Suporte a Funções

- Endereço de retorno é o endereço da instrução após a instrução que muda o fluxo de execução (`jal`)
- endereço de retorno só é conhecido em tempo de execução
- no MIPS o endereço de retorno é sempre armazenado em $31
- a chamada de função no MIPS é

  `jal EnderDaFunção # jump-and-link $31 ← PC+4`

  que faz o salto e carrega o endereço de retorno (PC+4) em $31.
- a última instrução da função deve ser

  `jr $31 # jump-register PC ← $31`

  cujo efeito é copiar o conteúdo de $31 para o PC, retornando para a instrução seguinte à invocação = (PC+4)

Suporte a Funções – exemplo

```
main:
      ...
2000: jal 8000          # salva o ender de retorno em $31
2004: add r16,r14,r2    # ESTE é o ender de retorno de B()
      ...
B:
8000: sw r5,0(sp)
      ...
      jr $31            # salta para endereço de retorno
```

Suporte a Funções – convenções

- Uma função deve salvar em memória registradores que modifica e que são usados pela função que a invocou
- A estrutura de dados onde os registradores são salvos é uma *pilha*
- $31 deve ser salvo na pilha antes que outra função seja invocada
- convenções do MIPS:
  - *caller save*: quem salva os registradores é quem chama
  - *callee save*: quem salva os registradores é a função chamada
  - endereço de retorno (return address) é $31 (ra) hw
  - apontador de pilha (stack pointer) é $29 (sp) sw
  - montador usa $4..$7 para passar 4 parâm em regs a0-a3 sw
  - 5º argumento e seguintes na pilha, antes de chamar função sw
  - valores são retornados em $2 e $3 v0,v1 sw
  - regs $26 e $27 reservados para sistema operacional k0,k1 sw

Uso de Registradores e Valores em funções

	preservados		**destruídos**
s0..s7	salvados	t0..t9	temporários
sp	apontador de pilha	a0..a3	argumentos
ra	endereço de retorno	v0..v1	resultados
gp	*global pointer*	k0..k1	usados pelo SO
		at	*assembler temporary*
	pilha acima do sp		pilha abaixo do sp

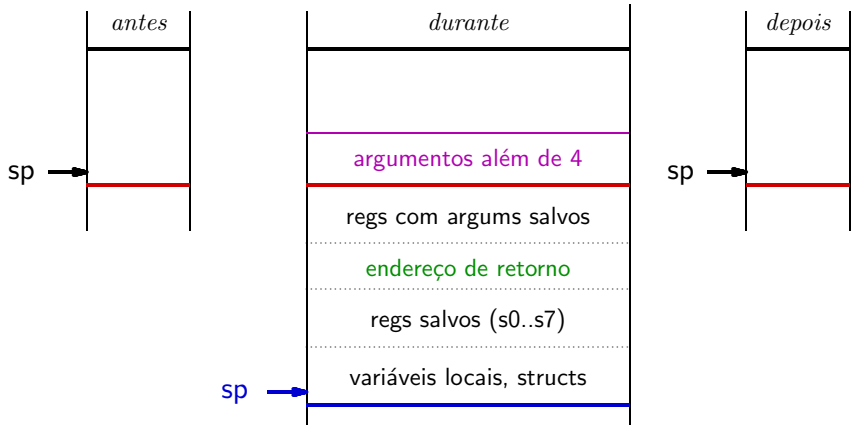
Uso da pilha: quem chama salva **XOR** chamado salva **nunca mistura**

Se início de `f()` move *sp* de $-8n$ posições,

então final de `f()` deve mover *sp* de $+8n$ posições

senão código não pode ser ligado **e** viola princípio de modularidade

Registro de ativação



Complicações de C

Como é codificada esta chamada de função da linguagem C?

```
k = fun(4, 16*x, gun(y,z,w), p, q*r, x*y, s/2);
```

Uso repetido de registradores

Onde são armazenadas as distintas cópias de r18, r19 e r20?

```
main() {              # x,y,z -> r18,r19,r20
   ...
   x = y - z;         sub r18, r19, r20
   ...
   x = B(y, z);       move a0, r19
                      move a1, r20
                      jal B
                      move r18, v0
   ...  }
int B(p, q) {         # p,q,r -> r18,r19,r20
   ...
   p = q & w;         and a0, ...
   q = p | u;         or a1, ...
   r = C(p,q);        jal C
                      move r20, v0
   ...  }
int C(x,y) {          # a,b -> r18,r19
   ...
   a = x + y + b;     add r18, a0, a1
   ...  }             add r18, r18, r19
```

Funções Aninhadas – pilha

```
programa C                PILHA          inicial   0xf000 0004
main() {                  main()x = $8             0xf000 0000
   ...                          y = $9             0xefff fffc
   x = y - z;                   z = $10            0xefff fff8
   ...
   x = B(y, z);
   ...
}
int B(p, q) {             B()       w               0xefff fff4
   ...                              u               0xefff fff0
   p = q & w;                       p = $8          0xefff ffec
   q = p | u;                       q = $9          0xefff ffe8
   r = C(p);                        r = $10         0xefff ffe4
   ...                              ra              0xefff ffe0
   return();
}
int C(x) {                C()             a         0xefff ffdc
   ...                                   x = $8    0xefff ffd8
   x = a + 5;                              ra      0xefff ffd4
   ...; return(x); }
```

Funções Aninhadas – código

```
main: ...
      jal   B              # salta, guarda end de retorno em $31
      add   $16, $14, $2   # ESTE é o endereço de retorno de B()
      ...
B:    addiu $29, $29, -12  # ajusta SP para empilhar 3 pals
      sw    $31, 0($29)    # empilha end para retornar a main()
      sw    $16, 4($29)    # empilha $16 e $17
      sw    $17, 8($29)
      jal   C              # salta, guarda end de retorno em $31
      add   $4, $5, $2     # ESTE é o endereço de retorno de C()
      ...
      lw    $31, 0($29)    # re-carrega end para retornar a main()
      lw    $16, 4($29)    # des-empilha $16 e $17
      lw    $17, 8($29)
      addiu $29, $29, 12   # ajusta stack-pointer
      jr    $31            # retorna para main() (ou outra)
      ...
C:    add   $16,...        # função folha, não salva $31
      ...                  #      não chama outra função
      jr    $31            # retorna para B() (ou outra)
```

Funções Recursivas – exemplo

Codifique e simule a execução e a pilha de:

```
1    int fat(int n) { // como é o registro de ativação
2       if (n==0)     // desta função?
3          return 1;
4       else
5          return (n * fat(n-1));
6    }
```

```
fat(4)  4·fat(3)                    desenrola enquanto n ≠ 0
fat(3)        3·fat(2)
fat(2)              2·fat(1)
fat(1)                    1·fat(0)
fat(0)                          1     n = 0: reenrola
fat(1)                    1·1
fat(2)              2·1
fat(3)        3·2
fat(4)  4·6
```

Mais um exemplo (i)

```
1    typedef struct tripla {
2       int x,y,z;
3    } triplaT;
4    int r;
5    triplaT V[1024];
6       ...
7       r = reduz(V, 1024);
8       ...
```

Mais um exemplo (ii)

```
1    int reduz( triplaT *V, int vSz ) {
2       int i = a = 0;
3       while (i < vSZ) {
4          V[i].z = V[i].x + V[i].y;
5          i *= 2;
6          a += V[i].z;
7       }
8       return(a);
9    }
```