PROJETOS DIGITAIS E MICROPROCESSADORES SUPORTE A FUNÇÕES

Marco A. Zanata Alves

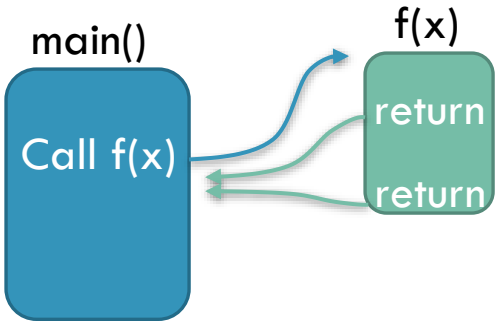
CHAMADA A FUNÇÕES

Como podemos fazer para que nosso programa possa chamar funções?

- Precisamos de alguma forma, sempre retornar para o endereço após a chamada da função.

Como fazer isso funcionar com funções escritas por outros programadores?

- Precisamos de alguma forma, sempre escrever o código respeitando uma certa padronização.



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 a `jal`, que faz o salto e carrega o endereço de retorno (PC+4) em $31

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

A última instrução da função deve ser `jr`, cujo efeito é copiar o conteúdo de $31 para o PC, retornando para a **instrução seguinte à invocação** = (PC+4)

```
jr $31               # jump-register      PC ← $31
```

