Instrukce a návrh instrukční sady

Lubomír Bulej
KDSS MFF UK

Pro připomenutí: počítač je (jen) stroj

Vykonává program

- Posloupnost instrukcí uložených v paměti.
- Vykoná instrukci a posune se na “následující”.
  - “Neví” co dělá, “nechápe” smysl programu.

Instrukce jsou velmi jednoduché

- Vesměs operace s čísly.

Vše je zakódováno do čísel

- Nejen vstupní a výstupní data...
  - Text, obrázky, hudba, 3D scéna, ...
- … ale také vykonávaný program!

Jaké instrukce potřebujeme?

It is easy to see by formal-logical methods that ***there exist certain [instruction sets] that are in abstract adequate to control and cause the execution of any sequence of operations****…*

… The really decisive considerations from the present point of view, in electing an [instruction set], are more of a practical nature: simplicity of the equipment demanded by the [instruction set], and the clarity of its application to the actually important problems together with the speed of its handling of those problems.

– Burks, Goldstine, and von Neumann, 1947

Aritmetické operace

Sčítání (dvou “proměnných”)

- Nejzákladnější ze základních operací.

```
add a, b, c          # a = b + c
```

- Sečte “proměnné” **b** a **c** a uloží výsledek do **a**.
- Pracuje se vždy se třemi operandy.
  - Pravidelnost (regularita) zjednodušuje návrh!

Sčítání tří (čtyř) proměnných

- Vyžaduje dvě (tři) operace

```
add a, b, c          # a = b + c
add a, a, d          # a = b + c + d
add a, a, e          # a = b + c + d + e
```

Přiřazovací příkazy (1)

Jednoduché výrazy

```
a := b + c;
d := a - e;
```

Překlad do assembleru

```
add a, b, c        # a = b + c
sub d, a, e        # d = a - e
```

Přiřazovací příkazy (2)

Složitější výrazy

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

- Překladač musí příkaz rozložit na více operací.

Překlad do assembleru

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

- Programátor pracuje pouze s proměnnými.
- Překladač určuje kam uložit mezivýsledky.

Operandy

Instrukce pracují pouze s *registry*

- Omezený počet paměťových míst v hardware dostupných programátorovi.
  - 32 v případě architektury MIPS.
  - Více než 16-32 není nutně lepší. Proč?
  - Menší často znamená rychlejší!
- Velikost registru je rovněž omezená.
  - 32 bitů (slovo) v případě 32-bitové architektury MIPS.

Výkon závisí na efektivním využití registrů

- Překladač určuje, v jakých registrech budou uloženy hodnoty používané v různých fázích běhu programu.

Označení registrů architektury MIPS

Číslo v instrukčním kódu

- 5 bitů pro označení registrů `0` – `31`.

Symbolická jména v assembleru

- Odráží typický způsob použití registru.
- `$r0` (`$zero`) a `$r31` (`$ra`) jsou speciální.

Jméno	Číslo	Použití	Jméno	Číslo	Použití
$zero	0	Konstantní hodnota 0.	**$t8 – $t9**	24 – 25	Další mezivýsledky.
$at	1	Rezervováno pro assembler.	**$k0 – $k1**	26 – 27	Rezervováno pro jádro OS.
$v0 – $v1	2 – 3	Hodnoty výsledků v výrazů.	**$gp**	28	Global pointer.
$a0 – $a3	4 – 7	Argumenty funkcí.	**$sp**	29	Stack pointer.
$t0 – $t7	8 – 15	Registry pro mezivýsledky.	**$fp / $s8**	30	Frame pointer.
$s0 – $s7	16 – 23	Uschovávané registry.	**$ra**	31	Návratová adresa.

Použití registrů při překladu

Složitější výraz

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

Kód v assembleru MIPS

- Překladač přiřadil hodnoty **`f`**, **`g`**, **`h`**, **`i`** a **`j`** do registrů **`$s0`**, **`$s1`**, **`$s2`**, **`$s3`** a **`$s4`**.

```
add $t0, $s1, $s2        # $t0 = g + h
add $t1, $s3, $s4        # $t1 = i + k
sub $s0, $t0, $t1        # f = $t0 - $t1
```

Paměťové operandy

Všechno je uloženo v paměti

- Proměnné a datové struktury obsahují typicky více prvků než je k dispozici registrů.
  - V registrech může být jen malé množství dat.

