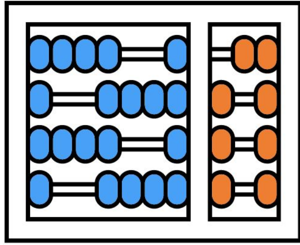


**Instituto de Computação**

UNIVERSIDADE ESTADUAL DE CAMPINAS

# Organização Básica de computadores e linguagem de montagem

## Implementação de rotinas

**Prof. Edson Borin**

https://www.ic.unicamp.br/~edson

Institute of Computing - UNICAMP

# Agenda

- **Chamada e retorno de rotinas**
- A pilha do programa
- ABI: *Application Binary Interface*
- Passagem de parâmetros e retorno de valores
- Variáveis locais
- Políticas de uso de registradores
- Quadro de pilha
- Implementação de rotinas compatíveis com a `ilp32`
- Chamadas de sistema (syscalls)

# Chamada e retorno de rotinas

- "`jal a0, L`": instrução de salto do RISC-V que armazena o endereço da próxima instrução (PC+4) no registrador indicado (`a0`) e depois salta para o rótulo indicado (L).

- "`jal L`": pseudo-instrução que é traduzida para a instrução: "`jal ra, L`".
  - `ra` (*return address*) é o apelido do registrador `x1`.

# Chamada e retorno de rotinas

Exemplo de chamada de subrotina:

```
    ...
    li   a0, 127
    jal  hash         # chama subrotina hash
    addi a1, a2, 1   # continua aqui
    ...
# Retorna o valor da função hash em a0
hash:
    andi a1, a0, 63    # a1 <= a0 & 0x3F
    srli a0, a0, 6     # a0 <= a0 >> 6
    and  a0, a1, a0    # a0 <= a1 & a0
    ret                # jalr x0, ra, 0
```

# Chamada e retorno de rotinas

Exemplo de chama

```
   ...
   li   a0
   jal  hash         # chama subrotina hash
   addi a1, a2, 1   # continua aqui
   ...
# Retorna o valor da função hash em a0
hash:
   andi a1, a0, 63
   srli a0, a0, 6
   and  a0, a1
   ret
```

Grava o endereço da instrução subsequente (`PC+4`) no registrador `ra` e salta para hash.

Salta para o endereço armazenado em `ra` (endereço de retorno).

# Chamada e retorno de rotinas

O que acontece com o valor de `ra` se a função `hash` chamar outra subrotina?

```
  ...
  li  a0, 127
  jal hash          # chama hash
  ...

# Retorna o valor da função hash em a0
hash:
  jal outra_rotina # chama outra_rotina
  ret
```

# Chamada e retorno de rotinas

O que acontece com o valor de `ra` se a função `hash` chamar outra subrotina?

```
  ...
  li  a0, 1
  jal hash
  ...

# Retorna o v     da função hash em a0
hash:
  jal outra_rotina # chama outra_rotina
  ret
```

O valor de `ra` será modificado, inviabilizando o retorno para a instrução subsequente à `jal hash`.

# Chamada e retorno de rotinas

**Solução**

- Salvar o endereço de retorno em ra antes de invocar uma rotina, e
- Recuperar o endereço após o retorno da rotina.

```
hash:
  # Salvar conteúdo de RA
  ...
  # Chamar a outra_rotina
  jal outra_rotina
  # Recuperar conteúdo de RA
  ...
  ret
```

# Chamada e retorno de rotinas

**Solução**

- Salvar o ender[...] invocar uma r[...]
- Recuperar o e[...]a.

O endereço de retorno em `ra` deve ser salvo na **pilha do programa**.

```
hash:
  # Salvar conteúdo de RA
  ...
  # Chamar a outra_rotina
  jal outra_rotina
  # Recuperar conteúdo de RA
  ...
  ret
```

# Agenda

- Chamada e retorno de rotinas
- **A pilha do programa**
- ABI: *Application Binary Interface*
- Passagem de parâmetros e retorno de valores
- Variáveis locais
- Políticas de uso de registradores
- Quadro de pilha
- Implementação de rotinas compatíveis com a `ilp32`
- Chamadas de sistema (syscalls)

# A pilha do programa

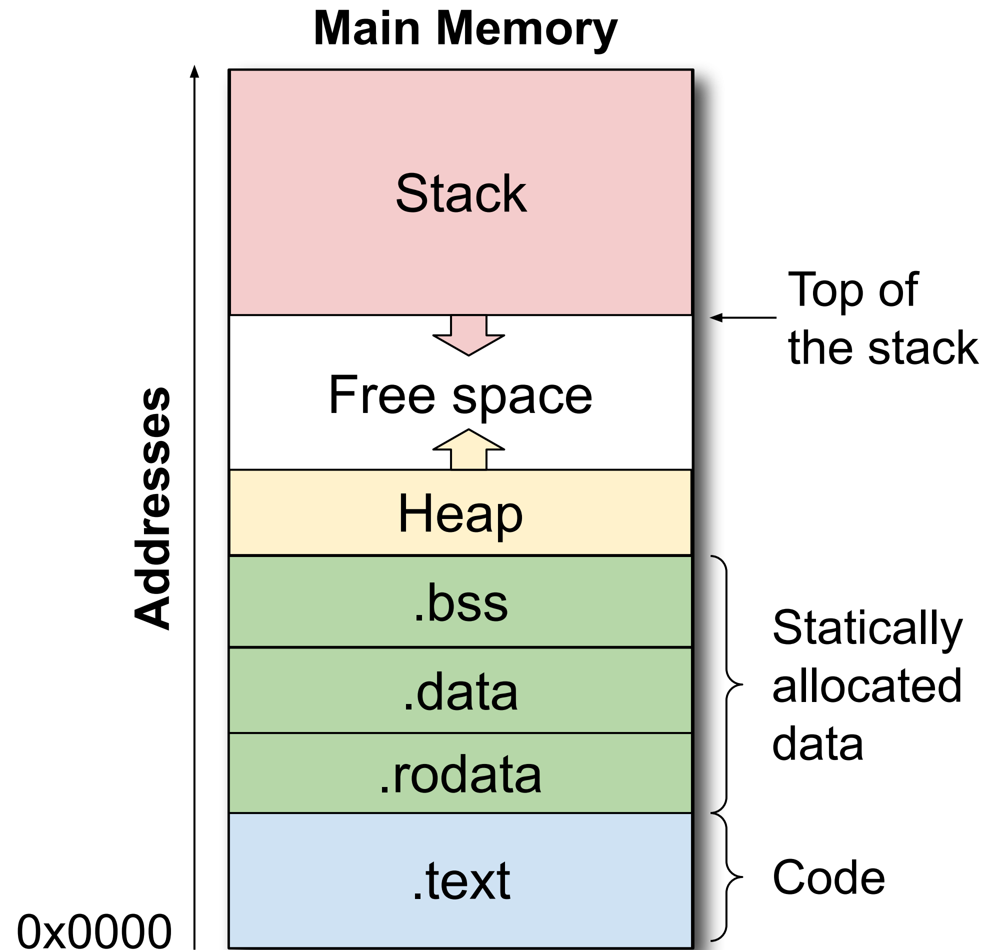
Uma **rotina ativa** é uma rotina que foi chamada mas ainda não retornou.

- Pode haver múltiplas rotinas ativas em cada instante de tempo.
- Os dados de rotinas ativas (endereço de retorno, variáveis locais, ...) são armazenados na pilha:
    - Sempre que uma rotina é invocada, o código da própria rotina aloca espaço na pilha para salvar suas informações;
    - Antes de retornar, o código da própria rotina desaloca o espaço alocado na pilha;

# A pilha do programa

A pilha do programa é utilizada principalmente para guardar valores temporários.

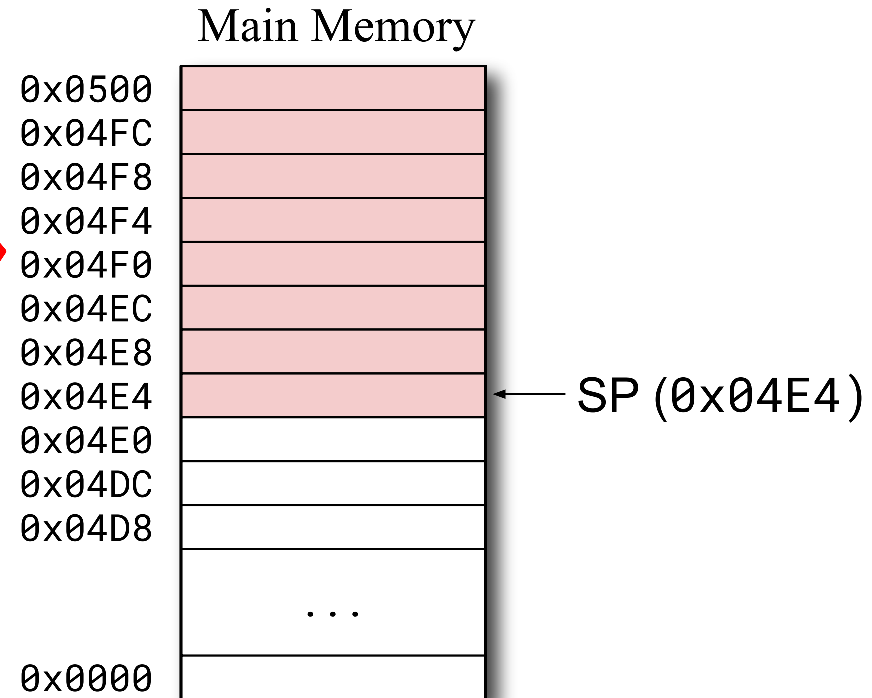
- Armazenada na memória principal
- A pilha geralmente cresce de endereços maiores para menores. Pilha descendente.



# A pilha do programa

O registrador `x2`, ou `SP` (*stack pointer*), aponta para o topo da pilha. Ele deve ser inicializado antes de utilizarmos a pilha.

Neste exemplo, a base da pilha está no endereço `0x0500` e o topo no endereço `0x04E4`. (`SP` contém o valor `0x04E4`)

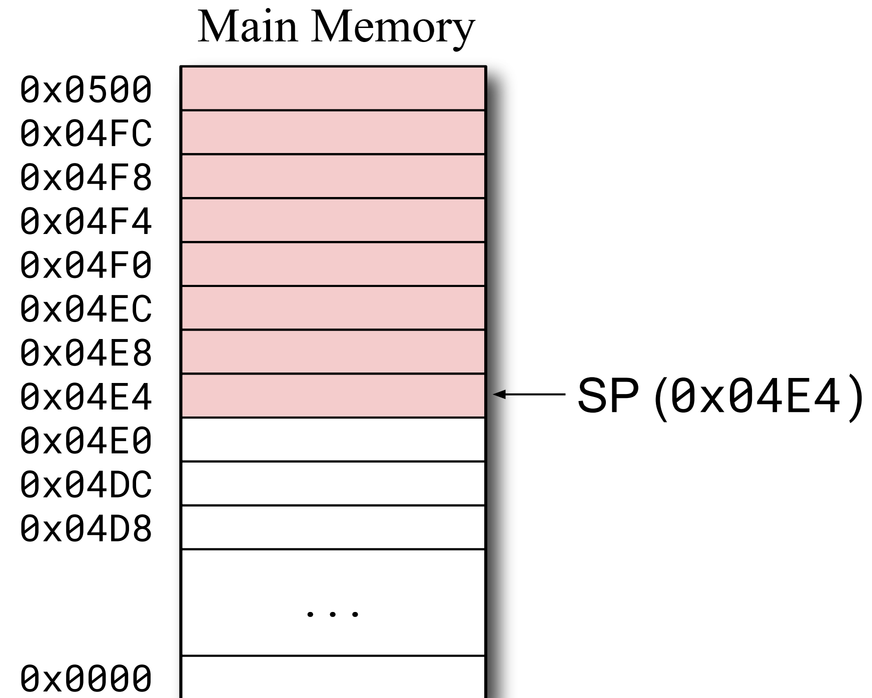


# A pilha do programa

O registrador `x2`, ou `SP` (*stack pointer*), aponta para o topo da pilha. Ele deve ser inicializado antes de utilizarmos a pilha.

Empilhar valor de `a0`

```
addi  sp, sp, -4
sw    a0, 0(sp)
```

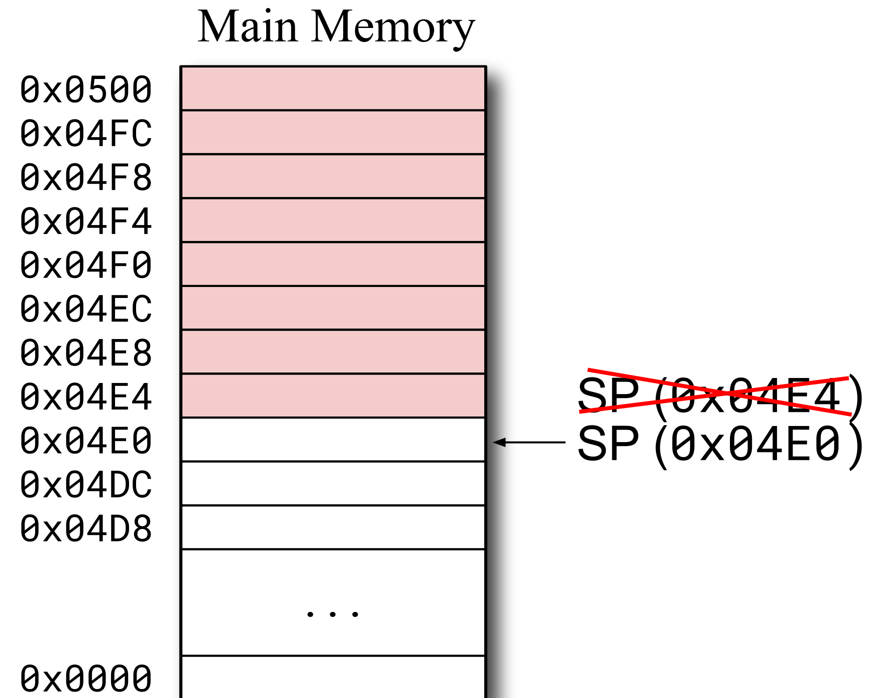


# A pilha do programa

O registrador `x2`, ou `SP` (*stack pointer*), aponta para o topo da pilha. Ele deve ser inicializado antes de utilizarmos a pilha.

