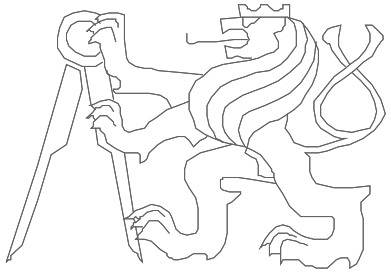
# Pokročilé architektury počítačů

Cesta instrukce a dat z vedlejší paměti až k procesoru z pohledu hierarchie pamětí, operačního systému a procesoru.

VYBRANÉ PARTIE

Poznámky k 1. semestrálnímu projektu



České vysoké učení technické, Fakulta elektrotechnická

# Rozdělení paměti

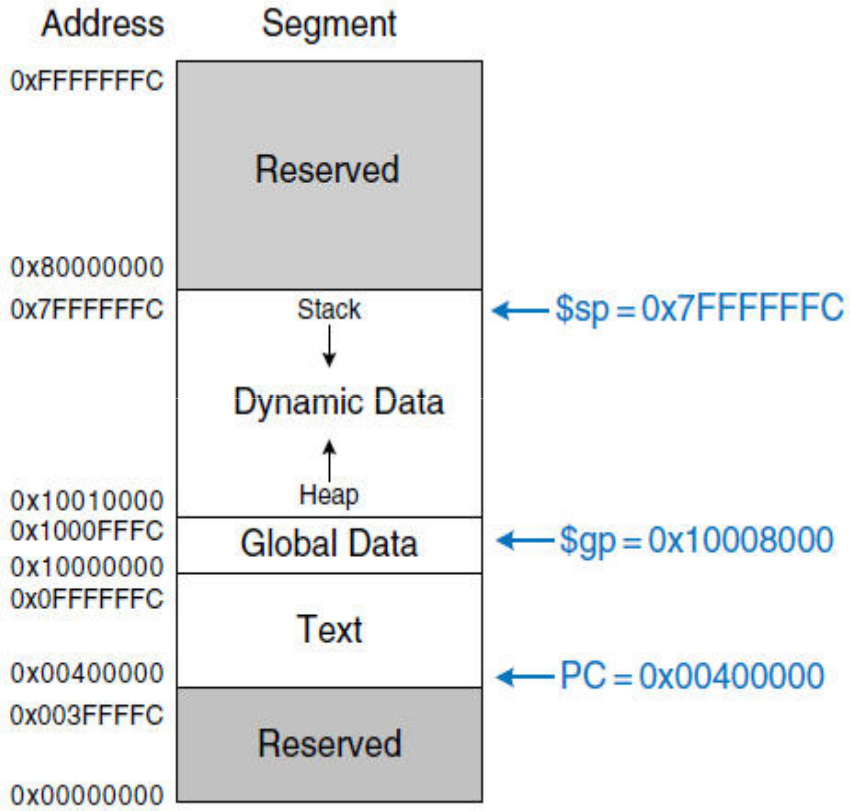
MIPS32: Adresa 32 bitů => 4GB

**Text segment:**
program (256MB): 4 nejvýzn. bity vždy 0…

**Global data segment:**
globální proměnné (64 KB) přístupné přes $gp (inicializován na 0x100080000) a 16-bitový offset

**Dynamic Data Segment:**
Uchovává Stack a Heap (2 GB).
Dynamicky alokováno.

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



# Jednoduchý příklad

**JAL -- *Jump and link***

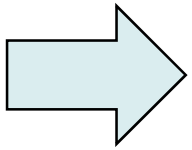
| | |
|---|---|
| Description: | For procedure call. |
| Operation: | $31 = PC + 8; PC = ((PC+4) & 0xf0000000) \| (target << 2) |
| Syntax: | jal target |
| Encoding: | `0000 11ii iiii iiii iiii iiii iiii iiii` |

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

**C/C++**

```
int main() {
  simple();
  …
}

void simple(){
  return;
}
```



**MIPS**

```
0x00400200 main: jal simple
0x00400204 …


0x00401020 simple: jr $ra
```

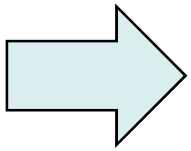
# Co když má funkce něco vracet nebo má argumenty?