Aritmetické operace pracují pouze s registry

- Pro přesuny dat mezi pamětí a registry jsou potřeba *instrukce přenos dat*.
  - Instrukce musí poskytnout *adresu* v paměti.
- Paměť je 1-rozměrné pole bajtů.
  - Adresa slouží jako index (s počátkem v 0).
  - 32-bitové adresy slov musí být zarovnané na 4 bajty.

Instrukce pro přenos dat

1 adresovací režim: Bázová adrese v registru, celočíselný offset v instrukci.

Load/store word

- `lw $rd, imm16 ($rs)`
  R[rd] = M[R[rs] + signext32 (imm16)]
- `sw $rt, imm16 ($rs)`
  M[R[rs] + signext32 (imm16)] = R[rt]

Load/store byte

- `lb $rd, imm16 ($rs)`
  R[rd] = signext32 (M[R[rs] + signext32 (imm16)]**[7:0]**)
- `lbu $rd, imm16 ($rs)`
  R[rd] = zeroext32 (M[R[rs] + signext32 (imm16)]**[7:0]**)
- `sb $rt, imm16 ($rs)`
  M[R[rs] + signext32 (imm16)]**[7:0]** = R[rt]**[7:0]**

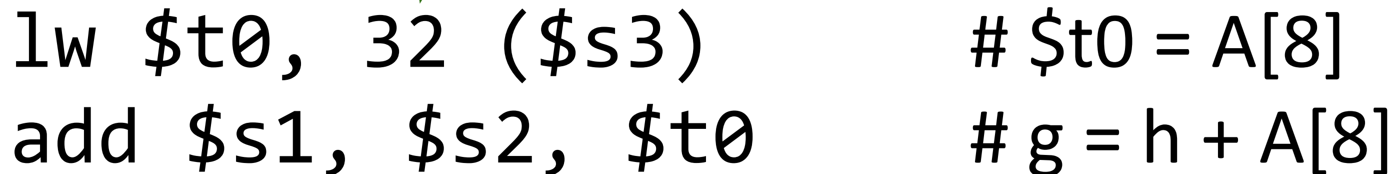
Použití paměťových operandů

Fragment programu

```
var A : array [0 .. 99] of Integer;
g := h + A[8];
```

Kód v assembleru MIPS

- Proměnné **g** a **h** přiřazeny do **$s1** a **$s2**.
- Bázová (počáteční) adrese pole **A** je v **$s3**.
- Offset prvku A[8] je 8×SizeOf(Integer)



```
lw $t0, 32 ($s3)         # $t0 = A[8]
add $s1, $s2, $t0        # g = h + A[8]
```

Použití instrukcí load a store

Fragment programu

- Jednoduché přiřazení, dva paměťové operandy.

```
var A : array [0 .. 99] of Integer;
A[12] := h + A[8];
```

Kód v assemleru MIPS

- Proměnná **h** přiřazena do **$s2**.
- Bázová adresa pole **A** je v **$s3**.

```
lw $t0, 32 ($s3)        # $t0 = A[8]
add $t0, $s2, $t0       # $t0 = h + A[8]
sw $t0, 48 ($s3)        # A[12] = h + A[8]
```

Konstanty a přímé operandy

Cíl: vyhnout se čtení běžných konstant z paměti

- Zvyšování/snižování řídící proměnné cyklu nebo indexu, inicializace součtů a součinů…
  - Běžné hodnoty: 0, 1, -1, 2, … (velikosti struktur)
  - Časté operace musí být rychlé!

Přímé operandy

- **`addi $rd, $rs, imm16`**

  add immediate, R[rd] = R[rs] + signext32 (imm16)
- **`li $rd, imm32`**

  load immediate, R[rd] = imm32

Nula je speciální, "zadrátovaná" v `$r0`

- **`move $rd, $rs = add $rd, $rs, $r0`**

  R[rd] = R[rs]

Logické operace

Operace s bity a bitovými poli v rámci slov

- Izolace, nastavování a nulování bitů.

Bitové operace

- **`and/or/xor/nor $rd, $rs, $rt`**
  - `not $rd, $rs = nor $rd, $rs, $rs/$r0`
- **`andi/ori/xori $rd, $rs, imm16`**
  R[rd] = R[rs] and/or/xor zeroext32 (imm16)

Operace posunu

- **`sll/slr $rd, $rs, shamt`**
  shift logical left/right, R[rd] = R[rs] << / >> shamt
- **`sra $rd, $rs, shamt`**
  shift arithmetic right, R[rd] = R[rs] >>> shamt

Použití logických operací

Fragment programu

```
shamt := (insn and $000007C0) shr 6;
```

Kód v assembleru MIPS

- Proměnné **shamt**, **insn** přiřazeny do **$s1**, **$s2**.

```
andi $t0, $s2, 0x7C0        # $t0 = insn & 0x7C0
srl $s1, $t0, 6             # shamt = $t0 >> 6
```

Instrukce pro podporu rozhodování (1)

Odlišují počítač od kalkulátoru

- Umožňují výběr instrukcí, které budou vykonány v závislosti na vstupech a mezivýsledcích.
  - Řídící příkazy v programovacích jazycích.