Empilhar valor de `a0`

```
addi sp, sp, -4
sw   a0, 0(sp)
```

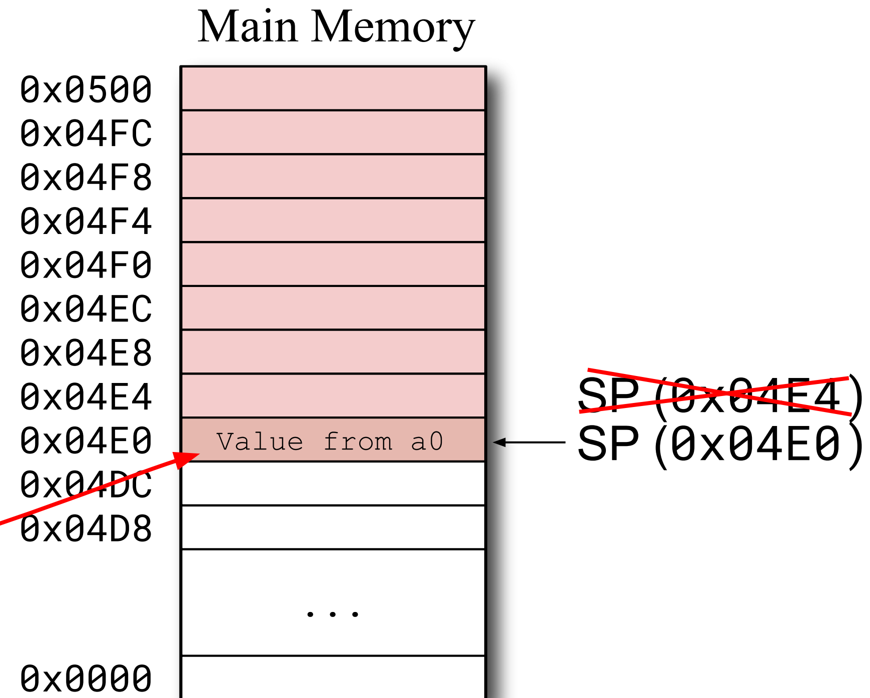


# A pilha do programa

O registrador `x2`, ou `SP` (*stack pointer*), aponta para o topo da pilha. Ele deve ser inicializado antes de utilizarmos a pilha.

Empilhar valor de `a0`

```
addi  sp, sp, -4
sw    a0, 0(sp)
```

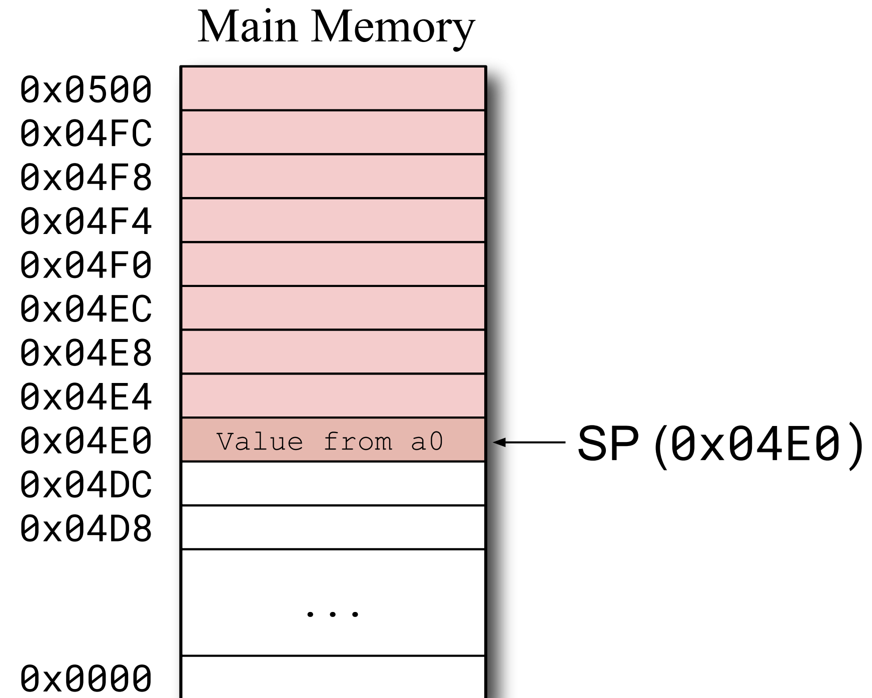


# A pilha do programa

O registrador `x2`, ou `SP` (*stack pointer*), aponta para o topo da pilha. Ele deve ser inicializado antes de utilizarmos a pilha.

Empilhar valor de `a0`

```
addi  sp, sp, -4
sw    a0, 0(sp)
```



# A pilha do programa

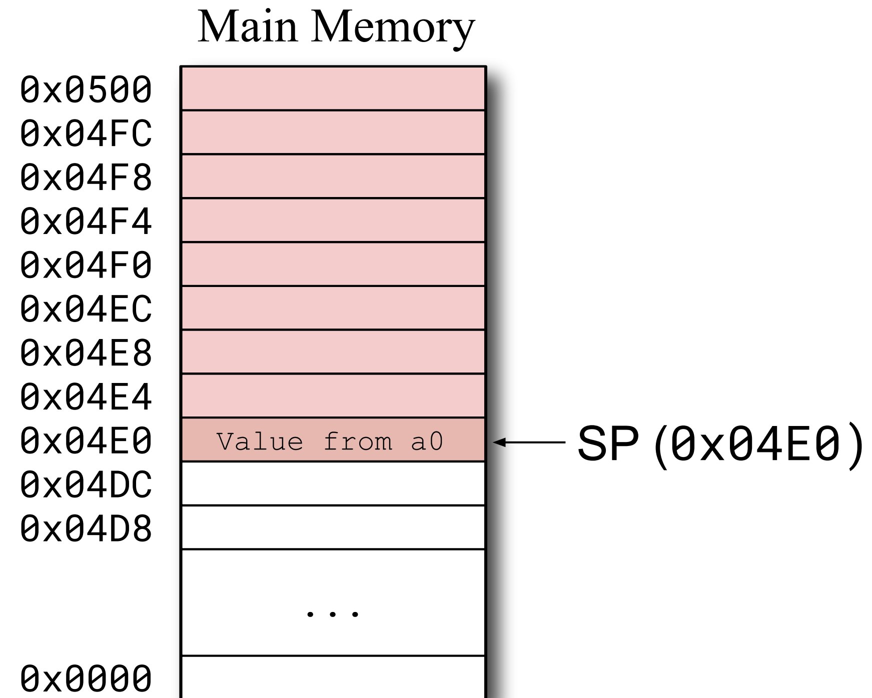
O registrador `x2`, ou `SP` (*stack pointer*), aponta para o topo da pilha. Ele deve ser inicializado antes de utilizarmos a pilha.

Empilhar valor de `a0`

```
addi sp, sp, -4
sw   a0, 0(sp)
```

Desempilhar valor em `a1`

```
lw   a1, 0(sp)
addi sp, sp, 4
```



# A pilha do programa

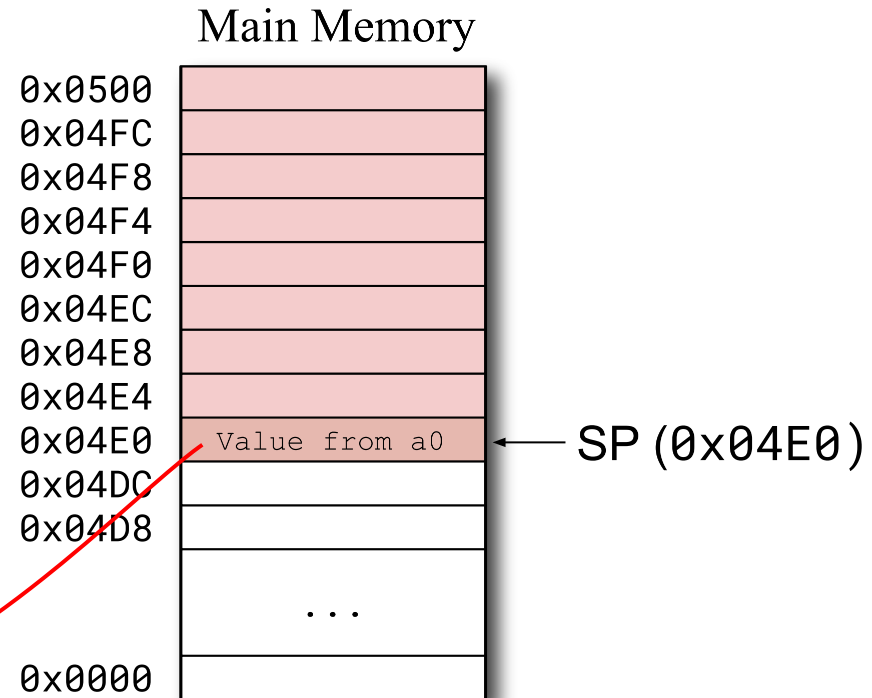
O registrador `x2`, ou `SP` (*stack pointer*), aponta para o topo da pilha. Ele deve ser inicializado antes de utilizarmos a pilha.

Empilhar valor de `a0`

```
addi  sp, sp, -4
sw    a0, 0(sp)
```

Desempilhar valor em `a1`

```
lw    a1, 0(sp)
addi  sp, sp, 4
```



# A pilha do programa

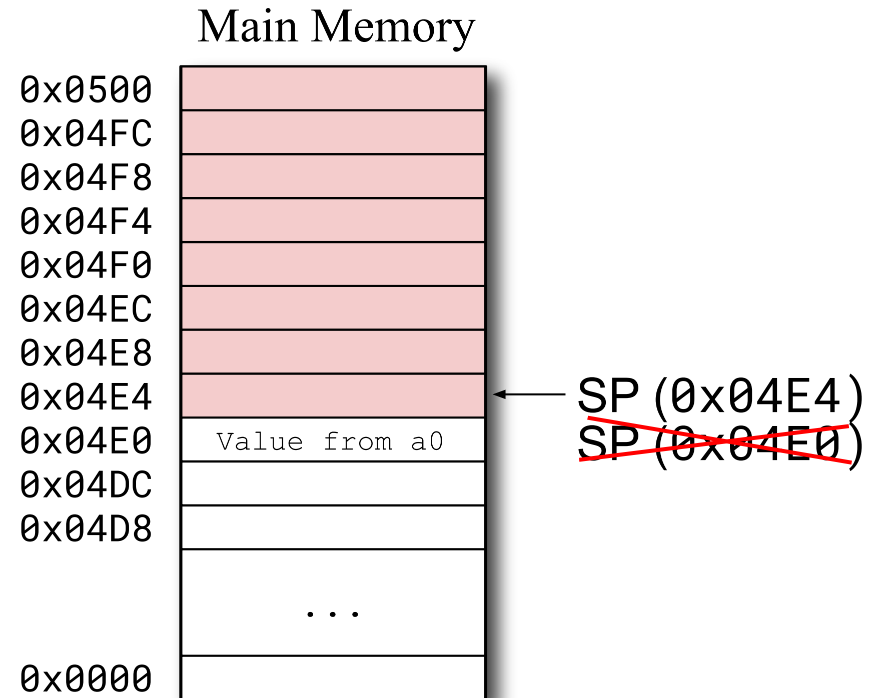
O registrador `x2`, ou `SP` (*stack pointer*), aponta para o topo da pilha. Ele deve ser inicializado antes de utilizarmos a pilha.

Empilhar valor de `a0`

```
addi sp, sp, -4
sw   a0, 0(sp)
```

Desempilhar valor em `a1`

```
lw   a1, 0(sp)
addi sp, sp, 4
```



# A pilha do programa

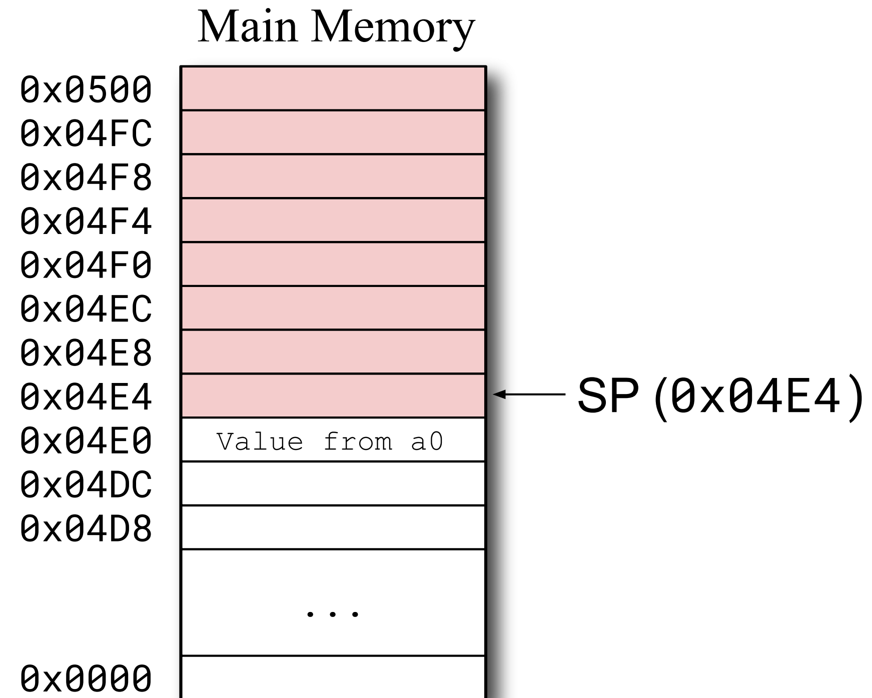
O registrador `x2`, ou `SP` (*stack pointer*), aponta para o topo da pilha. Ele deve ser inicializado antes de utilizarmos a pilha.

