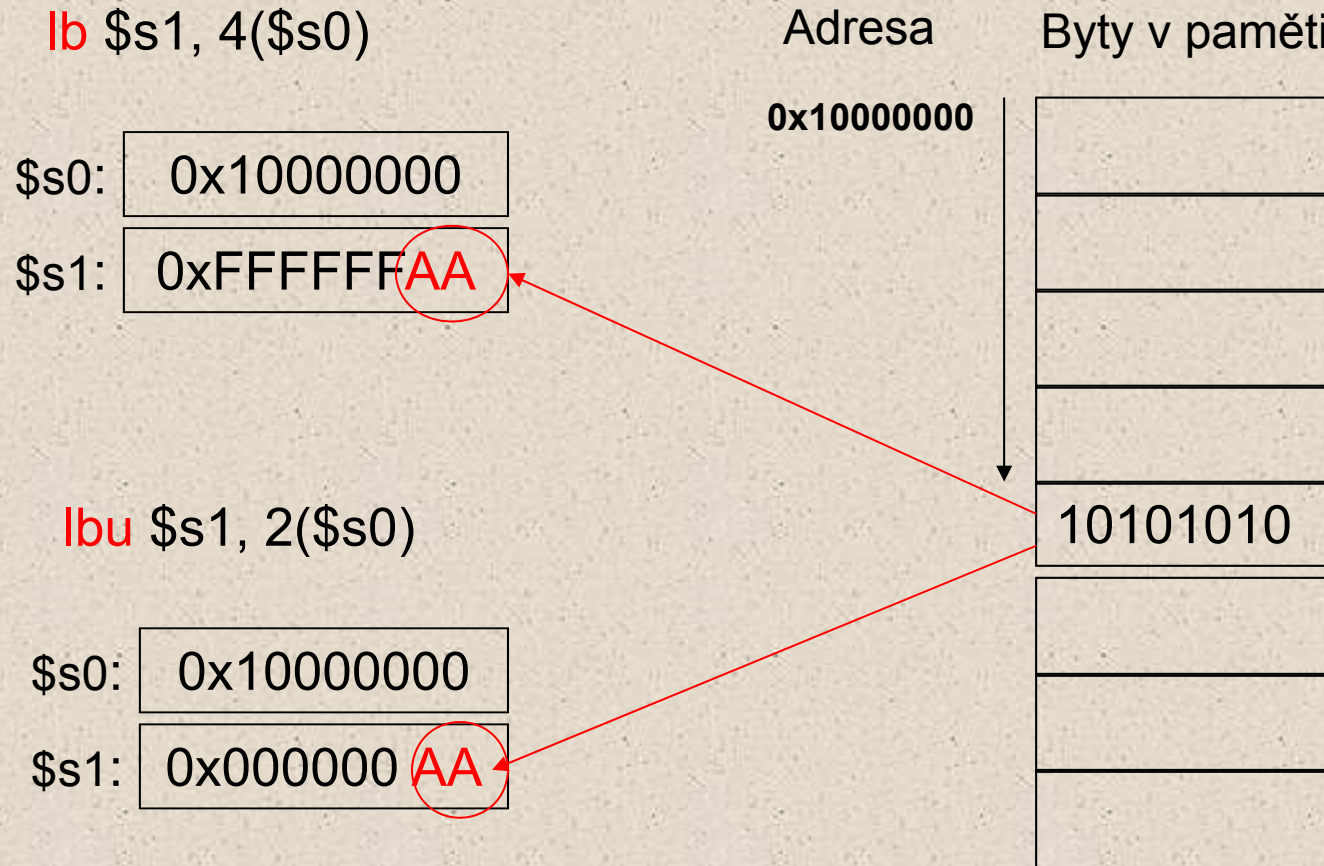
Datové typy MIPS (1/2)

- Základní „strojní“ datové typy: 32-bit slovo
 - 0100 0011 0100 1001 0101 0011 0100 0101
 - Integer čísla (se znaménkem a bez znaménka)
 - 1,128,878,917
 - Floating point čísla
 - 201.32421875
 - 4 ASCII znaky
 - C I S E
 - Adresy do paměti (pointery)
 - 0x43495345
 - Instrukce

Datové typy MIPS (2/2)

- 16-bitové konstanty (immediates)
 - `addi $s0, $s1, 0x8020`
 - `lw $t0, 20($s0)`
- Half word (16 bitů)
 - **lh** (**lhu**): load half word `lh $t0, 20($s0)`
 - **sh**: save half word `sh $t0, 20($s0)`
- Byte (8 bitů)
 - **lb** (**lbu**): load byte `sh $t0, 20($s0)`
 - **sb**: save byte `sh $t0, 20($s0)`

Instrukce pro bytové operace



U instrukce **lb** dochází k rozšíření znaménka

Manipulace s řetězcí

```
Void strcpy (char[], char y[]) {  
    int i;  
    i = 0;  
    while ((x[i]=y[i]) != 0)  
        i = i + 1;  
}
```

Konvence C :

Nulový byte (00000000)
reprezentuje konec řetězce

strcpy:

```
    subi $sp, $sp, 4  
    sw   $s0, 0($sp)  
    add  $s0, $zero, $zero  
L1: add $t1, $a1, $s0 ←  
    lb   $t2, 0($t1)  
    add  $t3, $a0, $s0  
    sb   $t2, 0($t3)  
    beq  $t2, $zero, L2  
    addi $s0, $s0, 1  
    j    L1  
L2: lw   $s0, 0($sp)  
    addi $sp, $sp, 4  
    jr   $ra
```

Důležitost komentářů pro MIPS!