Větvení / Podmíněné skoky

- **`beq $rd, $rs, addr`**
  branch if eq, **if** R[rs] == R[rt] **then** PC = addr **else** PC = PC + 4
- **`bne $rd, $rs, addr`**
  branch not eq, **if** R[rs] <> R[rt] **then** PC = addr **else** PC = PC + 4

Nepodmíněné skoky

- **`j addr`**
  jump, PC = addr

Adresa následující instrukce

Překlad příkazů *if-then-else*

Fragment programu

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

- Proměnné **`f`**, **`g`**, **`h`**, **`i`** a **`j`** přiřazeny do registrů **`$s0`**, **`$s1`**, **`$s2`**, **`$s3`** a **`$s4`**.

Kód v assembleru MIPS

```
    bne $s3, $s4, Else        # (i <> j) ⇒ PC = Else
    add $s0, $s1, $s2         # f = g + h
    j End                     # PC = End
Else:
    sub $s0, $s1, $s2         # f = g - h
End:
    ...
```

Překlad *while* cyklu

Fragment programu

```
while (save [i] = k) do
    i := i + 1;
```

Kód v assembleru MIPS

- Proměnné **i**, **k** přiřazeny do **$s3**, **$s5**; bázová adresa pole **save** je v **$s6**.

```
Loop:
   sll $t1, $s3, 2          # $t1 = i × 4
   add $t1, $t1, $s6        # $t1 = @save[i]
   lw $t0, 0 ($t1)          # $t0 = save[i]
   bne $t0, $s5, End        # (save[i] <> k) ⇒ PC = End
   addi $s3, $s3, 1         # i = i + 1
   j Loop                   # PC = Loop
End:
```

Instrukce pro podporu rozhodování (2)

Set on less than

- Umožňuje testovat všechny vztahy (v kombinaci s instrukcemi *beq*/*bne*)

Znaménková varianta

- **`slt $rd, $rs, $rt`**

  if R[rs] $<_s$ R[rt] **then** R[rd] = 1 **else** R[rd] = 0
- **`slti $rd, $rs, imm16`**

  if R[rs] $<_s$ signext32 (imm16) **then** R[rd] = 1 **else** R[rd] = 0

Bez-znaménková varianta

- **`sltu $rd, $rs, $rt`**

  if R[rs] $<_U$ R[rt] **then** R[rd] = 1 **else** R[rd] = 0
- **`sltiu $rd, $rs, imm16`**

  if R[rs] $<_U$ zeroext32 (imm16) **then** R[rd] = 1 **else** R[rd] = 0

Překlad cyklu *repeat-until* a *do-while*

Fragment programu

```
i := 0;
repeat
    i := i + 1;
until i >= k;
```

Kód v assembleru MIPS

- Proměnné **i** a **k** přiřazeny do registrů **$s3**, a **$s5**.

```
   move $s3, $zero              # i = 0
Loop:
   addi $s3, $s3, 1             # i = i + 1
   slt $t0, $s3, $s5            # $t0 = (i < k)
   bne $t0, $zero, Loop         # ($t0 <> 0) ⇒ PC = Loop
End:
```

Překlad *for* cyklu (1)

Fragment programu

```
var
   a : array [0 .. 4] of Integer;
   s, i : integer;
begin
   s := 0;
   for i := 0 to 4 do begin
      s := s + a[i];
   end;
end.
```

Překlad *for* cyklu (2)

Kód v assembleru MIPS

```
    move $s2, $zero          # s = 0
    move $s1, $zero          # i = 0
    j Condition              # PC = Condition
Body:
    sll $t0, $s1, 2          # $t0 = i × 4
    add $t0, $t0, $s0        # $t0 = @a[i]
    lw $t1, 0 ($t0)          # $t1 = a[i]
    add $s2, $s2, $t1        # s = s + a[i]
    addi $s1, $s1, 1         # i = i + 1
Condition:
    slti $t2, $s1, 5         # $t2 = (i < 5)
    bne $t2, $zero, Body     # ($t2 <> 0) ⇒ PC = Body
End:
```

Podpora pro procedury/funkce (1)

Základní nástroj pro strukturování programů

- Volání odkudkoliv se vstupními parametry.
- Návrat do místa volání s návratovou hodnotou.
- Jedna z cest k abstrakci a znovupoužitelnosti kódu.