SUPORTE A FUNÇÕES

```
main:
        ...
2000: jal 8000              # Salva o end. de retorno em $31
2004: add r16,r14,r2        # Este é o end. 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

Os fabricantes de CPU (ou SO) fornecem padrões ou esquemas para os compiladores (Calling conventions)

Se todos compiladores utilizarem o mesmo esquema, as funções de um podem ser utilizadas pelo outro

Essencial para utilização de bibliotecas do SO

caller save: quem salva os registradores é quem chama

callee save: quem salva os registradores é a função chamada (**comum em MIPS**)

SUPORTE A FUNÇÕES – CONVENÇÕES MIPS

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 parâmetros em regs	a0-a3	sw
5 parâmetro 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 - CONVENÇÃO

#	Nome	Função	Preservado
0	$zero	Constante	-
1	$at	Reservado para Assembly	Não
2	$v0	Retorno da Função	Não
3	$v1		
4	$a0	Argumentos da Função	Não
5	$a1		
6	$a2		
7	$a3		
8	$t0	Temporários	Não
...			
15	$t7		

#	Nome	Função	Preservado
16	$s0	Temporários Salvos	Sim
...			
23	$s7		
24	$t8	Temporários	Não
25	$t9		
26	$k0	Reservado para SO/Kernel	-
27	$k1		
28	$gp	Ptr. Área Global	Sim
29	$sp	Ptr. Pilha	Sim
30	$fp	Ptr. Frame	Sim
31	$ra	End. Retorno	-

REGISTROS DE ATIVAÇÃO

A medida que um programa executa e rotinas vão sendo chamadas, a pilha de execução cresce, refletindo as chamadas.

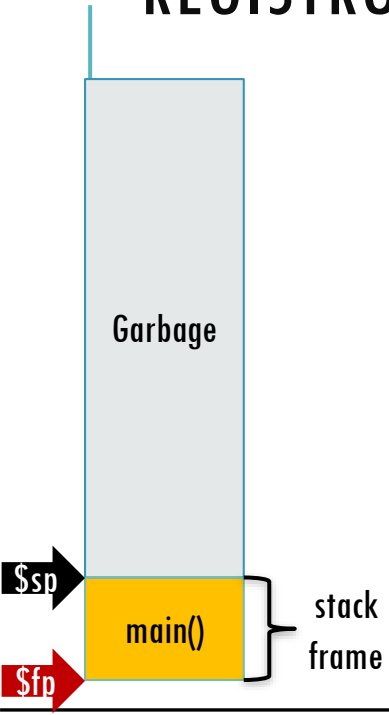
Além dos endereços de retorno, é comum cada procedimento usar a pilha para armazenar variáveis locais e outros temporários.

Por isso, associamos a cada chamada de procedimento uma área da pilha, chamada de registro de ativação.