Konstanty

- Časté používání malých konstant (50% operandů)
 - např.: $A = A + 5;$
- Řešení
 - Uložení 'typických konstant' do paměti a jejich používání.
 - Vytvoření HW registrů (jako \$zero) i pro některé další konstanty, např.: 1.
- Instrukce MIPS:
 - slt \$8, \$18, 10
 - andi \$29, \$29, 6
 - ori \$29, \$29, 0x4a
 - addi \$29, \$29, 4

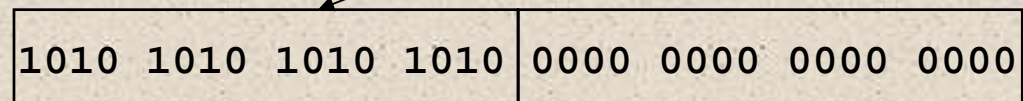
8	29	29	4
101011	10011	01000	0000 0000 0011 0100

Velké konstanty

- Naplnění 32 bitového registru konstantou:

1. Naplnění (16) vyšších bitů

lui \$t0, 1010101010101010



1010 1010 1010 1010	0000 0000 0000 0000
---------------------	---------------------

2. Pak se musí nižší bity přesunout doprava, t. zn.

ori \$t0, \$t0, 1010101010101010

\$t0:

1010 1010 1010 1010	0000 0000 0000 0000
---------------------	---------------------

ori

0000 0000 0000 0000	1010 1010 1010 1010
---------------------	---------------------

1010 1010 1010 1010	1010 1010 1010 1010
---------------------	---------------------

Adresní režimy

- Adresy pro *data* a *instrukce*
- Data (operandy a výsledky)
 - Registry
 - Místa v paměti
 - Konstanty
- Úsporné kódování adres (prostor: 32 bitů)
 - Registry (32) => použito 5 bitů pro zakódování adresy
 - *Destruktivní* instrukce: $\text{reg2} = \text{reg2} + \text{reg1}$
 - Akumulátor
 - Stack
- **Ortogonalita** operačního kódu, adresních režimů a datových typů

Adresní režimy dat

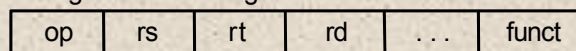
- Registrové adresování
 - Nejobvyklejší způsob (nejrychlejší a nejkratší)
 - `add $3, $2, $1`
- Bázované adresování
 - Operand je v paměti na místě udaném **offsetem**
 - `lw $t0, 20 ($t1)`
- „Immediate“ operandy
 - Operand je malá **konstanta** uvnitř instrukce
 - `addi $t0, $t1, 4` (16-bit integer se znaménkem)