Základní kroky při vykonání procedury/funkce

- Uložení parametrů na místo dostupné rutině.
- Předání řízení do kódu rutiny.
- Alokace paměti nutné pro vykonání úkolu.
- Provedení požadovaného úkolu.
- Uložení výsledků na místo dostupné volajícímu.
- Návrat do místa volání.

Podpora pro procedury/funkce (2)

Volání podpogramu

- `jal addr`
  jump and link, \$ra = R[31] = PC + 4; PC = addr
- `jalr $rs`
  jump and link register, \$ra = R[31] = PC + 4; PC = R[rs]

Adresa následující instrukce

Nepřímé skoky / návrat z podprogramu

- `jr $rs`
  jump register, PC = R[rs]

Registry používané při volání rutin

- První 4 argumenty předávané v `$a0` – `$a3`
- Návratová hodnota vracena v `$v0` – `$v1`
- Návratové adrese předávaná v `$ra` (`$r31`)

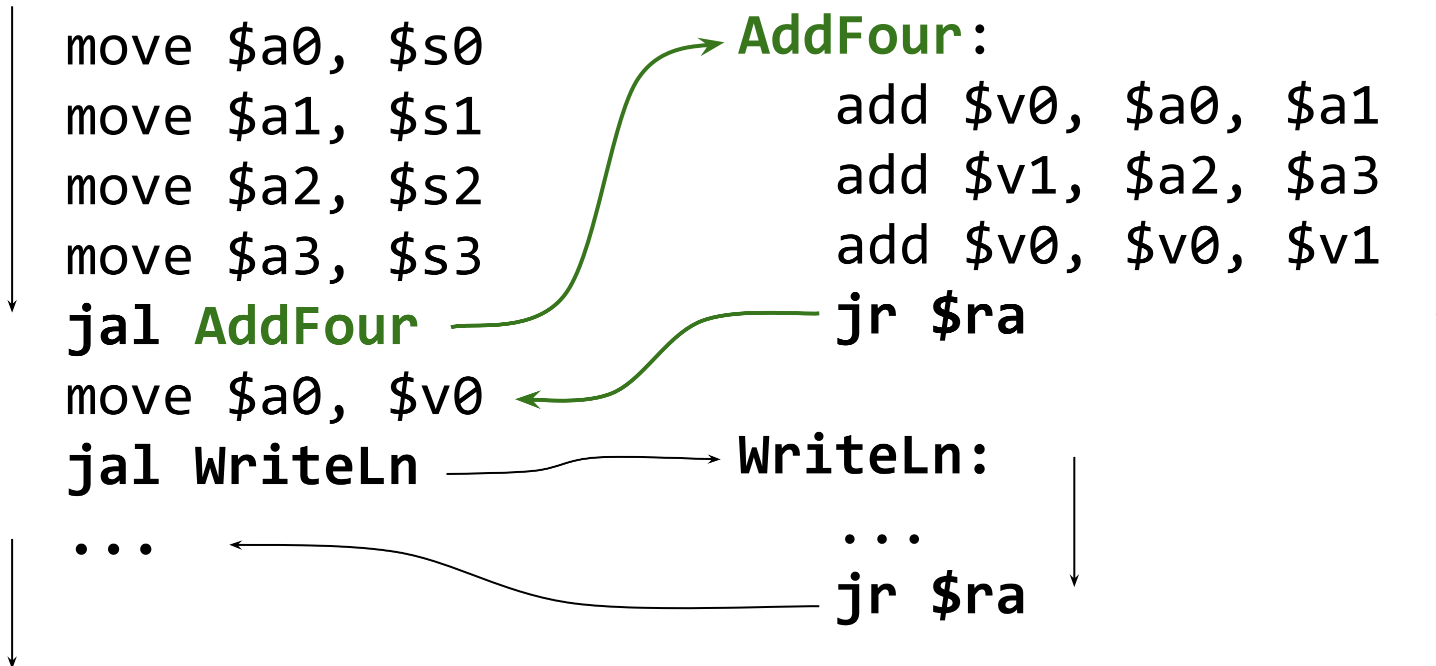
Jednoduché volání funkce

Fragment programu

```
WriteLn (AddFour (a, b, c, d));
```

Kód v assembleru MIPS

- Proměnné **a**, **b**, **c** a **d** přiřazeny do **$s0**, **$s1**, **$s2** a **$s3**.