Empilhar valor de `a0`

```
addi sp, sp, -4
sw   a0, 0(sp)
```

Desempilhar valor em `a1`

```
lw   a1, 0(sp)
addi sp, sp, 4
```



# A pilha do programa

**Pilha descendente**: A pilha cresce do maior endereço para o menor.

**Pilha ascendente**: A pilha cresce do menor endereço para o maior.

# A pilha descendente

**Pilha cheia**: O `SP` aponta para um dado que está no topo pilha.

- Empilhar: Decrementa `SP` e depois armazena o dado
- Desempilhar: Lê o dado e depois incrementa `SP`.

**Pilha vazia**: O `SP` aponta para a posição subsequente à do dado que está no topo da pilha (esta posição está vazia).

- Empilhar: Escreve o dado e depois decrementa `SP`
- Desempilhar: Incrementa `SP` e depois lê o dado

# A pilha descendente-cheia

- A pilha padrão do RISC-V é **descendente-cheia**, ou *Full-Descendant*.

Empilhar valor de `a0`

```
addi sp, sp, -4
sw   a0, 0(sp)
```

Aloca espaço na pilha

Salva valor de `a0`

Desempilhar valor em `a1`

```
lw   a1, 0(sp)
addi sp, sp, 4
```

Recupera valor de `a0`

Desaloca espaço da pilha

# (Des)empilhando múltiplos valores

Empilhando múltiplos valores

```
addi sp, sp, -12
sw   a0, 8(sp)
sw   a1, 4(sp)
sw   a2, 0(sp)
```

Aloca espaço na pilha

Salva valores de `a0`, `a1` e `a2` na pilha

# (Des)empilhando múltiplos valores

Empilhando múltiplos valores

```
addi sp, sp, -12
sw   a0, 8(sp)
sw   a1, 4(sp)
sw   a2, 0(sp)
```

Aloca espaço na pilha

Salva valores de a0, a1 e a2 na pilha

Desempilhando múltiplos valores

```
lw   a2, 0(sp)
lw   a1, 4(sp)
lw   a0, 8(sp)
addi sp, sp, 12
```

Recupera valores de a2, a1 e a0 da pilha

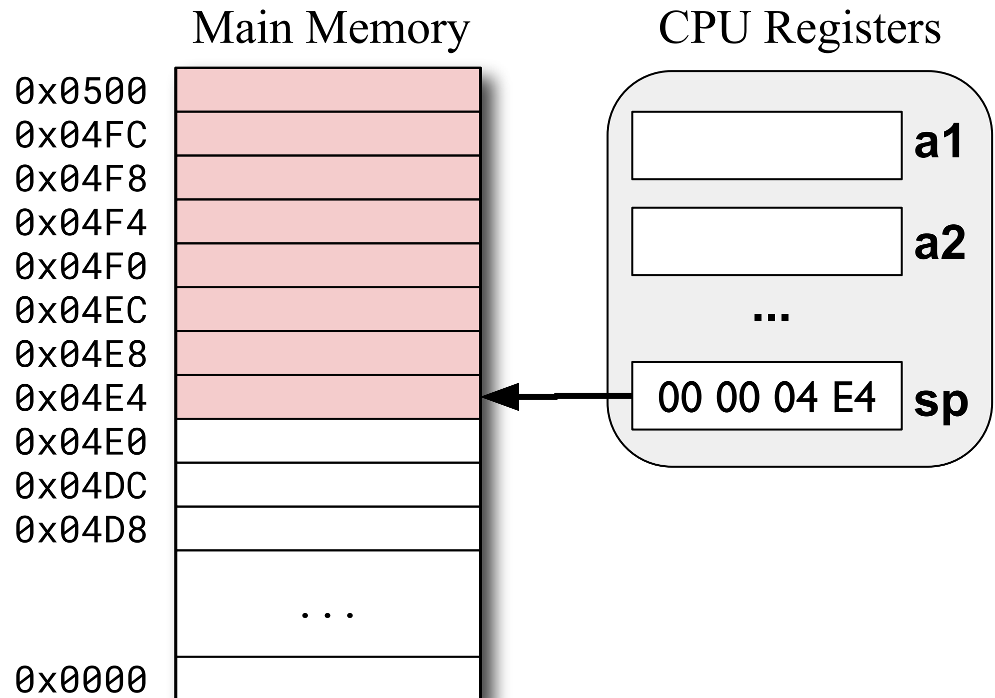
Desaloca espaço da pilha

# Exercício

```
li   a1, 1
li   a2, 2
addi sp, sp, -4
sw   a1, 0(sp)
addi sp, sp, -4
sw   a2, 0(sp)
li   a1, 0
li   a2, 0
lw   a1, 0(sp)
addi sp, sp, 4
lw   a2, 0(sp)
addi sp, sp, 4
# Qual o valor de a1 e a2 após a execução da
# última instrução?
```

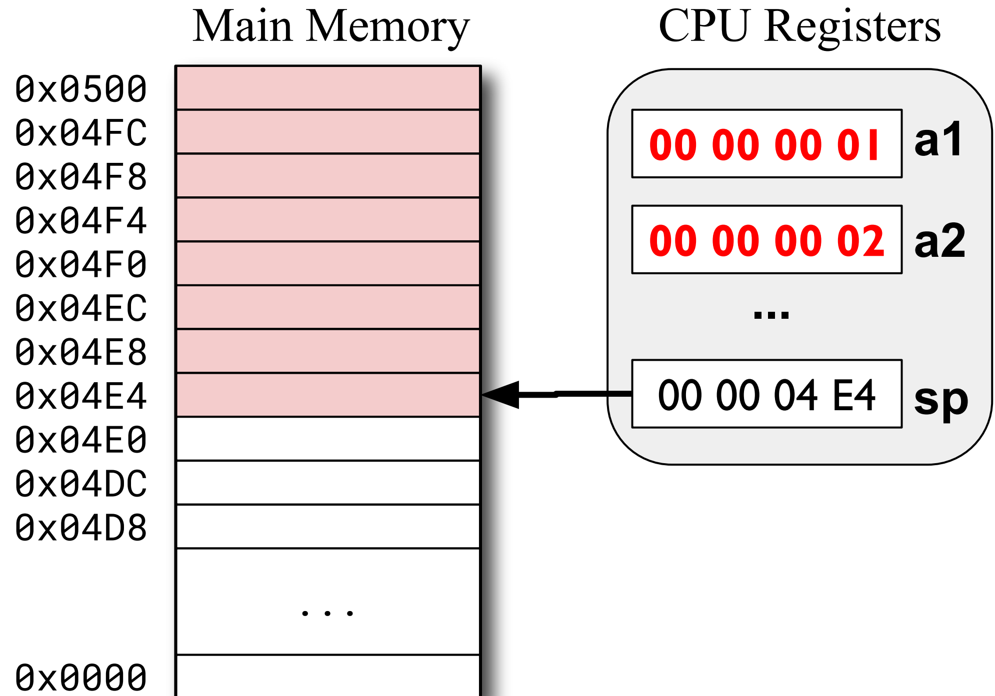
# Exercício

```
li   a1, 1
li   a2, 2
addi sp, sp, -4
sw   a1, 0(sp)
addi sp, sp, -4
sw   a2, 0(sp)
li   a1, 0
li   a2, 0
lw   a1, 0(sp)
addi sp, sp, 4
lw   a2, 0(sp)
addi sp, sp, 4
# Qual o valor de a1 e a2 após a execução da
# última instrução?
```



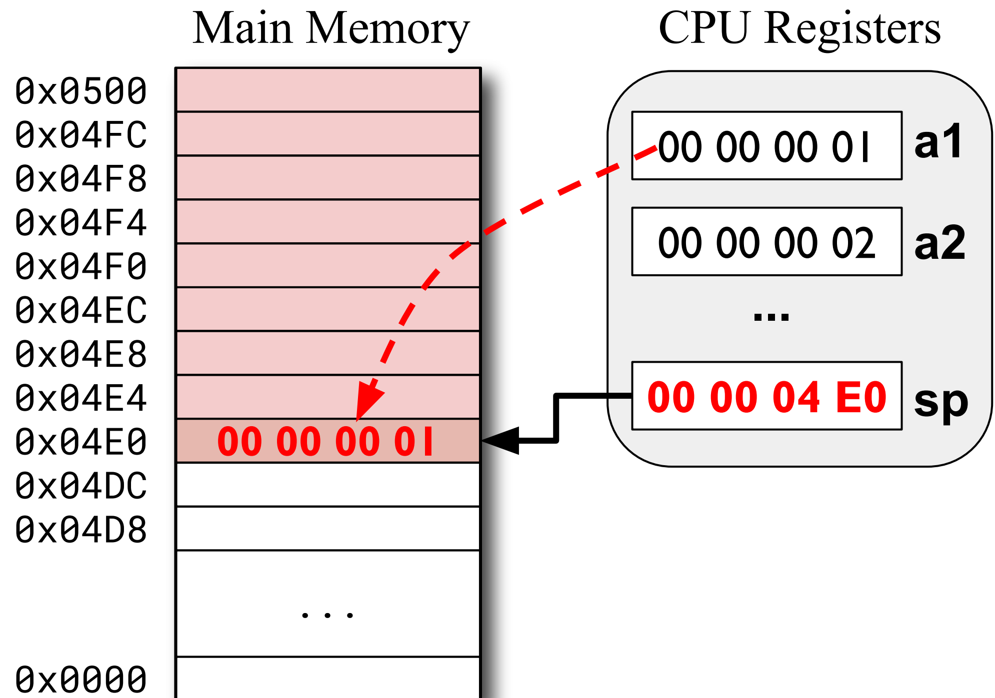
# Exercício

```
li    a1, 1
li    a2, 2
addi sp, sp, -4
sw    a1, 0(sp)
addi sp, sp, -4
sw    a2, 0(sp)
li    a1, 0
li    a2, 0
lw    a1, 0(sp)
addi sp, sp, 4
lw    a2, 0(sp)
addi sp, sp, 4
# Qual o valor de a1 e a2 após a execução da
# última instrução?
```



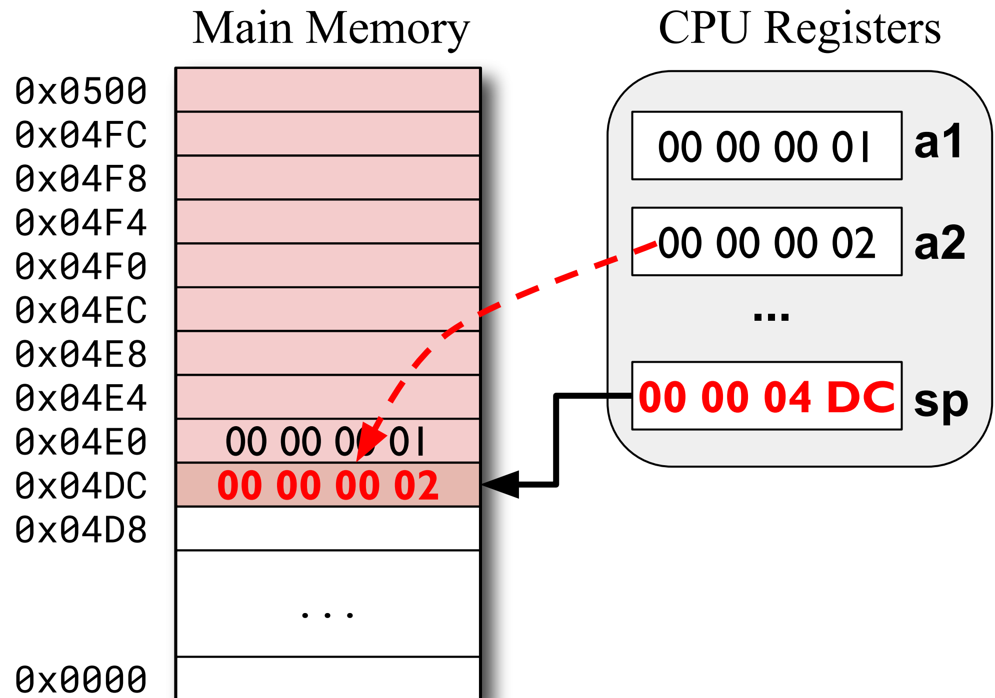
# Exercício

```
li   a1, 1
li   a2, 2
addi sp, sp, -4
sw   a1, 0(sp)
addi sp, sp, -4
sw   a2, 0(sp)
li   a1, 0
li   a2, 0
lw   a1, 0(sp)
addi sp, sp, 4
lw   a2, 0(sp)
addi sp, sp, 4
# Qual o valor de a1 e a2 após a execução da
# última instrução?
```



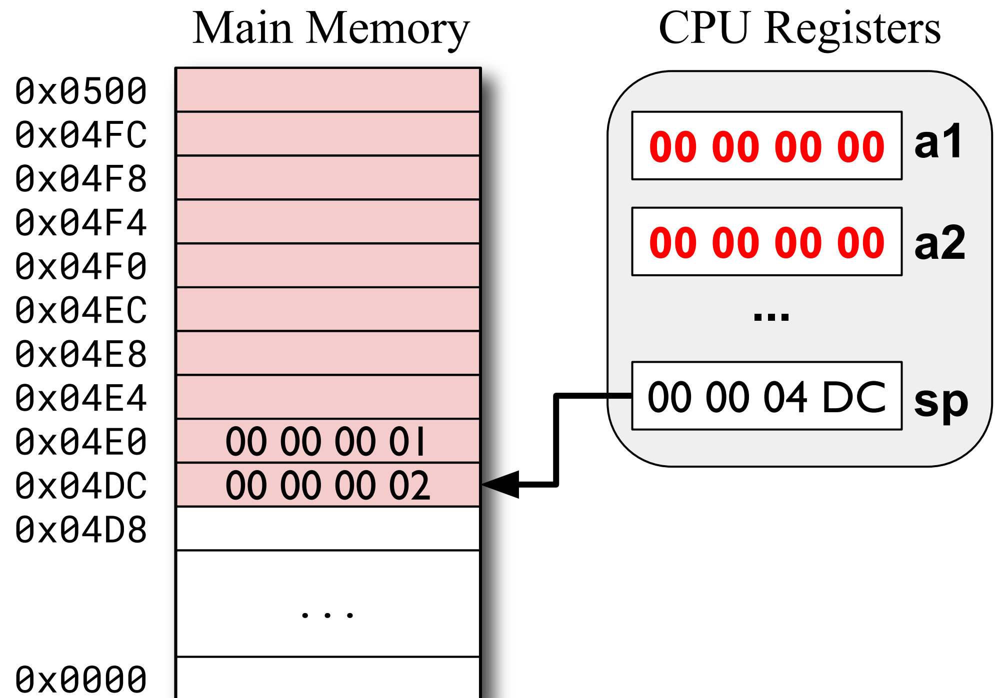
# Exercício

```
li   a1, 1
li   a2, 2
addi sp, sp, -4
sw   a1, 0(sp)
addi sp, sp, -4
sw   a2, 0(sp)
li   a1, 0
li   a2, 0
lw   a1, 0(sp)
addi sp, sp, 4
lw   a2, 0(sp)
addi sp, sp, 4
# Qual o valor de a1 e a2 após a execução da
# última instrução?
```