Adresní módy

1. Immediate addressing



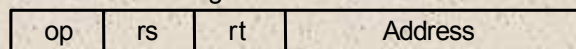
2. Register addressing



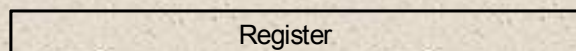
Registers

Register

3. Base addressing



Memory



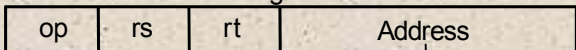
+

Byte

Halfword

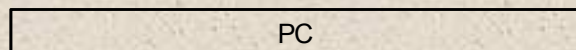
Word

4. PC-relative addressing



* 4

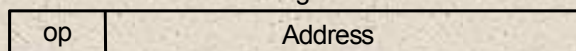
Memory



+

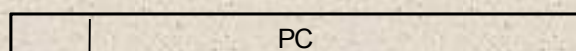
Word

5. Pseudodirect addressing



* 4

Memory



+

Word

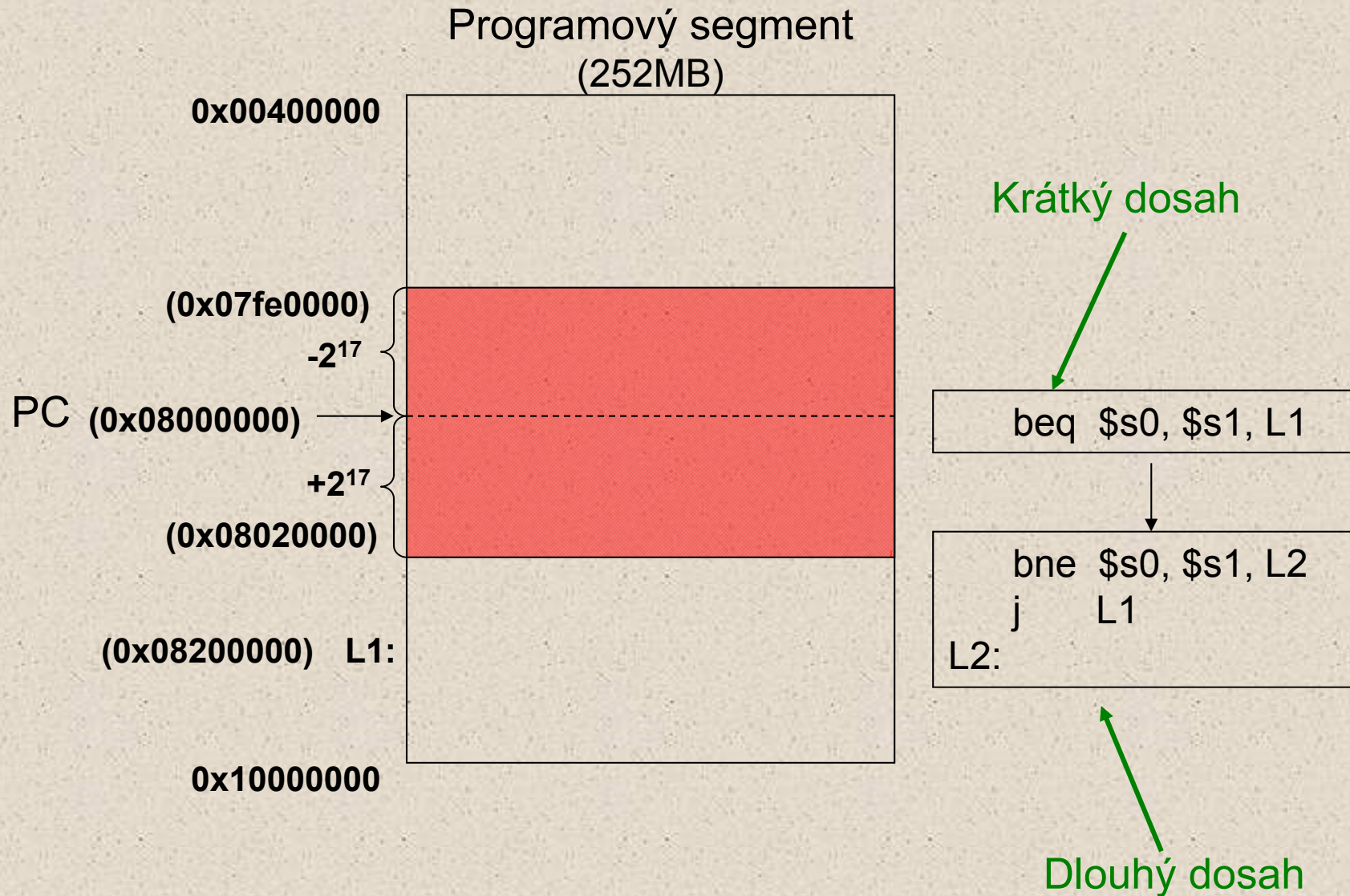
Adresní módy instrukcí

- Adresy jsou dlouhé 32 bitů
- Speciální registr **PC** (**P**rogram **C**ounter) obsahuje adresu právě prováděné instrukce
- PC-relativní adresování (větvení, skoky)
 - Adresa: $PC + (\text{konstanta v instrukci}) * 4$
 - `beq $t0, $t1, 20` (`0x15090005`)
- „Pseudopřímé“ adresování (skoky)
 - Adresa: $PC[31:28] : (\text{konstanta v instrukci}) * 4$

Kód SPIM

PC	MIPS	strojn� kód	Pseudo MIPS
main			
[0x00400020]	add \$9, \$10, \$11	(0x014b4820)	main: add \$t1, \$t2, \$t3
[0x00400024]	j 0x00400048 [exit]	(0x08100012)	j exit
[0x00400028]	addi \$9, \$10, -50	(0x2149ffce)	addi \$t1, \$t2, -50
[0x0040002c]	lw \$8, 5(\$9)	(0x8d280005)	lw \$t0, 5(\$t1)
[0x00400030]	lw \$8, -5(\$9)	(0x8d28fffb)	lw \$t0, -5(\$t1)
[0x00400034]	bne \$8, \$9, 20 [exit-PC]	(0x15090005)	bne \$t0, \$t1, exit
[0x00400038]	addi \$9, \$10, 50	(0x21490032)	addi \$t1, \$t2, 50
[0x0040003c]	bne \$8, \$9, -28 [main-PC]	(0x1509fff9)	bne \$t0, \$t1, main
[0x00400040]	lb \$8, -5(\$9)	(0x8128fffb)	lb \$t0, -5(\$t1)
[0x00400044]	j 0x00400020 [main]	(0x08100008)	j main
[0x00400048]	add \$9, \$10, \$11	(0x014b4820)	exit: add \$t1, \$t2, \$t3
exit			

Dlouhé cílové adresy



Pointery

- Pointer: proměnná, která obsahuje adresu jiné proměnné
 - Výraz pocházející z HLL pro adresu v paměti
- Proč používat pointery?
 - Někdy je to jediná cesta pro rychlý výpočet
 - Často jediná cesta pro získání úsporného kódu
- Proč ne?
 - Častý zdroj chyb v softwaru
 - 1) „Nestálé“ reference (předčasně uvolněné)
 - 2) „Díry“ v paměti (pozdě uvolněné): dlouho trvající úlohy nelze provozovat bez periodického restartu (???)

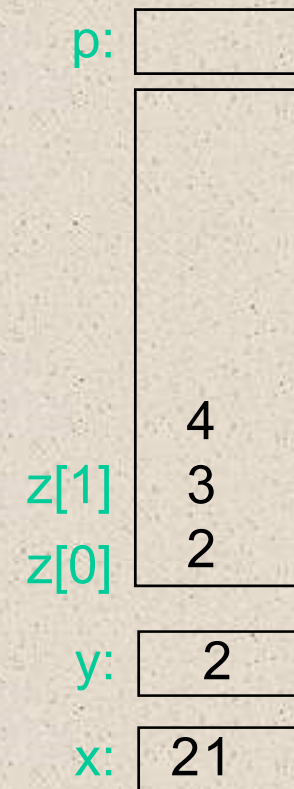
Operátory pro pointery v C

- Předpokládejme, že `c` má hodnotu 100 a leží v paměti na adrese 0x10000000
- Unární operátor `&` dává adresu:
`p = &c;` obsahem `p` bude adresa `c`
 - `p` “ukazuje na” `c` (`p == 0x10000000`)
- Unární operátor `*` dává hodnotu, na kterou pointer ukazuje
 - if `p = &c => *p == 100` („Dereferencování“ pointeru)
- Dereferencing \Rightarrow přenos dat v assembleru
 - `... = ... *p ...;` \Rightarrow **load**
(čtení hodnoty z místa kam ukazuje `p`)
 - `*p = ...;` \Rightarrow **store**
(uložení hodnoty do místa kam ukazuje `p`)