Podpora pro procedury/funkce (3)

Ukládání obsahu registrů do paměti

- Po návratu z funkce volající očekává v registrech hodnoty, které tam uložil.
- Rutina pracuje s více hodnotami než kolik má k dispozici registrů.

Předávání argumentů přes paměť

- Rutina může mít více než 4 parametry.

Vracení hodnot přes paměť

- Návratová hodnota může být struktura.

Alokace paměti pro lokální proměnné

- Řídící proměnné cyklů, mezivýsledky, ...

Alokace prostoru pro lokální data

V paměti, ale kde?

- Místo nemůže být pevné, protože rutina může být volána z více míst.
  - V důsledku přímé nebo nepřímé (tranzitivní) rekurze.
  - Rutina může být volána z více vláken.

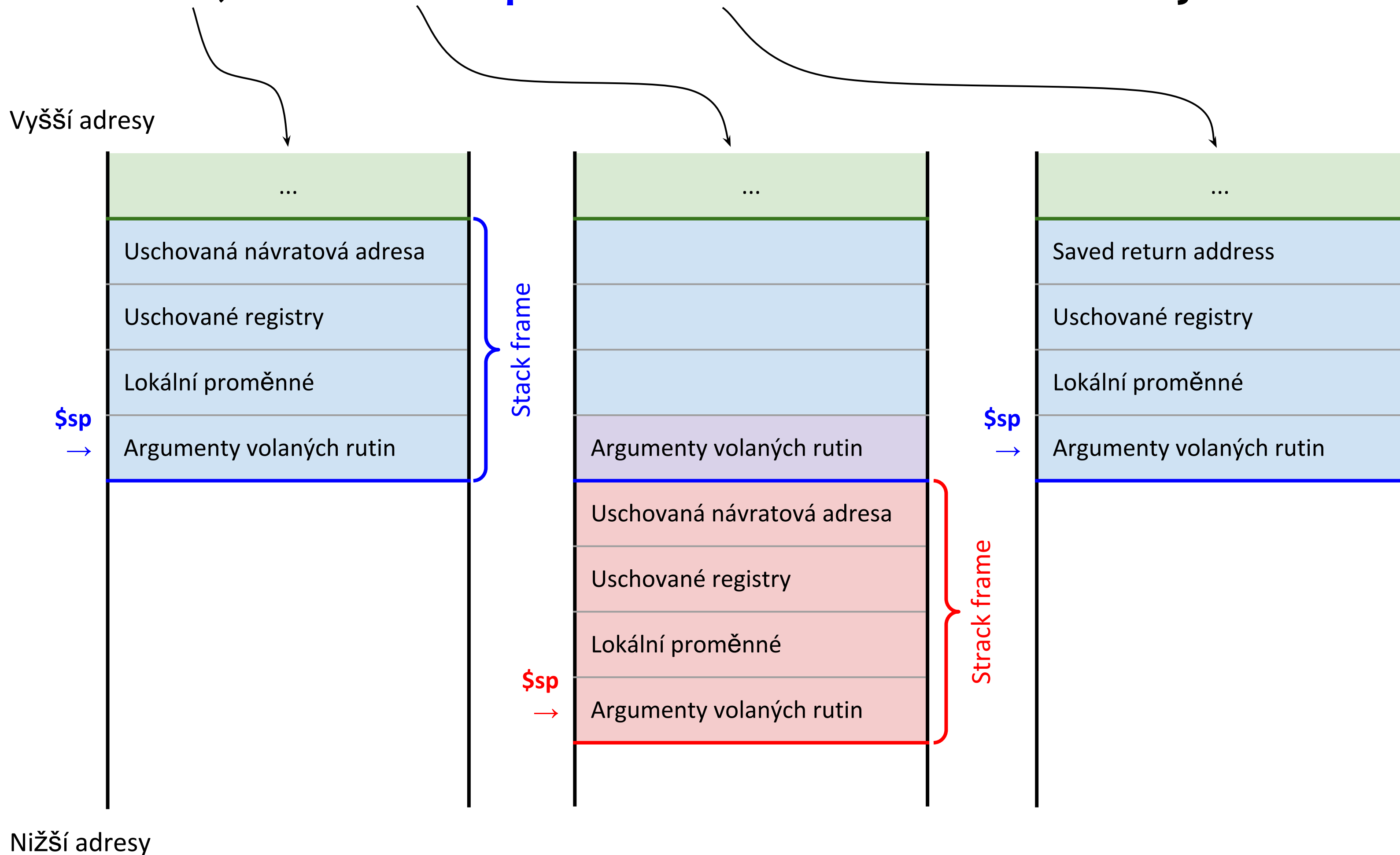
Zásobníková datová struktura (Last In First Out)