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

**C/C++**

```
int main() {
  int y;
  y=fun(2,3,4,5)
  …
}

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



**MIPS**

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

fun:
add $t0, $a0, $a1
add $t1, $a2, $a3
sub $s0, $t0, $t1
add $v0, $s0, $0
jr $ra
```

# Volaný ale nemá modifikovat obsahy registrů…!!!

- Volaný uloží registry na zásobník (stack) před tím než je modifikuje a obnoví je před tím než se vrátí. Podrobněji:
  1. vytvoří prostor na zásobníku pro zálohování registrů (nebo lokálních proměnných)
  2. zálohuje hodnoty pracovních registrů
  3. vykoná vlastní tělo funkce
  4. obnoví registry použitím zásobníku
  5. dealokuje prostor na zásobníku

- Prostrou na zásobníku, který vytvoří procedura hovoříme ***Stack frame***

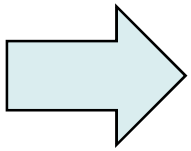
- Na vrchol zásobníku ukazuje ***Stack pointer*** – *registr* ***$sp***

# Volaný ale nemá modifikovat obsahy registrů…!!!

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

**C/C++**

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



**MIPS**

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

# Které registry mám tedy zálohovat?

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

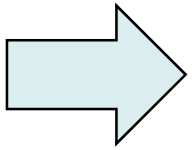
| Preserved | Nonpreserved |
|---|---|
| Saved registers: `$s0-$s7` | Temporary registers: `$t0-$t9` |
| Return address: `$ra` | Argument registers: `$a0-$a3` |
| Stack pointer: `$sp` | Return value registers: `$v0-$v1` |
| Stack above the stack pointer | Stack below the stack pointer |

*callee-save* *caller-save*

# Rekurze?

**C/C++**

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

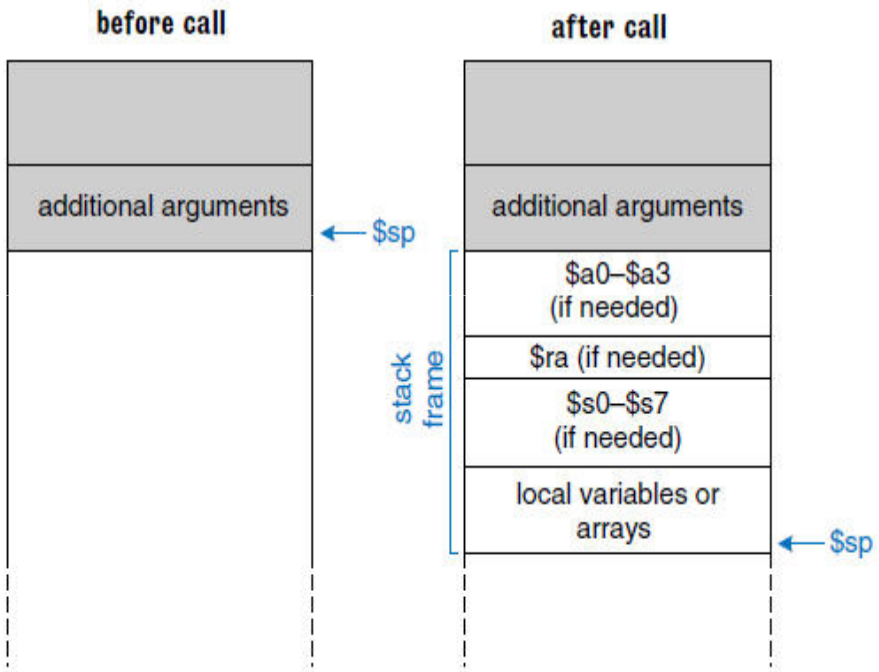


**MIPS**

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

# Co když je parametrů víc než 4?

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




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

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

# x86

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

```
_simple:
pushl %ebp
movl  %esp, %ebp
movl  28(%ebp),%eax
movl  8(%ebp),%edx
movl  %edx, %ecx
subl  %eax, %ecx
movl  %ecx, %eax
popl  %ebp
ret
```

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