Aritmetika pointerů

```
int x = 1, y = 2;    /* x a y jsou proměnné typu integer */
int z[10];           /* pole 10 int, z ukazuje na počátek */
int *p;              /* p je pointer na int */

x = 21;              /* přiřadí x novou hodnotu 21 */
z[0] = 2; z[1] = 3    /* přiřadí 2 prvému, 3 dalšímu prvku pole */
p = &z[0];            /* p ukazuje na první prvek z */
p = z;               /* totéž jako; p[i] == z[i] */
p = p+1;             /* nyní pointer ukazuje na další prvek, z[1] */
p++;                 /* a opět na další, tentokrát na z[2] */
*p = 4;              /* přiřadí tam 4, z[2] == 4 */
p = 3;               /* špatně! Je to absolutní adresa !!! */
p = &x;              /* p ukazuje na x, *p == 21 */
z = &y               nepřípustné !!!!! jméno pole není proměnná
```



Pointery a assembler

c je int, má hodnotu 100, v paměti na adrese 0x10000000,
p je v \$a0, x je v \$s0

1. `p = &c; /* p gets 0x10000000 */`

`lui $a0, 0x1000 # p = 0x10000000`

2. `x = *p; /* x gets 100 */`

`lw $s0, 0($a0) # dereferencing p`

3. `*p = 200; /* c gets 200 */`

`addi $t0, $0, 200`

`sw $t0, 0($a0) # dereferencing p`

Příklad

```
int strlen(char *s) {  
    char *p = s;          /* p points to chars */  
    while (*p != '\0')  
        p++;              /* points to next char */  
    return p - s;         /* end - start */  
}
```

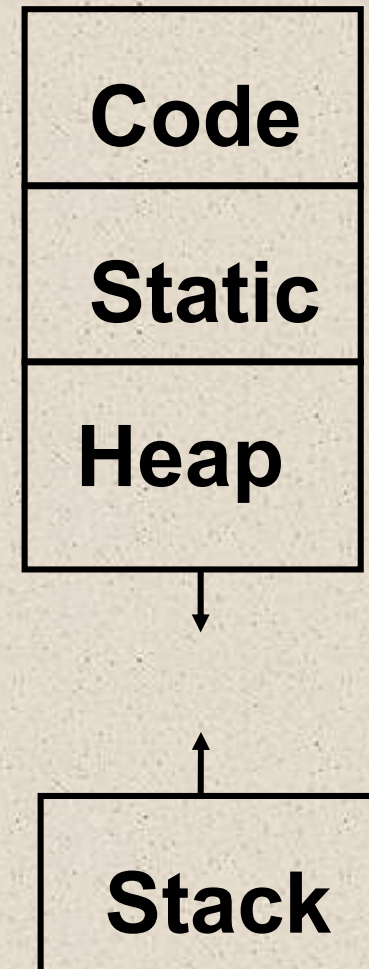
```
        mov    $t0,$a0  
        lbu    $t1,0($t0) /* derefence p */  
        beq    $t1,$zero, Exit  
Loop:   addi    $t0,$t0,1  /* p++ */  
        lbu    $t1,0($t0) /* derefence p */  
        bne    $t1,$zero, Loop  
Exit:   sub     $v0,$t0,$a0  
        jr     $ra
```

Předávání argumentů

- 2 možnosti
 - “Volání hodnotou”: funkci/proceduře se předá kopie položky (argumentu)
 - “Volání odkazem”: funkci/proceduře se předá pointer na položku (argument)
- Proměnné s délkou 1 slovo se předávají hodnotou
- Předání pole ? např., `a[100]`
 - Pascal (volání hodnotou) kopíruje 100 slov z pole `a[]` do stacku
 - C (volání odkazem) předává se pouze pointer (1 slovo) na pole `a[]` v registru

Paměť, její časový rámec a dosah

- Automaticky (přidělen stack)
 - Typicky lokální proměnné uvnitř funkce
 - Vytvořeny po volání **call**, uvolněny po **return**
 - Platnost uvnitř funkce
- Přidělen heap
 - Vytvořen pomocí **malloc**, uvolněn pomocí **free**
 - Přístup pomocí pointerů
- Externí / statická
 - Existuje pro celý program



Pole, pointery a funkce

- 4 verze funkcí, které sčítají dvě pole a ukládají součty do třetího pole (*sumarray*)
 1. Třetí pole je předáváno funkci adresou v parametrech
 2. Použití lokálního pole (ve stacku) pro výsledek a předání pointeru na toto pole
 3. Třetí pole je alokováno v heapu
 4. Třetí pole je deklarováno jako statické
- Smyslem příkladu je ukázat interakci příkazů jazyka C, pointerů a přidělovacích mechanismů paměti

Verze 1

```
int x[100], y[100], z[100];  
sumarray(x, y, z);
```

- Volání v C je interpretováno:

```
sumarray(&x[0], &y[0], &z[0]);
```

- Skutečné předání pointerů na pole

```
addi $a0,$gp,0    # x[0] starts at $gp  
addi $a1,$gp,400  # y[0] above x[100]  
addi $a2,$gp,800  # z[0] above y[100]  
jal  sumarray
```

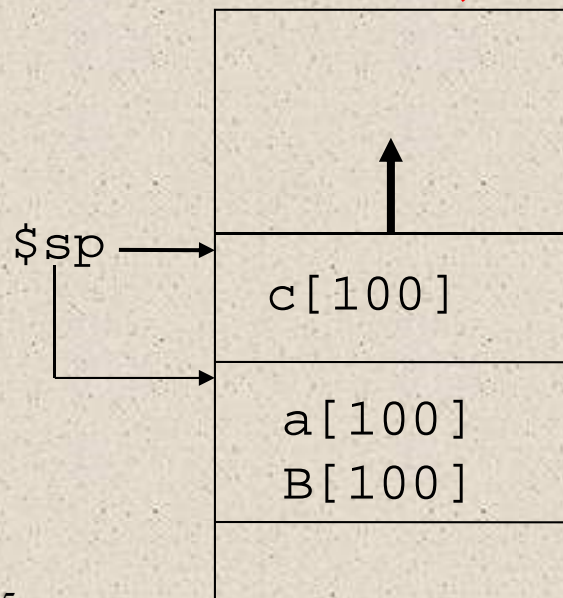
Verze 1: přeložený kód

```
void sumarray(int a[], int b[], int c[]) {  
    int i;  
    for(i = 0; i < 100; i = i + 1)  
        c[i] = a[i] + b[i];  
}
```