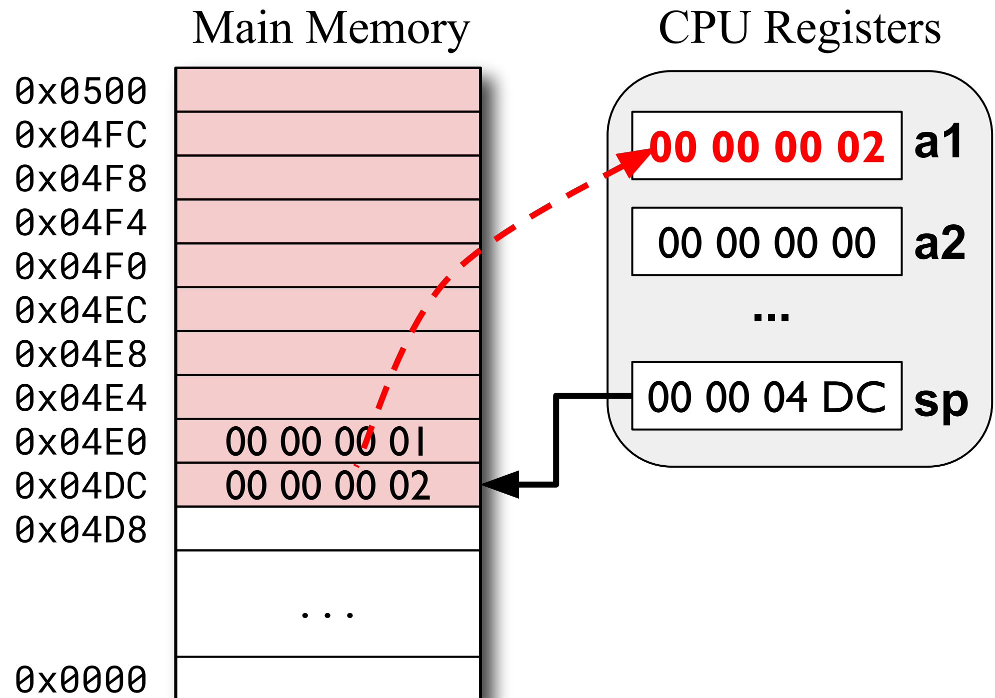
# Exercício

```
li   a1, 1
li   a2, 2
addi sp, sp, -4
sw   a1, 0(sp)
addi sp, sp, -4
sw   a2, 0(sp)
li   a1, 0
li   a2, 0
lw   a1, 0(sp)
addi sp, sp, 4
lw   a2, 0(sp)
addi sp, sp, 4
# Qual o valor de a1 e a2 após a execução da
# última instrução?
```



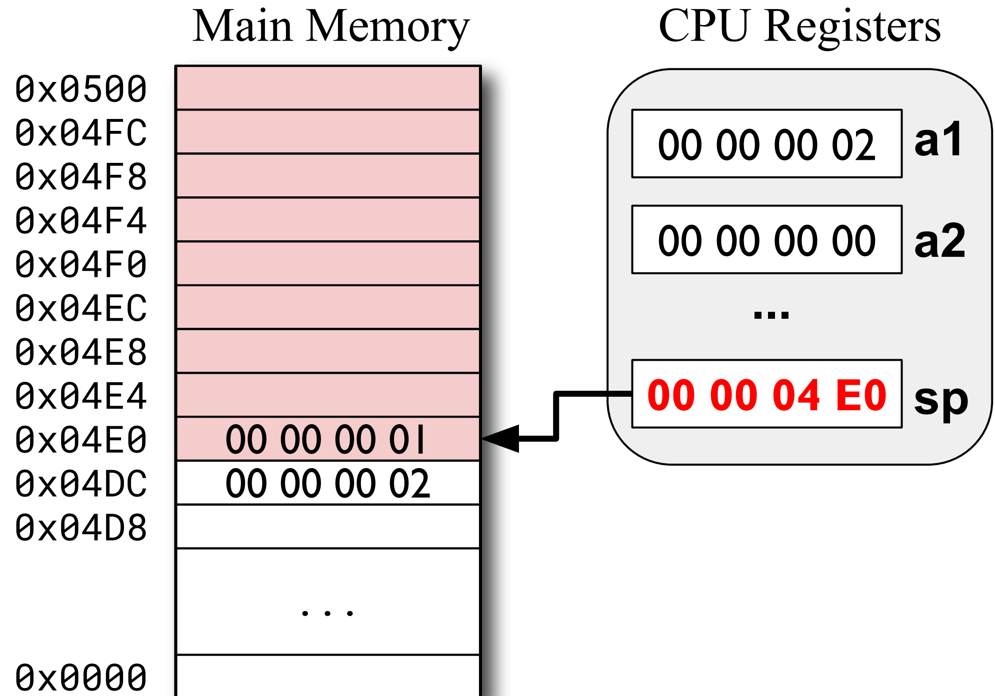
# Exercício

```
li   a1, 1
li   a2, 2
addi sp, sp, -4
sw   a1, 0(sp)
addi sp, sp, -4
sw   a2, 0(sp)
li   a1, 0
li   a2, 0
lw   a1, 0(sp)
addi sp, sp, 4
lw   a2, 0(sp)
addi sp, sp, 4
# Qual o valor de a1 e a2 após a execução da
# última instrução?
```



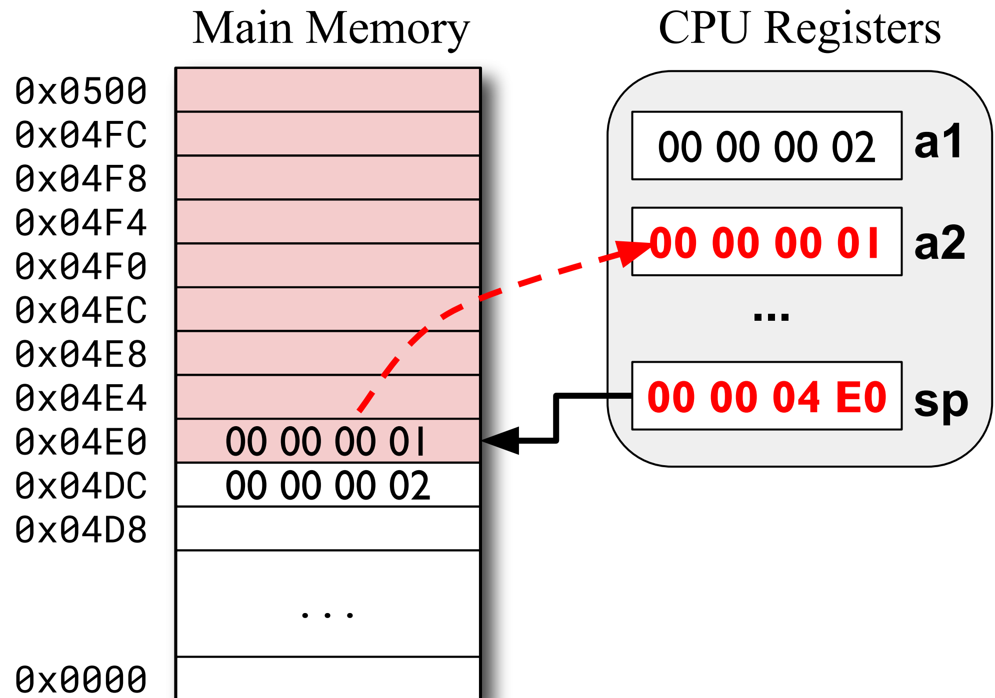
# Exercício

```
li   a1, 1
li   a2, 2
addi sp, sp, -4
sw   a1, 0(sp)
addi sp, sp, -4
sw   a2, 0(sp)
li   a1, 0
li   a2, 0
lw   a1, 0(sp)
addi sp, sp, 4
lw   a2, 0(sp)
addi sp, sp, 4
# Qual o valor de a1 e a2 após a execução da
# última instrução?
```



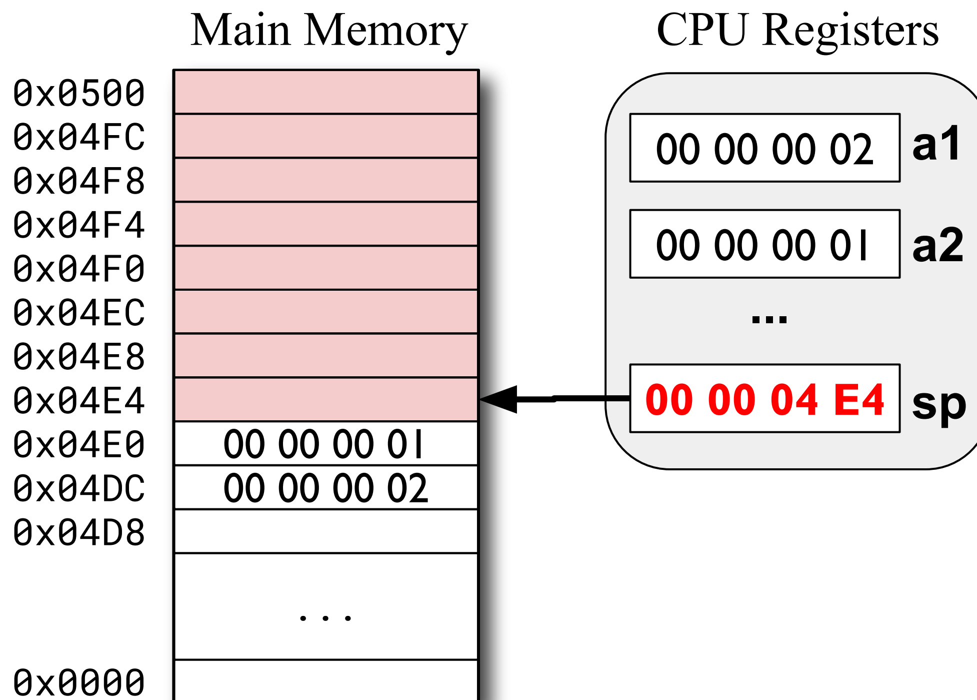
# Exercício

```
li   a1, 1
li   a2, 2
addi sp, sp, -4
sw   a1, 0(sp)
addi sp, sp, -4
sw   a2, 0(sp)
li   a1, 0
li   a2, 0
lw   a1, 0(sp)
addi sp, sp, 4
lw   a2, 0(sp)
addi sp, sp, 4
# Qual o valor de a1 e a2 após a execução da
# última instrução?
```



# Exercício

```
li   a1, 1
li   a2, 2
addi sp, sp, -4
sw   a1, 0(sp)
addi sp, sp, -4
sw   a2, 0(sp)
li   a1, 0
li   a2, 0
lw   a1, 0(sp)
addi sp, sp, 4
lw   a2, 0(sp)
addi sp, sp, 4
# Qual o valor de a1 e a2 após a execução da
# última instrução?
```



# Salvando o endereço de retorno

Exemplo:

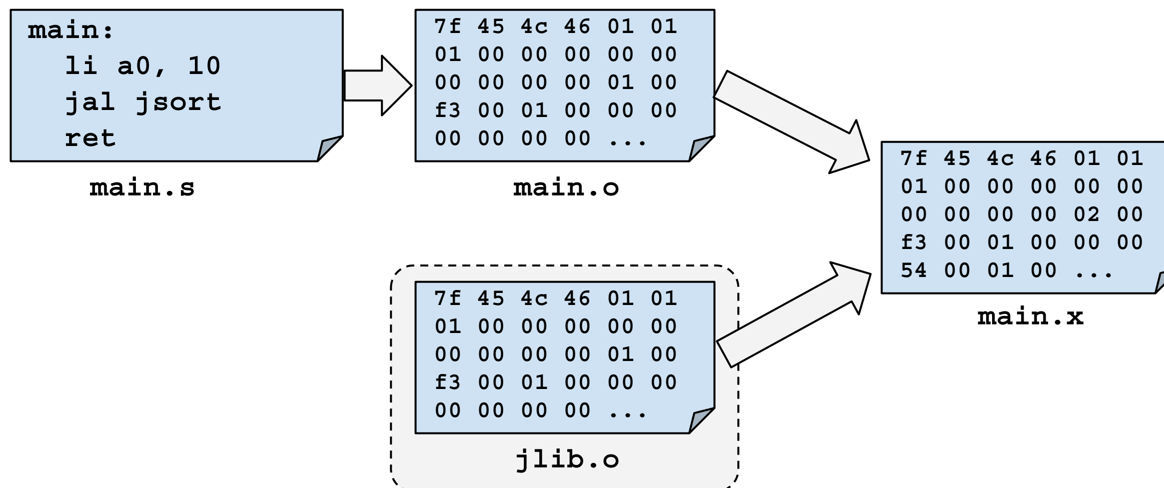
```
hash:
  addi sp, sp, -4  # Salvar conteúdo de RA
  sw   ra, (sp)    # na pilha do programa
  ...
  jal outra_rotina # Chamar a outra_rotina
  ...
  lw   ra, (sp)    # Recuperar conteúdo de
  addi sp, sp, 4   # RA da pilha
  ret              # Retornar
```