O registro de ativação é a região da pilha alocada para cada chamada de procedimento.

Para facilitar o acesso aos valores armazenados no registro de ativação, em geral se mantém um ponteiro para o "início" dele.

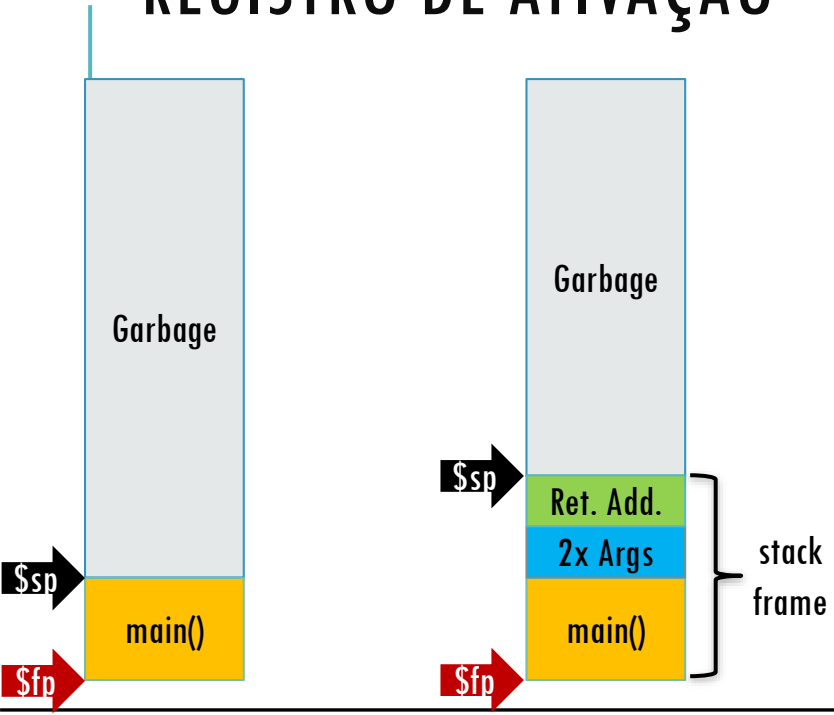
REGISTRO DE ATIVAÇÃO



O espaço reservado da função é chamado de stack frame.

O primeiro elemento é sempre o endereço ponteiro para o registro de ativação anterior

REGISTRO DE ATIVAÇÃO

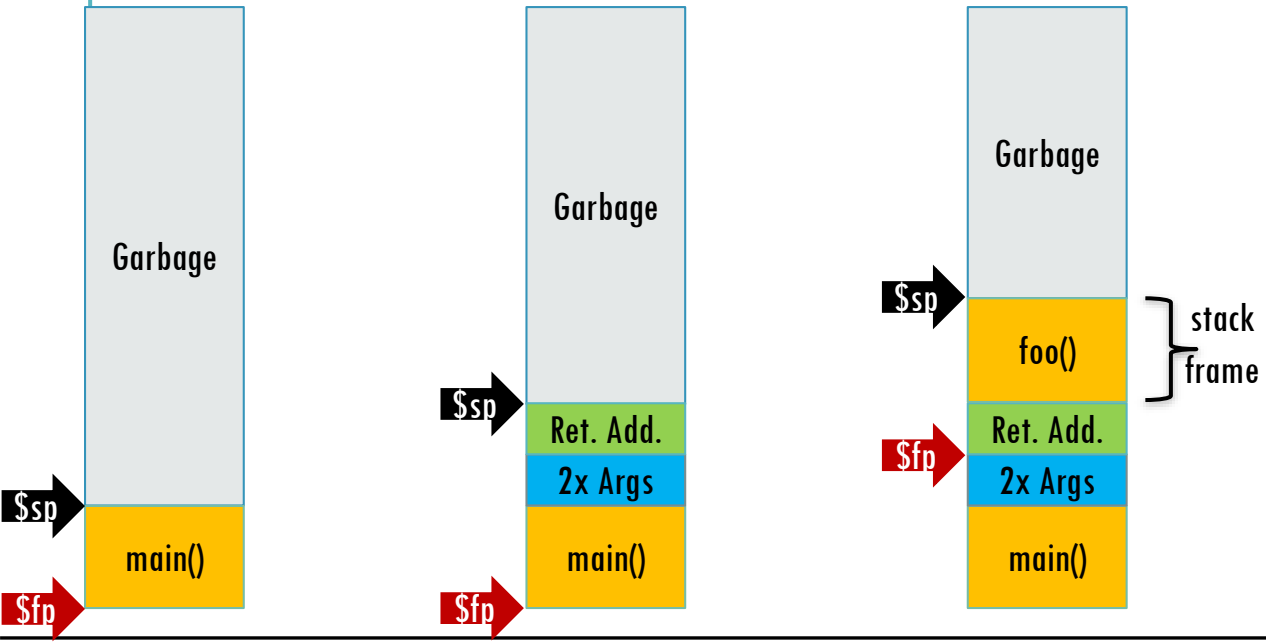


Supondo que main() vai chamar foo() passando 2 argumentos.

Podemos ver o main() aumentando o stack e também o stack frame.

Os últimos elementos sempre são: argumentos da chamada do próximo procedimento aninhado, e endereço de retorno.

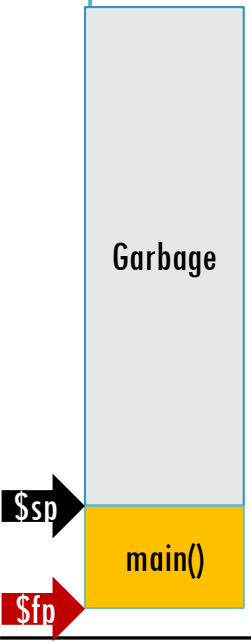
REGISTRO DE ATIVAÇÃO

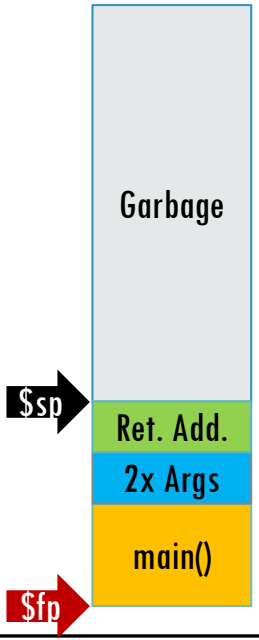


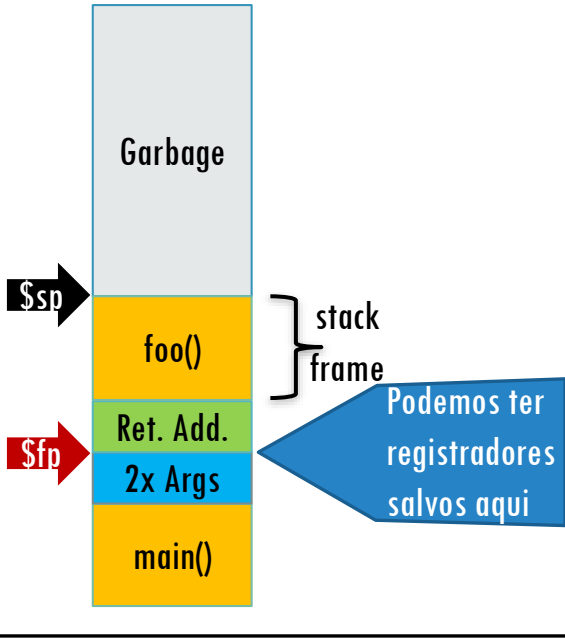
Uma vez que entramos no código de foo() criamos as variáveis locais

A função foo() acessa seus argumentos pois estão no lugar esperado

REGISTRO DE ATIVAÇÃO



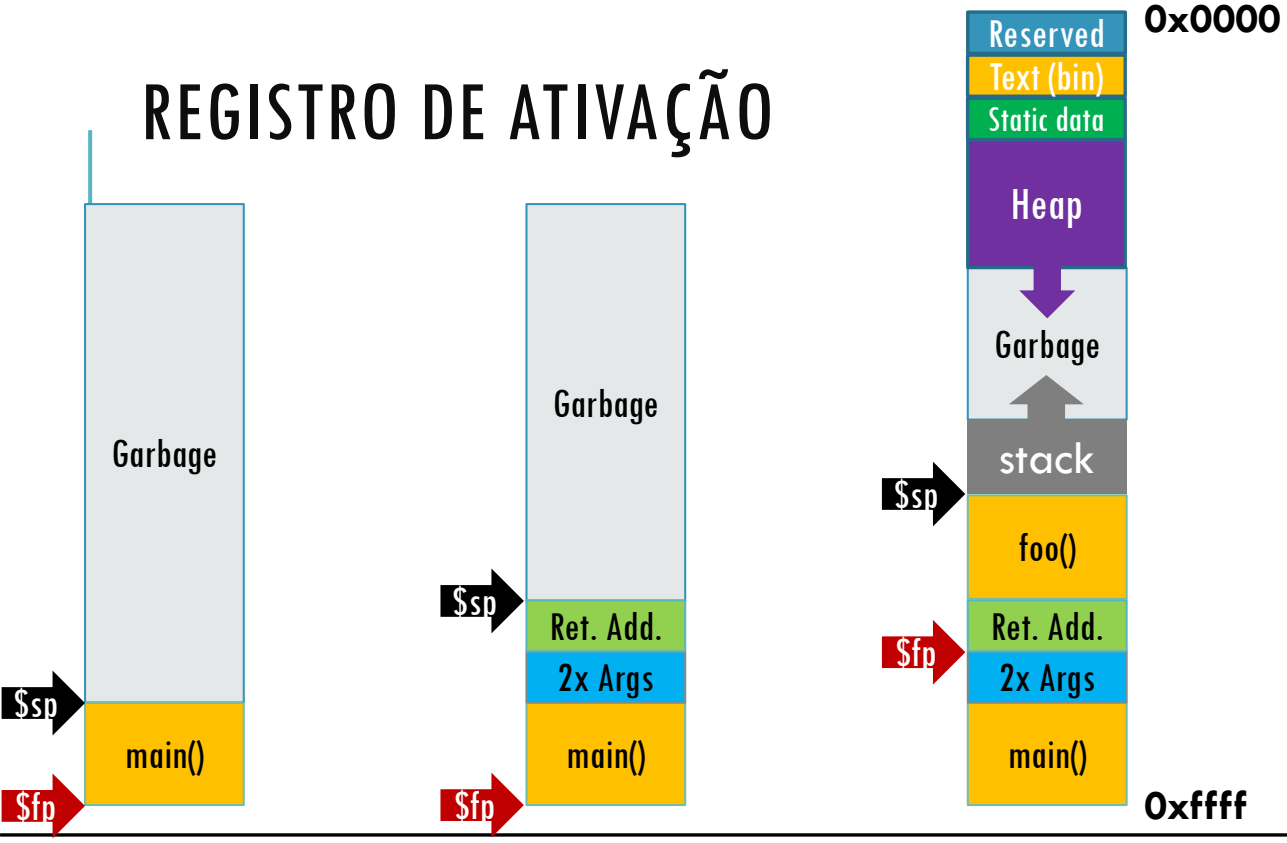




Uma vez que entramos no código de foo() criamos as variáveis locais

A função foo() acessa seus argumentos pois estão no lugar esperado

REGISTRO DE ATIVAÇÃO



Eventuais alocações dinâmicas são alocadas no **heap**, que cresce em direção contrária ao **stack**

REGISTRADORES E MEMÓRIA

Os operandos das instruções aritméticas e lógicas têm de estar em registradores

O compilador associa as variáveis com registradores

Como lidar com programas com muitas variáveis?

- Usar load/store pra transferir dados da memória para os registradores