	addi	\$t0,\$a0,400	# beyond end of a[]
Loop:	beq	\$a0,\$t0,Exit	
	lw	\$t1, 0(\$a0)	# \$t1=a[i]
	lw	\$t2, 0(\$a1)	# \$t2=b[i]
	add	\$t1,\$t1,\$t2	# \$t1=a[i] + b[i]
	sw	\$t1, 0(\$a2)	# c[i]=a[i] + b[i]
	addi	\$a0,\$a0,4	# \$a0++
	addi	\$a1,\$a1,4	# \$a1++
	addi	\$a2,\$a2,4	# \$a2++
	j	Loop	
Exit:	jr	\$ra	

Verze 2

```
int *sumarray(int a[], int b[]) {  
    int i, c[100];  
    for(i=0; i<100; i=i+1)  
        c[i] = a[i] + b[i];  
    return c;  
}
```



Dostaneme
správný výsledek ??

```
addi $t0,$a0,400 # beyond end of a[]  
addi $sp,$sp,-400 # space for c  
addi $t3,$sp,0    # ptr for c  
addi $v0,$t3,0    # $v0 = &c[0]
```

Loop: beq \$a0,\$t0,Exit

```
lw  $t1, 0($a0) # $t1=a[i]  
lw  $t2, 0($a1) # $t2=b[i]  
add $t1,$t1,$t2 # $t1=a[i] + b[i]  
sw  $t1, 0($t3) # c[i]=a[i] + b[i]  
addi $a0,$a0,4 # $a0++  
addi $a1,$a1,4 # $a1++  
addi $t3,$t3,4 # $t3++
```

j Loop

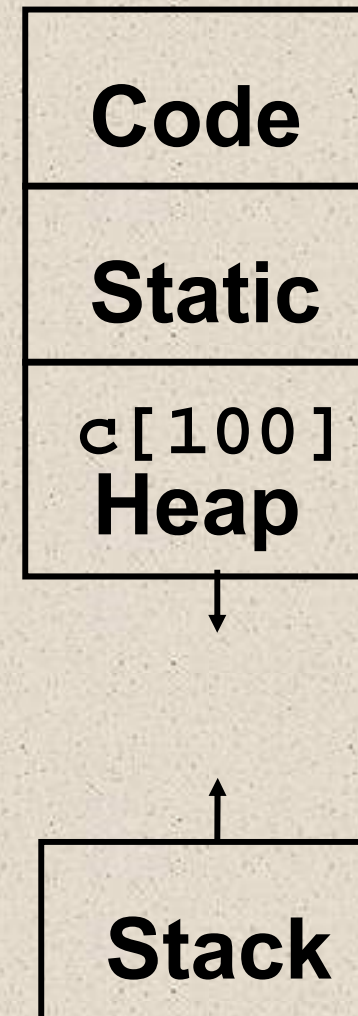
Exit: addi \$sp,\$sp, 400 # pop stack

jr \$ra

Verze 3

```
int * sumarray(int a[], int b[])
{
    int i;
    int *c;
    c = (int *) malloc(100*sizeof(int));
    for(i=0; i<100; i=i+1)
        c[i] = a[i] + b[i];
    return c;
}
```

- Dokud se neuvolní, nelze znova použít
 - Mohou vznikat „díry“ v paměti
 - Java, ... mají na uvolňování prostoru garbage kolektory



Verze 3: přeložený kód

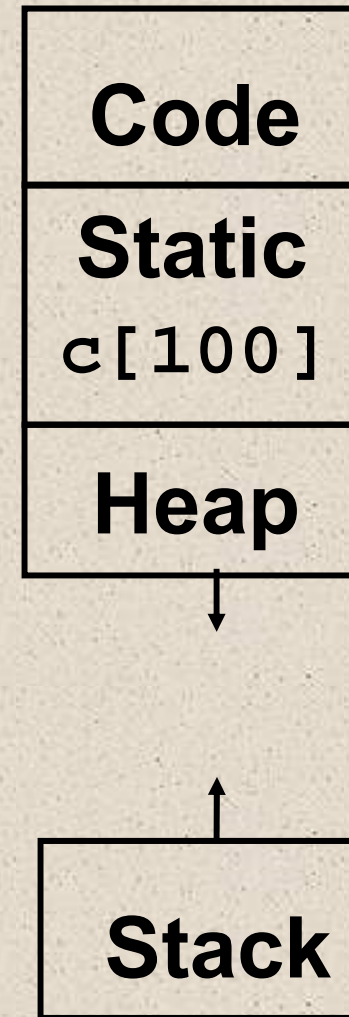
```
addi    $t0,$a0,400 # beyond end of a[]
addi    $sp,$sp,-12 # space for regs
sw      $ra, 0($sp) # save $ra
sw      $a0, 4($sp) # save 1st arg.
sw      $a1, 8($sp) # save 2nd arg.
addi    $a0,$zero,400
jal     malloc
addi    $t3,$v0,0    # ptr for c
lw      $a0, 4($sp) # restore 1st arg.
lw      $a1, 8($sp) # restore 2nd arg.
Loop:   beq    $a0,$t0,Exit
        ... (smyčka jako na předcházejícím snímku )
        j      Loop
Exit:   lw     $ra, 0($sp) # restore $ra
        addi   $sp, $sp, 12 # pop stack
        jr     $ra
```

Verze 4

```
int * sumarray(int a[],int b[])
{
    int i;
    static int c[100];

    for(i=0; i<100; i=i+1)
        c[i] = a[i] + b[i];
    return c;
}
```

- Kompilátor přidělí jednou pro funkci, prostor je znovu využit
 - Změní se při příštím volání sumarray
 - Používáno v knihovnách C