# Agenda

- Chamada e retorno de rotinas
- A pilha do programa
- **ABI: *Application Binary Interface***
- Passagem de parâmetros e retorno de valores
- Variáveis locais
- Políticas de uso de registradores
- Quadro de pilha
- Implementação de rotinas compatíveis com a `ilp32`
- Chamadas de sistema (syscalls)

# ABI: *Application Binary Interface*

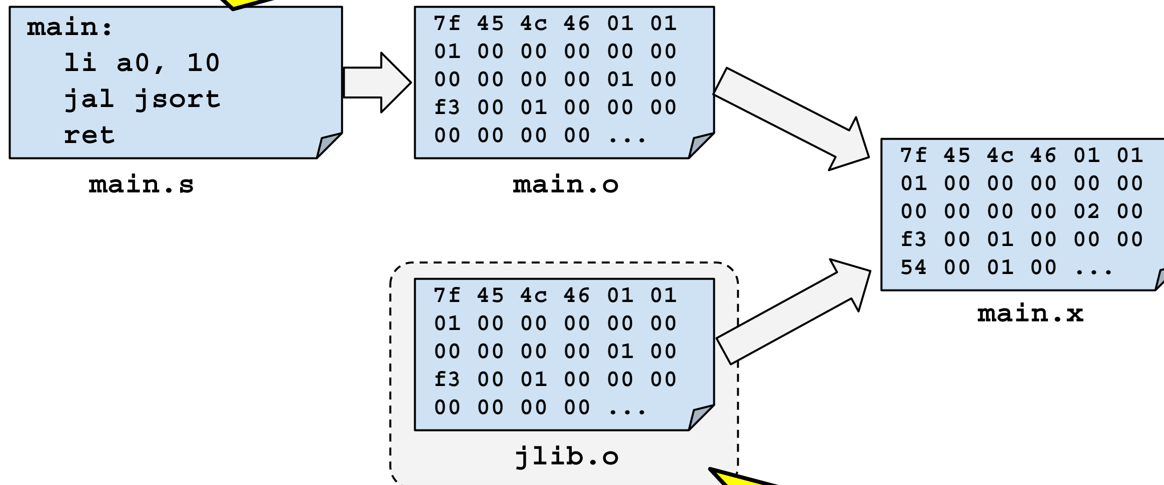
## O programa da Maria



# ABI: *Application Binary Interface*

- O p

O programa da Maria chama a rotina
`void jsort(char* a, int n)`

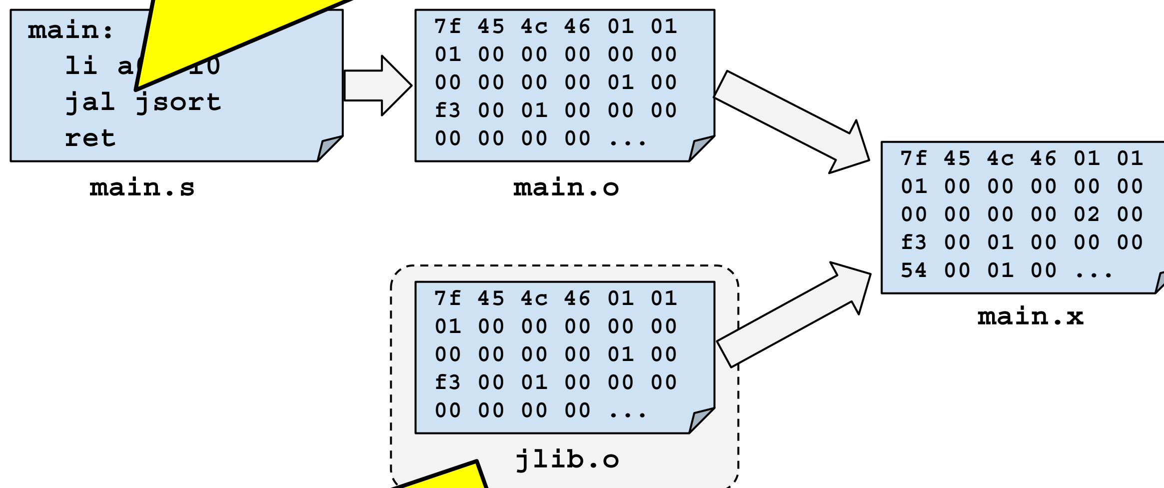



A biblioteca do João (`jlib`) contém a rotina
`void jsort(char* a, int n)`

# ABI: *Application Binary Interface*

O p

A Maria pode chamar a rotina identificada pelo símbolo `jsort` com a instrução `jal`.




Mas onde a Maria passa os parâmetros `a` e `n` para a rotina `jsort`?

`void jsort(char* a, int n)`

# ABI: *Application Binary Interface*

ABI

- Conjunto de regras que definem a interface entre binários.
- Convenção de chamada (*calling convention*):
  - Define onde os parâmetros devem ser passados e onde os valores devem ser retornados.
  - Definida pela ABI.

Focaremos na ABI **RISC-V ilp32**

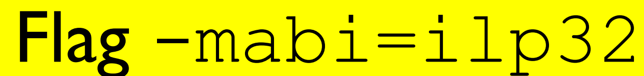
https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md

# ABI: *Application Binary Interface*

Gerando código compatível com a **RISC-V ilp32**

Compilador GCC:

```
$ riscv64-unknown-elf-gcc -mabi=ilp32 -march=rv32i -S main.c -o main.s
```



# Agenda

- Chamada e retorno de rotinas
- A pilha do programa
- ABI: *Application Binary Interface*
- **Passagem de parâmetros e retorno de valores**
- Variáveis locais
- Políticas de uso de registradores
- Quadro de pilha
- Implementação de rotinas compatíveis com a `ilp32`
- Chamadas de sistema (syscalls)

# Retorno de valor em funções

RISC-V ABI

- Se for um valor de até 32 *bits* retornamos o valor em `a0`.
- Se o valor tiver de 32 a 64 *bits*, retornamos o valor no par `a1:a0`.

# Passagem de Parâmetros

- **Por Registrador**
  - Parâmetros são colocados em registradores
  - Podem ser especificados parâmetros de entrada ou de saída (retorno de valores)
- **Pela Pilha**
  - Parâmetros são colocados na pilha do programa

- ABI RISC-V ILP32
  - Faz uso de registradores e da pilha!

# Passagem de Parâmetros

ABI RISC-V ILP32

- Para ilustrar os conceitos focaremos nos casos onde os parâmetros são valores escalares que podem ser armazenados com 32 ou menos *bits*.
  - Para outros casos veja a documentação da ABI em https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md

Valores escalares de 32 ou menos *bits*.

- 8 primeiros parâmetros vão em: `a0`, `a1`, ... `a7`
- Parâmetros restantes vão na pilha, empilhados de trás para frente (i.e., último parâmetro é empilhado primeiro e o parâmetro 9 por último).

# Uso da pilha para parâmetros

- Antes da chamada, os parâmetros são empilhados
- Dentro do procedimento, os parâmetros são lidos com o auxílio do registrador de pilha, o `SP` (*stack pointer*).
  - **OBS**: Rotina que está recebendo os parâmetros não deve desempilhar (remover da pilha) os parâmetros!
- Após a instrução de chamada, o espaço alocado para os parâmetros na pilha é desalocado
  - Os parâmetros devem ser desalocados (removidos) da pilha pela mesma rotina que os empilhou!

# Passagem de Parâmetros

Exemplo:

```
# Implementação da rotina sum10
int sum10(int a, int b, int c, int d, int e
          int f, int g, int h, int i, int j)
{
  return a+b+c+d+e+f+g+h+i+j;
}
```

```
# Chamando a rotina sum10
sum10(10,20,30,40,50,60,70,80,90,100);
```

# Passagem de Parâmetros

Exemplo:

```
# Chamando a rotina sum10
sum10(10,20,30,40,50,60,70,80,90,100);
```

```
li   a0, 10       # Parâmetro 1
li   a1, 20       # Parâmetro 2
li   a2, 30       # Parâmetro 3
li   a3, 40       # Parâmetro 4
li   a4, 50       # Parâmetro 5
li   a5, 60       # Parâmetro 6
li   a6, 70       # Parâmetro 7
li   a7, 80       # Parâmetro 8
addi sp, sp, -8   # Aloca espaço na pilha
li   t1, 100      # Empilha parâmetro 10
sw   t1, 4(sp)
li   t1, 90       # Empilha parâmetro 9
sw   t1, 0(sp)
jal  sum10        # Chama "sum10"
addi sp, sp, 8    # Desaloca parâmetros
```

ABI RISC-V ilp32: 8 primeiros parâmetros nos registradores `a0` a `a7` e parâmetros restantes na pilha, empilhados de trás para frente!

# Passagem de Parâmetros

Exemplo:

```
# Implementação da rotina sum10
int sum10(int a, int b, int c, int d, int e,
          int f, int g, int h, int i, int j)
{ return a+b+c+d+e+f+g+h+i+j; }
```

```
sum10:
 lw  t1, 0(sp)   # Carrega parâmetro 9 em t1
 lw  t2, 4(sp)   # Carrega parâmetro 10 em t2
 add a0, a0, a1 # Soma parâmetros
 add a0, a0, a2
 add a0, a0, a3
 add a0, a0, a4
 add a0, a0, a5
 add a0, a0, a6
 add a0, a0, a7
 add a0, a0, t1
 add a0, a0, t2 # Resultado em a0
 ret            # Retorna
```

Implementação da rotina `sum10`. Parâmetros 9 (i) e 10 (j) são acessados da pilha com auxílio do *stack pointer* (`sp`)

# Passagem de Parâmetros

Exemplo:

```
# Implementação da rotina sum10
int sum10(int a, int b, int c, int d, int e,
          int f, int g, int h, int i, int j)
{ return a+b+c+d+e+f+g+h+i+j; }
```

```
sum10:
 lw  t1, 0(sp)   # Carrega parâmetro 9 em t1
 lw  t2, 4(sp)   # Carrega parâmetro 10 em t2
 add a0, a0,   1 # Soma parâmetros
 add a0, a0, a
 add a0, a0, a
 add a0, a0, a4
```

**OBS**: Os valores são acessados com a ajuda de `sp`, mas a função não deve desalocar os parâmetros que recebeu pela pilha! Antes de retornar, o topo da pilha deve apontar para onde apontava quando a função foi chamada!

# Passagem de parâmetros por valor

Parâmetros podem ser passados por valor ou por referência

- **Por valor**: Uma cópia do valor é colocado no registrador ou na pilha
- **Por referência**: O endereço da variável que contém o valor é colocado no registrador ou na pilha

# Passagem de parâmetros por valor

Suponha a função C

```
int pow2(int v)
{
  return v*v;
}
```

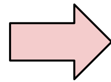
Exemplo de chamada

```
int main()
{
  return pow2(42);
}
```

# Passagem de parâmetros por valor

## Suponha a função C

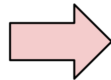
```
int pow2(int v)
{
  return v*v;
}
```



```
# pow2
# entrada: valor v em a0
# saida: v*v em a0
pow2:
   mul a0, a0, a0 # a0 = a0*a0
   ret
```

## Exemplo de chamada

```
int main()
{
  return pow2(42);
}
```

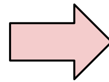


```
main:
  ...          # salva RA
  li  a0, 42   # a0 = 42
  jal pow2     # chama pow2
  ...          # recupera RA
  ret          # retorna
```

# Passagem de parâmetros por valor

Suponha a função C

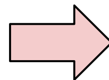
```
int pow2(int v)
{
  return v*v;
}
```



```
# pow2
# entrada: valor v em a0
# saida: v*v em a0
pow2:
   mul a0, a0, a0 # a0 = a0*a0
   ret
```

Exemplo de chamada

```
int x;
int main()
{
  return pow2(x);
}
```



```
main:
  ...              # salva RA
  la  t1, x        # t1 = &x
  lw  a0, (t1)     # a0 = x
  jal pow2         # chama pow2
  ...              # recupera RA
  ret              # retorna
```

# Passagem de parâmetros por referência

Suponha a função C

```
void inc(int* v)
{
  *v = *v + 1;
}
```

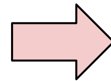
Exemplo de chamada

```
int x;
void inc_x()
{
  inc(&x);
}
```

# Passagem de parâmetros por referência

## Suponha a função C

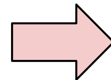
```
void inc(int* v)
{
  *v = *v + 1;
}
```



```
inc:
  lw   t1, (a0)   # t1 = *v
  addi t1, t1, 1 # t1 = t1 + 1
  sw   t1, (a0)   # *v = t1
  ret
```

## Exemplo de chamada

```
int x;
void inc_x()
{
  inc(&x);
}
```

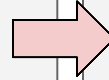


```
.data
x: .skip 4   # int x;
.text
inc_x:
  ...          # salva RA
  la  a0, x # a0 = &x
  jal inc      # chama inc
  ...          # recupera RA
  ret          # retorna
```

# Passagem de parâmetros por referência

Suponha a função C

```
void troca(int* a, int* b)
{
  int t = *a;
  *a = *b;
  *b = t;
}
```



```
troca:
  lw  t1, (a0) # t1 = *a
  lw  t2, (a1) # t2 = *b
  sw  t2, (a0) # *a = t2
  sw  t1, (a1) # *b = t1
  ret          # retorna
```

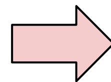
# Passagem de parâmetros por referência

## Suponha a função C

```
void troca(int* a, int* b)
{
  int t = *a;
  *a = *b;
  *b = t;
}
```

## Exemplo de chamada

```
int x, y;
void foo()
{
  troca(&x,&y);
}
```



```
.text
.common x, 4, 2 # int x
.common y, 4, 2 # int y
foo:
  ...            # salva RA
  la  a0, x      # a0 = &x
  la  a1, y      # a1 = &y
  jal troca      # chama troca
  ...            # recupera RA
  ret            # retorna
```

# Agenda

- Chamada e retorno de rotinas
- A pilha do programa
- ABI: *Application Binary Interface*
- Passagem de parâmetros e retorno de valores
- **Variáveis locais**
- Políticas de uso de registradores
- Quadro de pilha
- Implementação de rotinas compatíveis com a `ilp32`
- Chamadas de sistema (syscalls)

# Variáveis locais

Idealmente, as variáveis locais de um procedimento ou função devem ser alocadas em registradores. Entretanto, em alguns casos, as variáveis locais precisam ser alocadas na memória. Por exemplo:

- Quando um procedimento utiliza muitas variáveis locais, e o número de registradores não é suficiente para alocá-las;
- Quando a variável local não cabe em um registrador (P. ex: é um registro ou vetor);
- Quando a rotina faz uso do endereço da variável local;

# Variáveis globais vs locais

**Variáveis globais** inicializadas e não inicializadas são colocadas nas seções .data e .bss, respectivamente.

- Só há uma cópia de cada variável global.
- Cada variável global é identificada por um rótulo!