- Ukazatel na vrchol zásobníku (stack pointer)
  - Adresa posledního použitého místa v paměti.
- Operace *push* a *pop*
  - Snížit/zvýšit stack pointer, uložit/přečíst hodnotu
- Přístup k datům relativně vůči stack pointeru.
- Umožňuje vnořovat volání rutin.

Alokace místa na zásobníku

Obsah zásobníku a registrů

- Před, během a po návratu z volání rutiny



Volání funkce s použitím zásobníku

Fragment programu

```
s := AddTwo (1, 2);
```

Kód v assembleru MIPS

- Pozn.: Argumenty by normálně byly pouze v registrech.

```
addi $sp, $sp, -40     # Alokace místa na zásobníku (včetně
...                    # místa pro argumenty všech funkcí)
li $a1, 2
sw $a1, 4 ($sp)        # Uložit 2. argument na zásobník
li $a0, 1
sw $a0, 0 ($sp)        # Uložit 1. argument na zásobník
jal AddTwo             # Volání rutiny (jump and link)
...
addi $sp, $sp, 40      # Uvolnění místa na zásobníku
```

Funkce pracující se zásobníkem (1)

Kód funkce AddTwo() v assembleru MIPS

- Pozn.: ukládat `$ra` (`$s0`, `$s1`) není striktně nutné.
- Pozn.: argumenty načteny ze stack frame volajícího.

```
AddTwo:
    addi $sp, $sp, -12          # Alokace místa na zásobníku
    sw $ra, 8 ($sp)             # Uschování návratové adresy
    sw $s1, 4 ($sp)             # Uschování registru $s1
    sw $s0, 0 ($sp)             # Uschování registru $s0

    lw $s0, 12 ($sp)            # Načtení 1. argumentu ze zásobníku
    lw $s1, 16 ($sp)            # Načtení 2. argumentu ze zásobníku
    add $v0, $s0, $s1           # Výpočet návratové hodnoty

    ... pokračování
```

Funkce pracující se zásobníkem (2)

Kód funkce AddTwo() v assembleru MIPS

```
... pokračování

lw $s0, 0 ($sp)          # Obnovit registr $s0
lw $s1, 4 ($sp)          # Obnovit registr $s1
lw $ra, 8 ($sp)          # Obnovit návratovou adresu
addi $sp, $sp, 12        # Uvolnění místa na zásobníku
jr $ra                   # Návrat do místa volání
```

Srovnání se procesory s HW podporu zásobníku

- Stack frame (aktivační záznam) každé funkce je alokován najednou, `$sp` se již po alokaci nemění.
  - Není vytvářen inkrementálně pomocí instrukcí *push*.
- Prostor pro argumenty všech volaných funkcí je součástí aktivačního záznamu → `$sp` se (v rámci funkce) nemění.

Instrukční sada MIPS (1)

Konstantní délka (32 bitů)

- Registrové instrukce (r-type)
  - Aritmetické a logické operace,
    nepřímé skoky (na adresu v registru)
- Instrukce s přímým operandem (i-type)
  - Aritmetické a logické operace, podmíněné skoky, přenos dat.
- Instrukce přímého skoku (j-type)
  - Nepodmíněné skoky na absolutní adresu.

r-type	op (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
i-type	op (6)	rs (5)	rt (5)	signed immediate (16)		
j-type	op (6)	target (26)				

op = operation code, rs = source register, rt = source register/target register/branch condition
rd = destination register, shamt = shift amount, funct = ALU function

Instrukční sada MIPS (2)

Dobrý návrh vyžaduje dobré kompromisy!

- Rozumný počet formátů instrukcí
  - Usnadňuje dekódování a vykonávání instrukcí.
  - Nesmí příliš omezovat sílu instrukční sady.
- Rozumný počet a velikost registrů
  - Rychlé provádění běžných operací s daty v registrech.
  - Čtení z registrů a zápis do registrů nesmí být pomalý.
- Optimalizováno pro stroje
  - Strojový kód bude primárně “psát” stroj, ne člověk.
  - Jednoduché ortogonální operace usnadňují návrh překladače.