Přehled datových oblastí

MIPS operands		
Name	Example	Comments
32 registers	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register \$zero always equals 0. Register \$at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

Přehled instrukcí

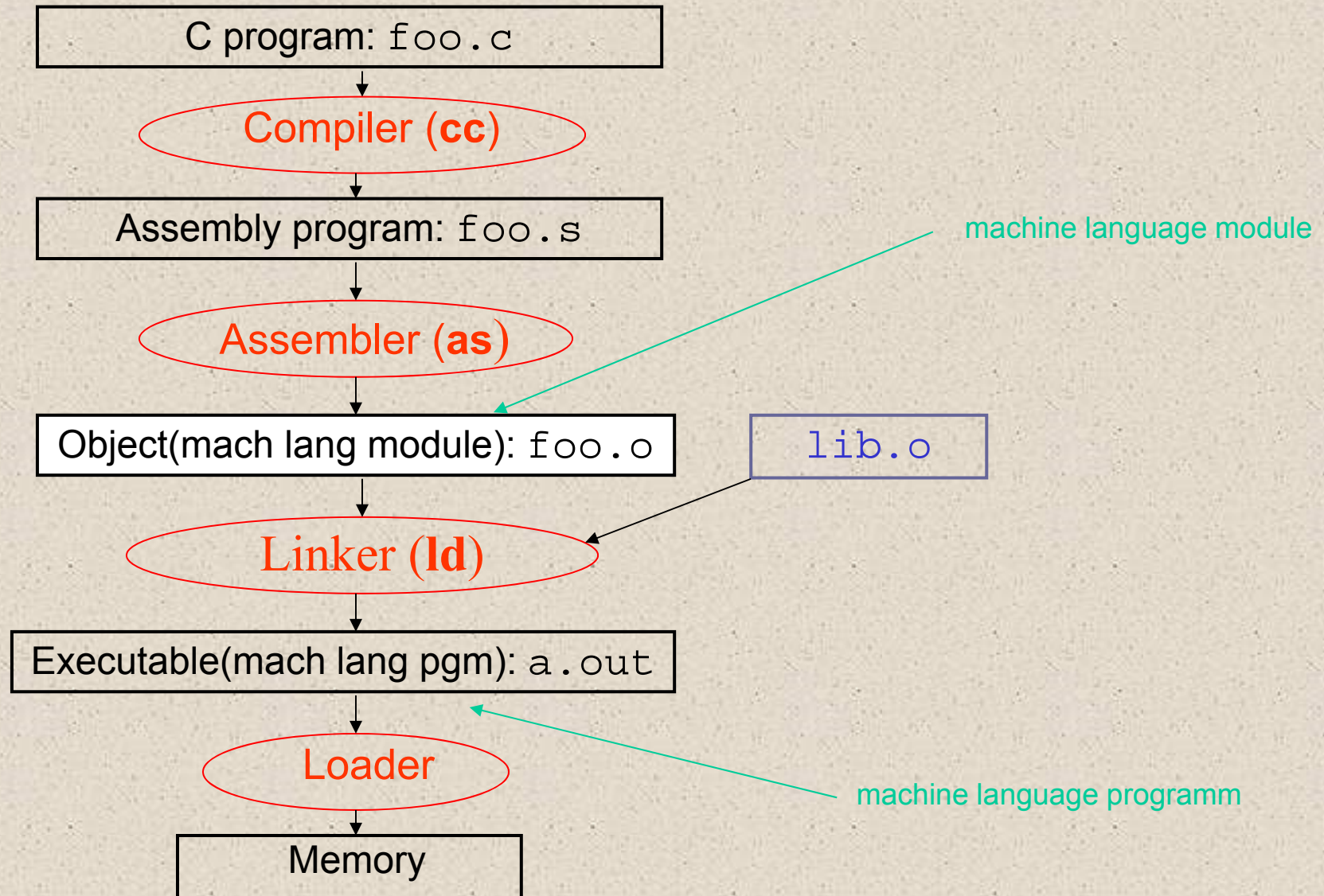
MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$	Three operands; data in registers
	subtract	sub \$s1, \$s2, \$s3	$\$s1 = \$s2 - \$s3$	Three operands; data in registers
	add immediate	addi \$s1, \$s2, 100	$\$s1 = \$s2 + 100$	Used to add constants
Data transfer	load word	lw \$s1, 100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Word from memory to register
	store word	sw \$s1, 100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Word from register to memory
	load byte	lb \$s1, 100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Byte from memory to register
	store byte	sb \$s1, 100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Byte from register to memory
	load upper immediate	lui \$s1, 100	$\$s1 = 100 * 2^{16}$	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq \$s1, \$s2, 25	if ($\$s1 == \$s2$) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1, \$s2, 25	if ($\$s1 \neq \$s2$) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1, \$s2, \$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than; for beq, bne
	set less than immediate	slti \$s1, \$s2, 100	if ($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than constant
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	$\$ra = PC + 4$; go to 10000	For procedure call

Nové - programy MIPS

- Datové typy a adresování zahrnuté v ISA
 - Kompromis mezi požadavky aplikací a hardwarovou implementací
- Datové typy MIPS
 - 32-bitová slova
 - 16-bitová poloviční slova
 - 8-bitové byty
- Adresní módy
 - Data
 - Registry
 - 16-bitové konstanty se znaménkem
 - Bázové adresování
 - Instrukce
 - PC-relativní
 - (Pseudo) přímé

Postup při vývoji programu



Assembler

- Čte a používá **direktivy**
- Nahrazuje makroinstrukce
 - `subu $sp,$sp,32` `addiu $sp, $sp, -32`
 - `sd $a0, 32($sp)` `sw $a0, 32($sp)`
 `sw $a1, 36($sp)`
 - `mul $t7,$t6,$t5` `mult $t6,$t5`
 `mflo $t7`
 - `la $a0, 0xAABBCCDD` `lui $at, 0xAABB`
 `ori $a0, $at, 0xCCDD`
- Generuje strojní jazyk
- Vytváří **objektový soubor (*.o)**