- Se não houver registradores suficientes?
  Spill (derramamento) de registradores pra memória

Lembre-se, o MIPS é uma arquitetura load/store

REGISTRADORES - DETALHES

Os registradores preservados não serão mudados durante chamadas de funções.

Os registradores `$s0~$s7` devem ser salvos na pilha pela função que irá usa-los.

Os registradores `$t0~$t9` devem ser salvos pelo programa antes de qualquer chamada de função.

Os registradores `$sp` e `$fp` são sempre incrementados durante a chamada de funções e decrementados após a função terminar.

O registrador `$ra` será atualizado automaticamente por qualquer chamada de função normal (usando `jal`)

COMPLICAÇÕES DE C

Como é codificada esta chamada de função da linguagem C?

```
A = f(4, 16*x, (int)sqrt(y*z+w), p, q*r, s=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

```
main() {                          PILHA            inicial 0xf000 0004
  ...                             main() x = $8             0xf000 0000
  x = y - z;                            y = $9             0xefff fffc
  ...                                   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 ffe4
  r = C(p);                             r = $10            0xefff ffe0
  ...
  return();
}
int C(x,y) {                      C()    a                  0xefff ffdc
  ...                                   x = $8             0xefff ffd8
  a = x + 5;                               ra              0xefff ffd4
  ...
  return();
}
```

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) {
2.   if (n==0)
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
```