**Variáveis locais** que precisam ser alocadas na memória são alocadas na pilha do programa.

- Pode haver múltiplas cópias de uma variável local, **uma para cada rotina ativa**!
- O endereço de variáveis locais é calculado com base no *stack pointer* (`sp`).

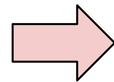
# Variáveis locais na memória

Variáveis locais que precisam ser alocadas na memória - são alocadas na pilha!

- O espaço para as variáveis locais é reservado na entrada da rotina, e desalocado ao final da rotina

## Exemplo 1

```
int foo()
{
 int id;
 get_uid(&id);
 return id;
}
```



```
.text
foo:
  ...                  # salva RA
  addi sp, sp, -4      # aloca id
  mv   a0, sp          # a0 = &id
  jal  get_uid         # chama get_uid
  lw   a0, (sp)        # a0 = id
  addi sp, sp, 4       # desaloca id
  ...                  # recupera RA
  ret                  # retorna
```

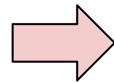
# Variáveis locais na memória

Variáveis locais que precisam ser alocadas na memória são alocadas na pilha!

- O espaço para as variáveis locais é reservado na entrada da rotina, e desalocado ao final da rotina

## Exemplo 2

```
int bar()
{
 int v[8];
 init_v(v);
 return v[4];
}
```



```
.text
bar:
  ...                   # salva RA
  addi sp, sp, -32 # aloca v
  mv   a0, sp          # a0 = v (&v[0])
  jal  init_v          # chama init_v
  lw   a0, 16(sp)    # a0 = v[4]
  addi sp, sp, 32   # desaloca v
  ...                   # recupera RA
  ret                   # retorna
```

# Agenda

- Chamada e retorno de rotinas
- A pilha do programa
- ABI: *Application Binary Interface*
- Passagem de parâmetros e retorno de valores
- Variáveis locais
- **Políticas de uso de registradores**
- Quadro de pilha
- Implementação de rotinas compatíveis com a `ilp32`
- Chamadas de sistema (syscalls)

# Política de uso dos registradores

- Rotinas pode utilizar muitos registradores
  - Armazenamento de variáveis, retorno de valores, passagem de parâmetros, ...

- Frequentemente, antes de utilizarmos um registrador para armazenar um novo valor, temos que salvar o valor anterior para que ele possa ser recuperado posteriormente.
  - Estes valores devem ser salvos na pilha do programa!

# Política de uso dos registradores

Exemplo:

- Rotina exchange modifica os registradores `a2` e `a3`
- Rotina mix chama exchange mas precisa preservar valor de `a2`

```
exchange:
  lw  a2, (a0) # a2 = *a
  lw  a3, (a1) # a3 = *b
  sw  a3, (a0) # *a = a3
  sw  a2, (a1) # *b = a2
  ret          # retorna
```

```
mix:
  ...             # Salva ra
  lw  a2, (a0)    # Carrega informação importante em a2
  la  a0, x       # Seta parâmetro 0 com &x
  la  a1, y       # Seta parâmetro 0 com &y
  jal exchange    # Chama a rotina exchange
  mv  a0, a2      # Retorna informação importante
  ...             # Recupera ra
  ret             # Retorna
```

# Política de uso dos registradores

Exemplo:

- Rotina exchange modifica os registradores `a2` e `a3`
- Rotina mix chama exchange mas precisa preservar valor de `a2`

```
exchange:
  lw   a2, (a0) # a2 = *a
  lw   a3, (a1) # a3 = *b
  sw   a3, (a0) # *a = a3
  sw   a2, (a1) # *b = a2
  ret
```

```
mix:
  ...              # Salva ra
  lw   a2, (a0)    # Carrega in
  la   a0, x       # Seta p   etr
  la   a1, y       # Se   parâmetro 0 com &y
  jal  exchange    # Chama a rotina exchange
  mv   a0, a2      # Retorna informação importante
  ...              # Recupera ra
  ret              # Retorna
```

Conteúdo de `a2` foi modificado pela rotina exchange.

# Política de uso dos registradores

Exemplo:

- Rotina exchange poderia salvar valor de `a2` e `a3` antes de alterá-los.

```
exchange:
  addi sp, sp, -8 # Aloca espaço na pilha
  sw   a2, 4(sp)  # Salva a2
  sw   a3, 0(sp)  # Salva a3
  lw   a2, (a0)   # a2 = *a
  lw   a3, (a1)   # a3 = *b
  sw   a3, (a0)   # *a = a3
  sw   a2, (a1)   # *b = a2
  lw   a3, 0(sp)  # Recupera a3
  lw   a2, 4(sp)  # Recupera a2
  addi sp, sp, 8  # Desaloca espaço
  ret             # retorna
```

# Política de uso dos registradores

Exemplo:

- Alternativamente, a rotina mix poderia salvar valor de `a2` e recuperá-lo após a execução da exchange.

```
mix:
  ...                # Salva ra
  lw   a2, (a0)      # Carrega informação importante em a2
  la   a0, x         # Seta parâmetro 0 com &x
  la   a1, y         # Seta parâmetro 0 com &y
  addi sp, sp, -4    # Aloca espaço na pilha
  sw   a2, 0(sp)     # Salva a2 antes de chamar exchange
  jal exchange       # Chama a rotina exchange
  lw   a2, 0(sp)     # Recupera a2 após executar exchange
  addi sp, sp, 4     # Desaloca espaço na pilha
  mv   a0, a2        # Retorna informação importante
  ...                # Recupera ra
  ret                # Retorna
```

# Política de uso dos registradores

**Quem deve salvar? A sub-rotina sendo chamada (`exchange`) ou a rotina que está chamando (`mix`)?**

# Política de uso dos registradores

**Quem deve salvar? A sub-rotina sendo chamada (`exchange`) ou a rotina que está chamando (`mix`)?**

- *Caller-saved*: Registradores que devem ser salvos pela rotina chamadora.
- *Callee-saved*: Registradores que devem ser salvos pela rotina sendo chamada.

ABIs do RISC-V:

  - `t0` a `t6`, `ra` e `a0` a `a7` => *caller-saved*
  - `s0` a `s11` => *callee-saved*

# Agenda

- Chamada e retorno de rotinas
- A pilha do programa
- ABI: *Application Binary Interface*
- Passagem de parâmetros e retorno de valores
- Variáveis locais
- Políticas de uso de registradores
- **Quadro de pilha**
- Implementação de rotinas compatíveis com a `ilp32`
- Chamadas de sistema (syscalls)

# Quadro de pilha

Uma **rotina ativa** é uma rotina que foi chamada mas ainda não retornou.

Pode haver múltiplas rotinas ativas em cada instante de tempo.

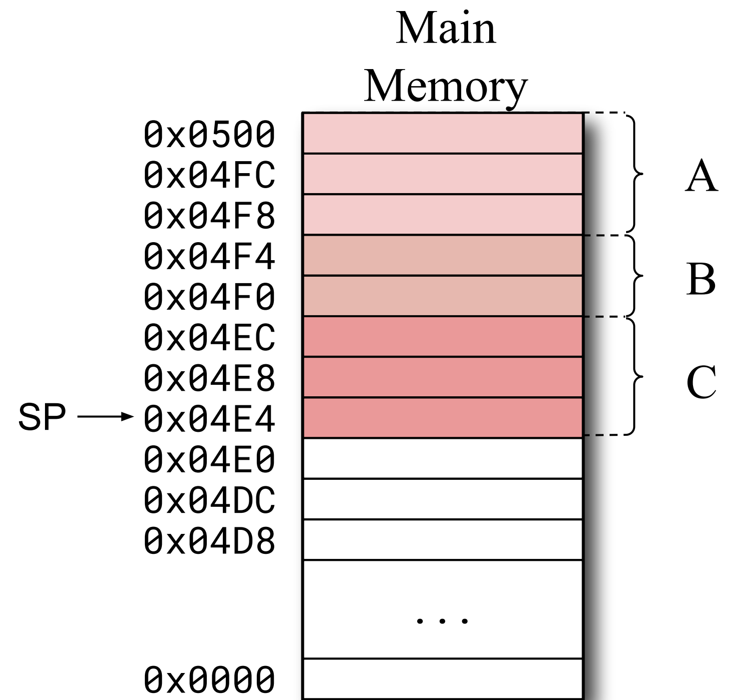
- Ex: Rotina A chama rotina B que chama rotina C.

Os dados de rotinas ativas (endereço de retorno, variáveis locais, …) **são armazenados na pilha**!

# Quadro de pilha

Os dados de rotinas ativas (endereço de retorno, variáveis locais, ...) **são armazenados na pilha**!

- Sempre que uma rotina é invocada, o código da própria rotina aloca espaço na pilha para salvar suas informações;
- Antes de retornar, o código da própria rotina desaloca o espaço alocado na pilha;

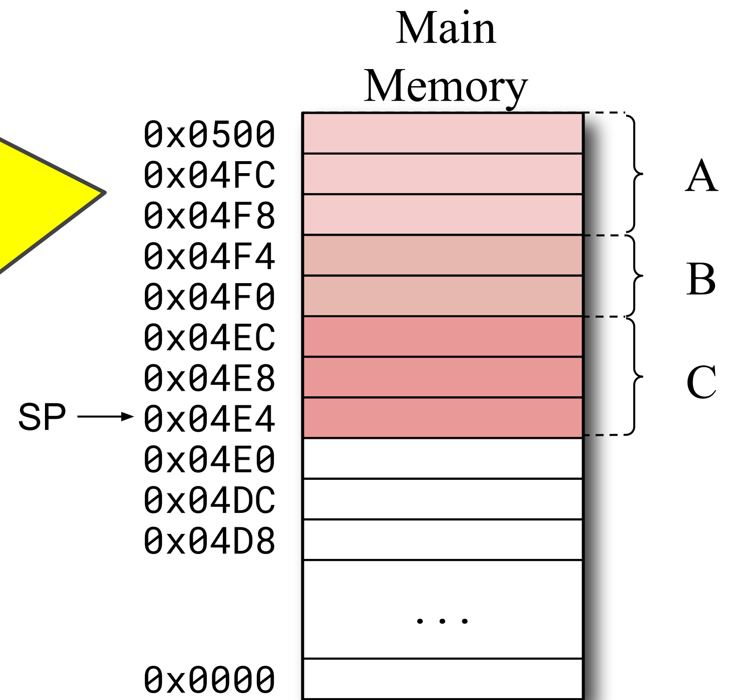


# Quadro de pilha

Os dados de rotinas ativas (endereço de retorno, variáveis locais, ...) **são armazenados na pilha**!

- Sempre que uma rotina é

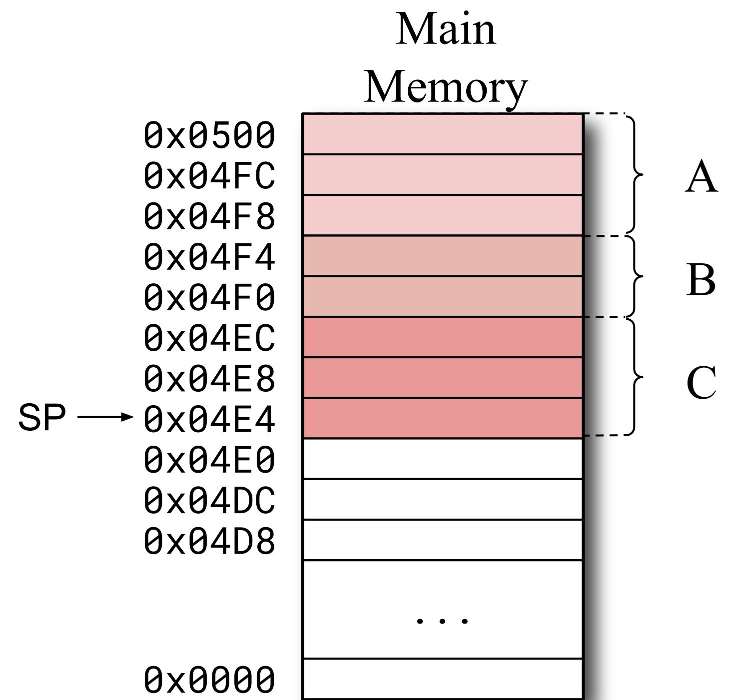
**O quadro de pilha** (ou ***stack frame***) é formado pelo conjunto de dados consecutivos na pilha adicionados por uma rotina ativa.



# Quadro de pilha

Os dados de rotinas ativas (endereço de retorno, variáveis locais, ...) **são armazenados na pilha**!