Direktivy assembleru

- Direktivy assembleru, které neprodukují strojní instrukce

<code>.align n</code>	Zarovnat další položku na 2^n bytové hranici
<code>.text</code>	Uložit další položky do uživatelského textového segmentu
<code>.data</code>	Uložit další položky do uživatelského datového segmentu
<code>.globl sym</code>	Na sym se lze odvolávat z jiných souborů
<code>.ascii str</code>	Uložit řetězec str do paměti
<code>.word w1...wn</code>	Uložit n 32-bitových položek do následujících slov v paměti
<code>.byte b1..bn</code>	Uložit n 8-bitových položek do následujících bajtů v paměti
<code>.float f1..fn:</code>	Uložit n floating-point čísel do následujících slov v paměti

Absolutní adresy

- Které instrukce vyžadují editaci relokační?
- Load/store do proměnných ve statické oblasti

lw/sw	\$gp	\$x	address
-------	------	-----	---------

- Podmíněné skoky

beq/bne	\$rs	\$rt	address
---------	------	------	---------

– PC-relativní adresování to nevyžaduje

- Nepodmíněné skokové instrukce

j/jal	xxxxxx
-------	--------

- Přímé (absolutní) reference na data (např. instrukce la)

Generování strojního kódu

- Jednoduché případy
 - Aritmetické a logické operace, posuvy, atd.
 - Všechny informace v instrukci jsou k dispozici.
- Podmíněné skoky (beq, bne)
 - Jakmile jsou makroinstrukce nahrazeny reálnými, lze určit cílové adresy skoků
 - PC-relativní, jednoduchá manipulace
- **Přímá (absolutní) adresa.**
 - Skoky (j a jal)
 - Přímé (absolutní) reference na data
 - **Nelze určit nyní, proto jsou vytvářeny dvě tabulky**

Tabulky assembleru

- **Tabulka symbolů**

- Seznam „položek“ tohoto souboru, který bude použit jinými soubory.
 - Návěští: volání funkcí
 - Data: cokoliv v sekci **.data**; proměnné, které mají být dostupné z více souborů
- První průchod: záznam dvojic návěští-adresa
- Druhý průchod: generování strojního kódu
- Lze skákat na návěští deklarovaná na vyšších adresách (dále v textu)

- **Relokační tabulka**

- Seznam „položek“, pro které je třeba adresa.
- Libovolné návěští, na které se skáče: j nebo jal
 - internal
 - external (včetně knihovních souborů)
- Jakákoliv data (např. instrukce **la**)

Formát objektového souboru

- Hlavička objektového souboru: velikost a poloha ostatních částí objektového souboru
- Kódový segment: strojní kód, binární reprezentace dat zdrojového souboru
- Relokační informace: identifikuje řádky kódu, který musí být “ošetřen”
- Tabulka symbolů: seznam návěští v souboru a data, na která bude dostupováno (budou reference)
- Informace pro debugger

Linker (Link Editor)

- Sestavuje objektové soubory (.o) a vytváří spustitelný soubor - program.
- Umožňuje oddělenou (nezávislou) kompilaci souborů.
 - Rekompilují se pouze pozměněné soubory (moduly)
 - **Windows NT zdrojový kód má >30 M řádek!**
 - **Windows XP patrně není známo ani Microsoftu 😊**
- Edituje “odkazy” ve skokových instrukcích, vyhodnocuje reference do paměti.
- Proces (vstup: objektové soubory vygenerované assemblerem).
 - Krok 1: slučuje kódové segmenty všech .o souborů
 - Krok 2: slučuje datové segmenty všech .o souborů a připojuje je na konec kódových segmentů
 - Krok 3: vyhodnocuje reference. Prochází relokatační tabulku a ošetří každou položku (doplní všude **absolutní adresy**)

Vyhodnocení referencí

- Čtyři typy referencí (adres)
 - PC-relativní (např. `beq`, `bne`): nikdy se nerelokují
 - Absolutní adresy (`j`, `jal`): vždy se relokují
 - Externí reference (`jal`): vždy se relokují
 - Datové reference (`lui` and `ori`): vždy se relokují
- Linker *předpokládá*, že prvé slovo prvního programového segmentu leží na adrese 0x00000000.
- Údaje, které linker zná:
 - Délka každého programového a datového segmentu
 - Uspořádání programových a datových segmentů
- Linker vypočítává:
 - Absolutní adresy všech návěští, na které se skáče (interní nebo externí) a každá data, na které se program odkazuje

Loader

- Spustitelný program je uložen na disku.
- Činnost loaderu: natažení programu do paměti a spuštění
- Ve skutečnosti, loader je částí operačního systému (OS)
 1. Čte hlavičku, aby určil velikost programu a datových segmentů
 2. Vytvoří nový adresní prostor pro program tak velký, aby mohl obsahovat kódové a datové segmenty i stackový segment
 3. Kopíruje instrukce a data ze souboru programu do paměti
 4. Kopíruje argumenty předané programu do stacku
 5. Inicializuje registry, **\$sp** = první volné místo ve stacku
 6. Skáče na startovací rutinu, která kopíruje argumenty programu ze stacku do registrů a nastavuje registr PC
 7. Když se rutina main vrací, startovací rutina ukončuje program systémovým voláním **exit**

Příklad : **C** => Asm => Obj => Exe => Run

```
#include <stdio.h>
int main (int argc, char *argv[]) {
    int i;
    int sum = 0;

    for (i = 0; i <= 100; i = i + 1)
        sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d\n",
        sum);
}
```


Příklad : C => **Asm** => Obj => Exe => Run

```
.text
.align      2
.globl      main
main:
    subu $sp,$sp,32
    sw      $ra, 20($sp)
    sd     $a0, 32($sp)
    sw      $0, 24($sp)
    sw      $0, 28($sp)
loop:
    lw      $t6, 28($sp)
    mul    $t7, $t6,$t6
    lw      $t8, 24($sp)
    addu   $t9, $t8,$t7
    sw      $t9, 24($sp)
```

```
    addu   $t0, $t6, 1
    sw      $t0, 28($sp)
    ble    $t0,100, loop
    la     $a0, str
    lw      $a1, 24($sp)
    jal     printf
    move   $v0, $0
    lw      $ra, 20($sp)
    addiu  $sp,$sp,32
    jr      $ra
    .data
    .align  0
str:
        .asciiz "The sum
    from 0 .. 100 is %d\n"
```

Příklad: C => Asm => **Obj** => Exe => Run

Nahradí makroinstrukce; přiřadí adresy (start na 0x00)

00	addiu	\$29,\$29,-32	30	addiu	\$8,\$14, 1
04	sw	\$31,20(\$29)	34	sw	\$8,28(\$29)
08	sw	\$4, 32(\$29)	38	slti	\$1,\$8, 101
0c	sw	\$5, 36(\$29)	3c	bne	\$1,\$0, loop
10	sw	\$0, 24(\$29)	40	lui	\$4, hi. str
14	sw	\$0, 28(\$29)	44	ori	\$4,\$4,lo. str
18	lw	\$14,28(\$29)	48	lw	\$5,24(\$29)
1c	mult	\$14,\$14	4c	jal	printf
20	mflo	\$15	50	add	\$2, \$0, \$0
24	lw	\$24,24(\$29)	54	lw	\$31,20(\$29)
28	addu	\$25,\$24,\$15	58	addiu	\$29,\$29,32
2c	sw	\$25,24(\$29)	5c	jr	\$31
			60	The	

Tabulka symbolů a relokační tabulka

- Tabulka symbolů

– Návěští	Adresa
main:	0x00000000
loop:	0x00000018
str:	0x10000430
printf:	0x004003b0

- Relokační informace

– Adresa	Typ instr.	Závislost
– 0x00000040	HI16	str
– 0x00000044	LO16	str
– 0x0000004c	jal	printf

Příklad : C => Asm => Obj => Exe => Run

00	addiu	\$29,\$29,-32	30	addiu	\$8,\$14, 1
04	sw	\$31,20(\$29)	34	sw	\$8,28(\$29)
08	sw	\$4,32(\$29)	38	slti	\$1,\$8, 101
0c	sw	\$5,36(\$29)	3c	bne	\$1,\$0, -9
10	sw	\$0, 24(\$29)	40	lui	\$4, 4096
14	sw	\$0, 28(\$29)	44	ori	\$4,\$4,1072
18	lw	\$14, 28(\$29)	48	lw	\$5,24(\$29)
1c	multu	\$14, \$14	4c	jal	1048812
20	mflo	\$15	50	add	\$2, \$0, \$0
24	lw	\$24, 24(\$29)	54	lw	\$31,20(\$29)
28	addu	\$25,\$24,\$15	58	addiu	\$29,\$29,32
2c	sw	\$25, 24(\$29)	5c	jr	\$31

Příklad : C => Asm => Obj => Exe => Run

0x00400000	001001111011110111111111111100000
0x00400004	101011111011111110000000000010100
0x00400008	101011111010010000000000000100000
0x0040000c	101011111010010100000000000100100
0x00400010	10101111101000000000000000011000
0x00400014	10101111101000000000000000011100
0x00400018	10001111101011100000000000011100
0x0040001c	10001111101110000000000000011000
0x00400020	00000001110011100000000000011001
0x00400024	00100101110010000000000000000001
0x00400028	00101001000000010000000001100101
0x0040002c	10101111101010000000000000011100
0x00400030	0000000000000000000111100000010010
0x00400034	00000011000011111100100000100001
0x00400038	00010100001000001111111111110111
0x0040003c	10101111101110010000000000011000
0x00400040	00111100000001000001000000000000
0x00400044	10001111101001010000000000011000
0x00400048	00001100000100000000000011101100
0x0040004c	00100100100001000000010000110000
0x00400050	10001111101111110000000000010100
0x00400054	00100111101111010000000000100000
0x00400058	0000001111100000000000000001000
0x0040005c	000000000000000000001000000100001

Souhrn - programy MIPS

- Kompilátor konvertuje HLL soubor do jednoho souboru – jazyk assembler.
- Assembler odstraní makroinstrukce, konvertuje vše co lze do strojového jazyka a vytváří relokační tabulku pro linker. Ten pro každý **.s** soubor vytváří **.o** soubor.
- Linker spojuje všechny **.o** soubory a vyhodnocuje absolutní adresy.
- Loader načítá spustitelný soubor do paměti a startuje provádění programu.

Závěr

- Objekty jsou v počítači reprezentovány jako bitové vzory:
 n bitů $\Rightarrow 2^n$ různých vzorů
- Dekadická čísla - konvence lidí
- Binární čísla – konvence u počítačů (fyzikálně opodstatněná !!!)
- Dvojkový doplněk – nejrozšířenější ve výpočetní technice: budeme ho dále používat
- Operace počítače nad číselnou reprezentací je *abstrakce* reálných operací nad reálnými objekty
- Přetečení:
 - Čísel je nekonečný počet
 - Počítače jsou “omezené”

Závěr

- Data mohou znamenat cokoliv
- Datové typy MIPS : Číslo, řetězec, boolean
- Adresování: Pointery, hodnoty
 - Mnoho adresních módů (adresa přímá, nepřímá,...)
 - Přístup k hlavní paměti – bazové adresování
- Pole: *velké „kusy“ paměti*
 - Pointery versus stack
 - Pozor na díry v paměti!