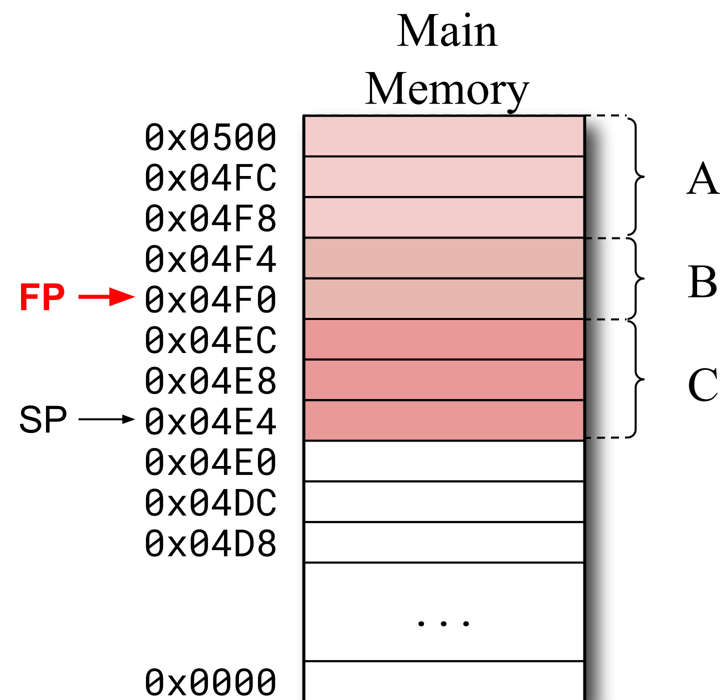
- Sempre que uma rotina é invocada, o código da própria rotina aloca um **quadro de pilha** para salvar suas informações;
- Antes de retornar, o código da própria rotina desaloca seu **quadro de pilha** da pilha;



# Quadro de pilha

O ***frame pointer***, ou **apontador de quadro**, é um apontador que aponta para o início do quadro atual - quadro da rotina ativa em execução.

- No RISC-V, o registrador `fp` (`x8`) é usado para armazenar o apontador de quadro.
- Estabelece um ponto fixo de acesso a parâmetros e variáveis locais!



# Quadro de pilha

Estabelece um ponto fixo de acesso a parâmetros e variáveis locais.

- Exemplo: Rotina `addijx`

```
int addijx(int a, int b, int c, int d, int e,
           int f, int g, int h, int i, int j)
{
    return get_x() + i + j;
}
```

# Quadro de pilha

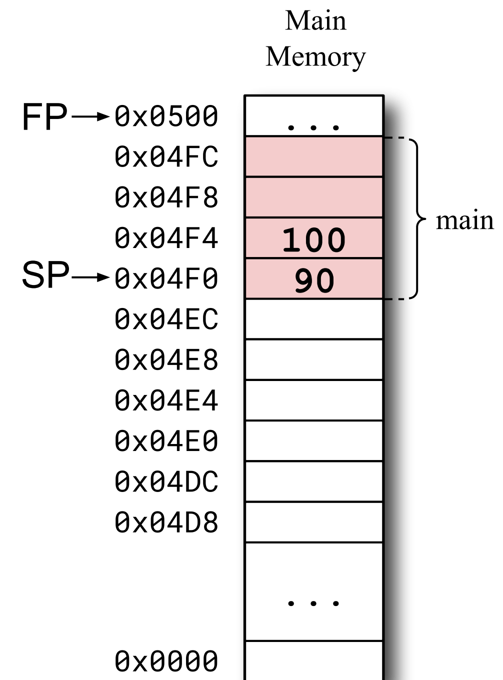
Estabelece um ponto fixo de acesso a parâmetros e variáveis locais.

- Exemplo: Rotina `addijx`

```
int addijx(int a, int b, int c, int d, int e,
           int f, int g, int h, int i, int j)
{
    return get_x() + i + j;
}
```

- Suporemos que a rotina main invocou a rotina `addijx`

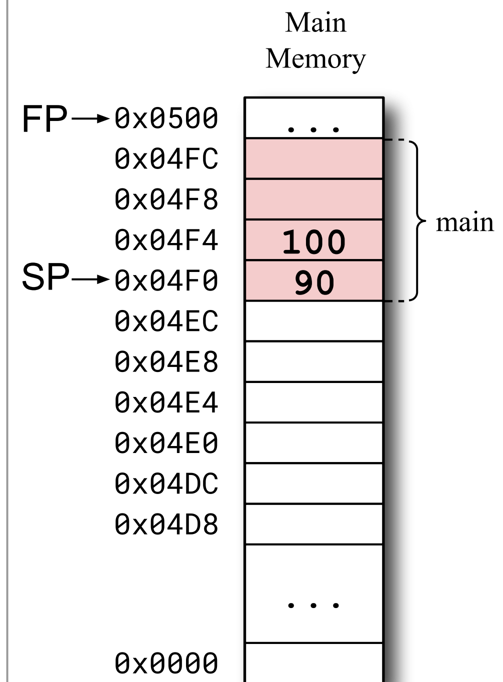
```
int main()
{
  return addijx(10,20,30,40,50,60,70,80,90,100);
}
```



# Quadro de pilha

Estabelece um ponto fixo de acesso a parâmetros e variáveis locais.

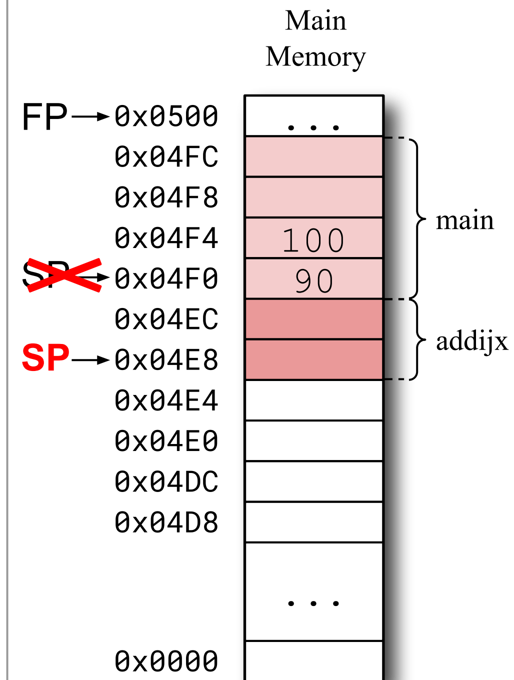
```
addijx:
  addi sp, sp, -8 # Aloca o quadro de pilha
  sw   ra, 4(sp)  # Salva o endereço de retorno
  sw   fp, 0(sp)  # Salva o frame pointer anterior
  addi fp, sp, 8  # Ajusta o FP p/ o quadro atual
  jal  get_x      # Invoca a rotina get_x
  lw   a1,  (fp)  # Carrega parâmetro 9 (i) em a1
  lw   a2, 4(fp)  # Carrega parâmetro 10 (j) em a2
  add  a0, a0, a1 # a0 = get_x() + i
  add  a0, a0, a2 # a0 = get_x() + i + j
  lw   fp, 0(sp)  # Recupera Frame Pointer anterior
  lw   ra, 4(sp)  # Recupera endereço de retorno
  addi sp, sp, 8  # Desaloca o quadro de pilha
  ret             # Retorna
```



# Quadro de pilha

Estabelece um ponto fixo de acesso a parâmetros e variáveis locais.

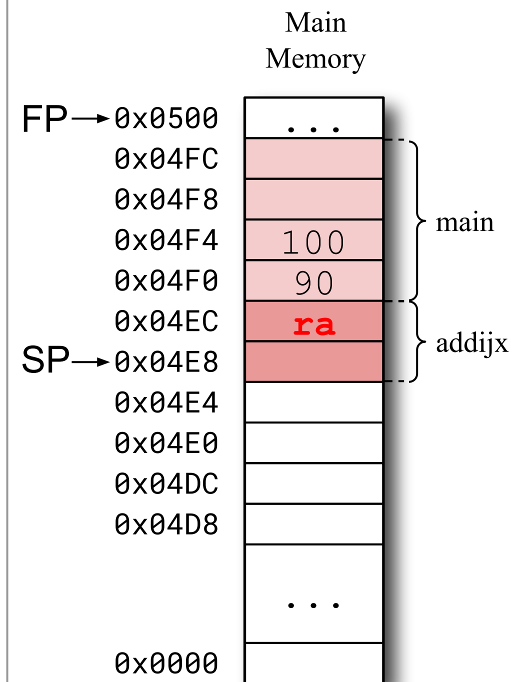
```
addijx:
  addi sp, sp, -8 # Aloca o quadro de pilha
  sw   ra, 4(sp)  # Salva o endereço de retorno
  sw   fp, 0(sp)  # Salva o frame pointer anterior
  addi fp, sp, 8  # Ajusta o FP p/ o quadro atual
  jal  get_x      # Invoca a rotina get_x
  lw   a1,  (fp)  # Carrega parâmetro 9 (i) em a1
  lw   a2, 4(fp)  # Carrega parâmetro 10 (j) em a2
  add  a0, a0, a1 # a0 = get_x() + i
  add  a0, a0, a2 # a0 = get_x() + i + j
  lw   fp, 0(sp)  # Recupera Frame Pointer anterior
  lw   ra, 4(sp)  # Recupera endereço de retorno
  addi sp, sp, 8  # Desaloca o quadro de pilha
  ret             # Retorna
```



# Quadro de pilha

Estabelece um ponto fixo de acesso a parâmetros e variáveis locais.

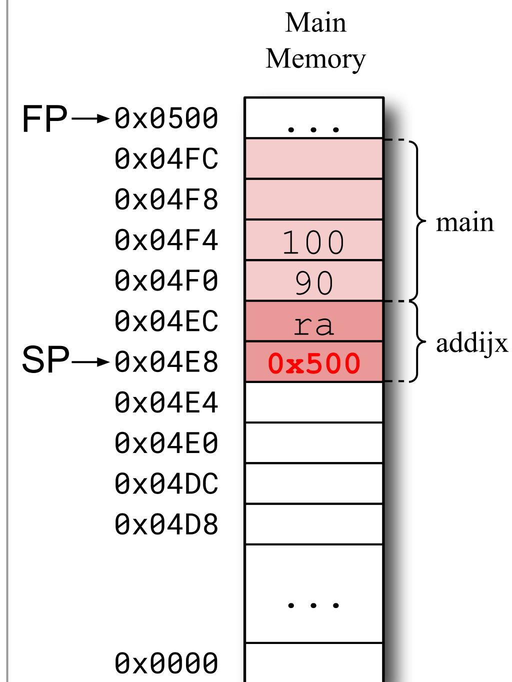
```
addijx:
  addi sp, sp, -8 # Aloca o quadro de pilha
  sw   ra, 4(sp)  # Salva o endereço de retorno
  sw   fp, 0(sp)  # Salva o frame pointer anterior
  addi fp, sp, 8  # Ajusta o FP p/ o quadro atual
  jal  get_x      # Invoca a rotina get_x
  lw   a1,  (fp)  # Carrega parâmetro 9 (i) em a1
  lw   a2, 4(fp)  # Carrega parâmetro 10 (j) em a2
  add  a0, a0, a1 # a0 = get_x() + i
  add  a0, a0, a2 # a0 = get_x() + i + j
  lw   fp, 0(sp)  # Recupera Frame Pointer anterior
  lw   ra, 4(sp)  # Recupera endereço de retorno
  addi sp, sp, 8  # Desaloca o quadro de pilha
  ret             # Retorna
```



# Quadro de pilha

Estabelece um ponto fixo de acesso a parâmetros e variáveis locais.

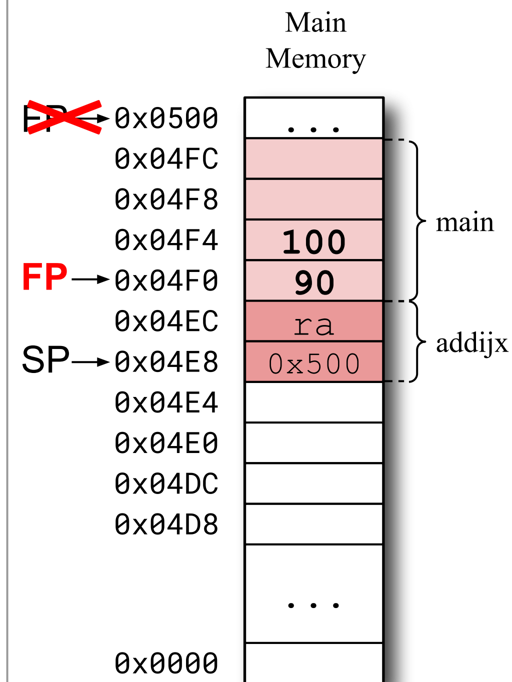
```
addijx:
  addi sp, sp, -8 # Aloca o quadro de pilha
  sw   ra, 4(sp)  # Salva o endereço de retorno
  sw   fp, 0(sp)  # Salva o frame pointer anterior
  addi fp, sp, 8  # Ajusta o FP p/ o quadro atual
  jal  get_x      # Invoca a rotina get_x
  lw   a1,  (fp)  # Carrega parâmetro 9 (i) em a1
  lw   a2, 4(fp)  # Carrega parâmetro 10 (j) em a2
  add  a0, a0, a1 # a0 = get_x() + i
  add  a0, a0, a2 # a0 = get_x() + i + j
  lw   fp, 0(sp)  # Recupera Frame Pointer anterior
  lw   ra, 4(sp)  # Recupera endereço de retorno
  addi sp, sp, 8  # Desaloca o quadro de pilha
  ret             # Retorna
```



# Quadro de pilha

Estabelece um ponto fixo de acesso a parâmetros e variáveis locais.

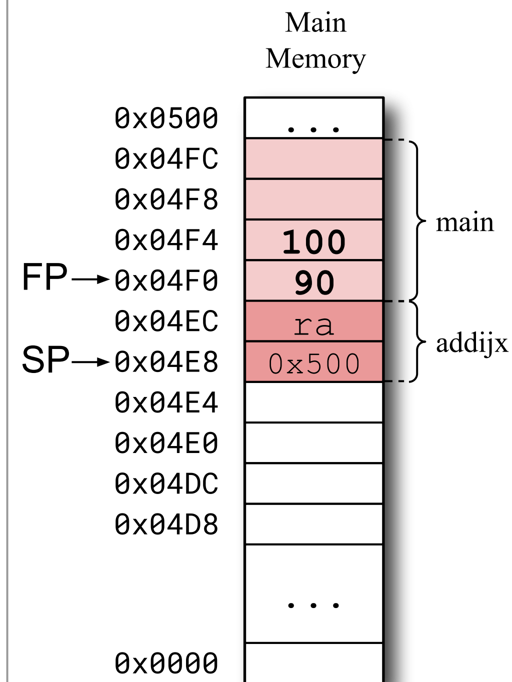
```
addijx:
  addi sp, sp, -8 # Aloca o quadro de pilha
  sw   ra, 4(sp)  # Salva o endereço de retorno
  sw   fp, 0(sp)  # Salva o frame pointer anterior
  addi fp, sp, 8  # Ajusta o FP p/ o quadro atual
  jal  get_x      # Invoca a rotina get_x
  lw   a1,  (fp)  # Carrega parâmetro 9 (i) em a1
  lw   a2, 4(fp)  # Carrega parâmetro 10 (j) em a2
  add  a0, a0, a1 # a0 = get_x() + i
  add  a0, a0, a2 # a0 = get_x() + i + j
  lw   fp, 0(sp)  # Recupera Frame Pointer anterior
  lw   ra, 4(sp)  # Recupera endereço de retorno
  addi sp, sp, 8  # Desaloca o quadro de pilha
  ret             # Retorna
```



# Quadro de pilha

Estabelece um ponto fixo de acesso a parâmetros e variáveis locais.

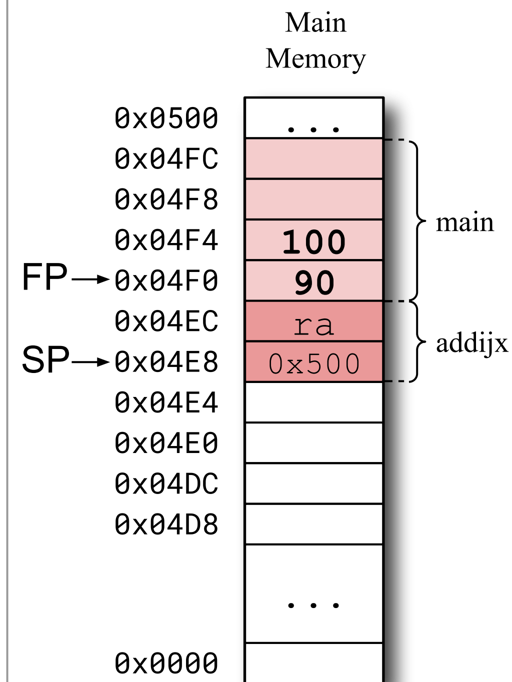
```
addijx:
  addi sp, sp, -8 # Aloca o quadro de pilha
  sw   ra, 4(sp)  # Salva o endereço de retorno
  sw   fp, 0(sp)  # Salva o frame pointer anterior
  addi fp, sp, 8  # Ajusta o FP p/ o quadro atual
  jal  get_x      # Invoca a rotina get_x
  lw   a1,  (fp)  # Carrega parâmetro 9 (i) em a1
  lw   a2, 4(fp)  # Carrega parâmetro 10 (j) em a2
  add  a0, a0, a1 # a0 = get_x() + i
  add  a0, a0, a2 # a0 = get_x() + i + j
  lw   fp, 0(sp)  # Recupera Frame Pointer anterior
  lw   ra, 4(sp)  # Recupera endereço de retorno
  addi sp, sp, 8  # Desaloca o quadro de pilha
  ret             # Retorna
```



# Quadro de pilha

Estabelece um ponto fixo de acesso a parâmetros e variáveis locais.

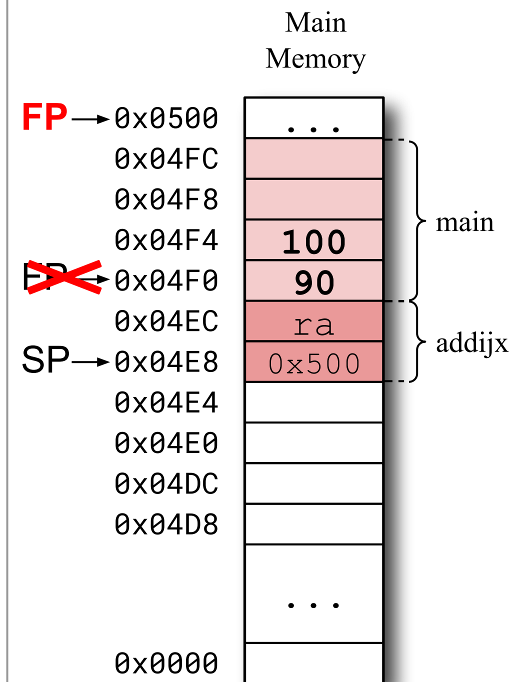
```
addijx:
  addi sp, sp, -8 # Aloca o quadro de pilha
  sw   ra, 4(sp)  # Salva o endereço de retorno
  sw   fp, 0(sp)  # Salva o frame pointer anterior
  addi fp, sp, 8  # Ajusta o FP p/ o quadro atual
  jal  get_x      # Invoca a rotina get_x
  lw   a1,  (fp)  # Carrega parâmetro 9 (i) em a1
  lw   a2, 4(fp)  # Carrega parâmetro 10 (j) em a2
  add  a0, a0, a1 # a0 = get_x() + i
  add  a0, a0, a2 # a0 = get_x() + i + j
  lw   fp, 0(sp)  # Recupera Frame Pointer anterior
  lw   ra, 4(sp)  # Recupera endereço de retorno
  addi sp, sp, 8  # Desaloca o quadro de pilha
  ret             # Retorna
```



# Quadro de pilha

Estabelece um ponto fixo de acesso a parâmetros e variáveis locais.

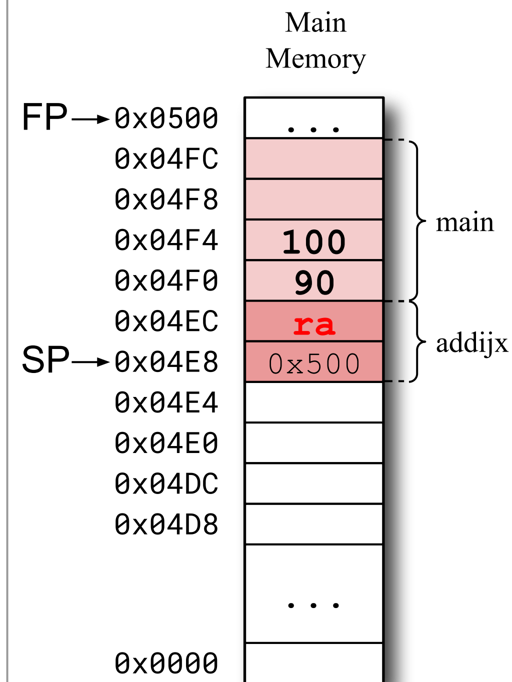
```
addijx:
  addi sp, sp, -8 # Aloca o quadro de pilha
  sw   ra, 4(sp)  # Salva o endereço de retorno
  sw   fp, 0(sp)  # Salva o frame pointer anterior
  addi fp, sp, 8  # Ajusta o FP p/ o quadro atual
  jal  get_x      # Invoca a rotina get_x
  lw   a1,  (fp)  # Carrega parâmetro 9 (i) em a1
  lw   a2, 4(fp)  # Carrega parâmetro 10 (j) em a2
  add  a0, a0, a1 # a0 = get_x() + i
  add  a0, a0, a2 # a0 = get_x() + i + j
  lw   fp, 0(sp)  # Recupera Frame Pointer anterior
  lw   ra, 4(sp)  # Recupera endereço de retorno
  addi sp, sp, 8  # Desaloca o quadro de pilha
  ret             # Retorna
```



# Quadro de pilha

Estabelece um ponto fixo de acesso a parâmetros e variáveis locais.

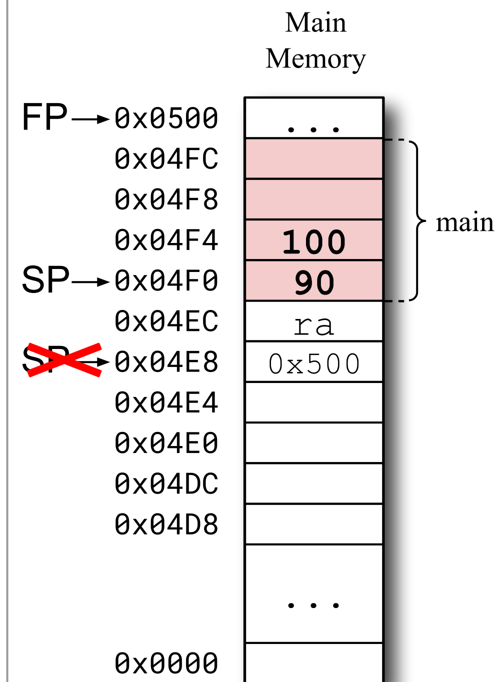
```
addijx:
  addi sp, sp, -8 # Aloca o quadro de pilha
  sw   ra, 4(sp)  # Salva o endereço de retorno
  sw   fp, 0(sp)  # Salva o frame pointer anterior
  addi fp, sp, 8  # Ajusta o FP p/ o quadro atual
  jal  get_x      # Invoca a rotina get_x
  lw   a1,  (fp)  # Carrega parâmetro 9 (i) em a1
  lw   a2, 4(fp)  # Carrega parâmetro 10 (j) em a2
  add  a0, a0, a1 # a0 = get_x() + i
  add  a0, a0, a2 # a0 = get_x() + i + j
  lw   fp, 0(sp)  # Recupera Frame Pointer anterior
  lw   ra, 4(sp)  # Recupera endereço de retorno
  addi sp, sp, 8  # Desaloca o quadro de pilha
  ret             # Retorna
```



# Quadro de pilha

Estabelece um ponto fixo de acesso a parâmetros e variáveis locais.

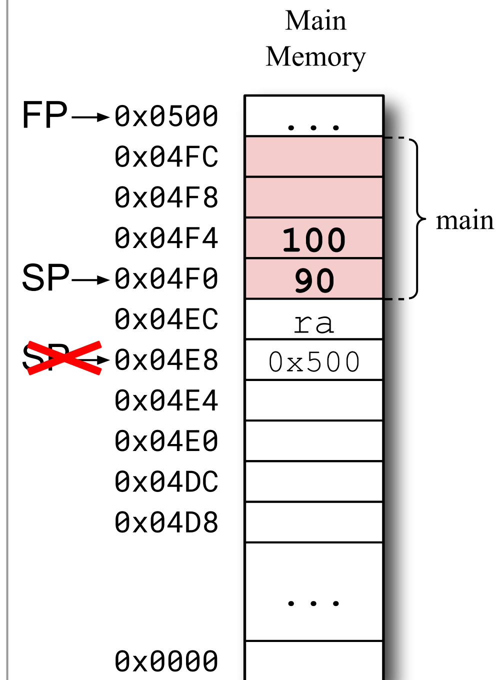
```
addijx:
  addi sp, sp, -8 # Aloca o quadro de pilha
  sw   ra, 4(sp)  # Salva o endereço de retorno
  sw   fp, 0(sp)  # Salva o frame pointer anterior
  addi fp, sp, 8  # Ajusta o FP p/ o quadro atual
  jal  get_x      # Invoca a rotina get_x
  lw   a1,  (fp)  # Carrega parâmetro 9 (i) em a1
  lw   a2, 4(fp)  # Carrega parâmetro 10 (j) em a2
  add  a0, a0, a1 # a0 = get_x() + i
  add  a0, a0, a2 # a0 = get_x() + i + j
  lw   fp, 0(sp)  # Recupera Frame Pointer anterior
  lw   ra, 4(sp)  # Recupera endereço de retorno
  addi sp, sp, 8  # Desaloca o quadro de pilha
  ret             # Retorna
```



# Quadro de pilha

Estabelece um ponto fixo de acesso a parâmetros e variáveis locais.

```
addijx:
  addi sp, sp, -8 # Aloca o quadro de pilha
  sw   ra, 4(sp)  # Salva o endereço de retorno
  sw   fp, 0(sp)  # Salva o frame pointer anterior
  addi fp, sp, 8  # Ajusta o FP p/ o quadro atual
  jal  get_x      # Invoca a rotina get_x
  lw   a1,  (fp)  # Carrega parâmetro 9 (i) em a1
  lw   a2, 4(fp)  # Carrega parâmetro 10 (j) em a2
  add  a0, a0, a1 # a0 = get_x() + i
  add  a0, a0, a2 # a0 = get_x() + i + j
  lw   fp, 0(sp)  # Recupera Frame Pointer anterior
  lw   ra, 4(sp)  # Recupera endereço de retorno
  addi sp, sp, 8  # Desaloca o quadro de pilha
  ret             # Retorna
```



# Quadro de pilha

Exemplos de valores que são adicionados ao quadro de pilha:

- Variáveis locais que precisam ser alocadas na memória;
- Parâmetros para rotinas que são chamadas (do 9º em diante)
- Registradores *callee-saved* que forem modificados pela rotina (p.ex: `s0/fp`, `s1`, ...)
- Registradores *caller-saved* que devem ter valores preservados após chamada de outras rotinas (p.ex: `ra`, `a0`, `a1`, ...)

# Mantendo o *stack pointer* alinhado

A ABI `ilp32` especifica que o apontador de pilha (SP) deve sempre estar alinhado a endereços múltiplos de 16.

- Uma forma de garantir esta propriedade é alocar/desalocar quadros de pilha em múltiplos de 16 *bytes*.

# Agenda

- Chamada e retorno de rotinas
- A pilha do programa
- ABI: *Application Binary Interface*
- Passagem de parâmetros e retorno de valores
- Variáveis locais
- Políticas de uso de registradores
- Quadro de pilha
- **Implementação de rotinas compatíveis com a `ilp32`**
- Chamadas de sistema (syscalls)

# Rotinas compatíveis com a `ilp32`

- O ponto de entrada de uma rotina é definido pelo seu rótulo.
  - Caso a rotina seja uma tradução de código em C, o nome do rótulo deve ser igual ao nome da rotina em C
- A instrução `ret` deve ser usada para retornar da rotina.
- Parâmetros devem ser acessados de acordo com a ABI `ilp32`
  - Valores até 32 *bits*: 8 primeiros em a0 a a7 e os restantes na pilha.
- Caso a rotina precise gravar valores temporários na memória (parâmetros para outras rotinas, variáveis locais, valores de registradores, ...) estes devem ser gravados no quadro da rotina
  - Neste caso, o quadro deve ser alocado/desalocado no início/final da rotina e o tamanho do quadro deve ser múltiplo de 16; e

# Rotinas compatíveis com a `ilp32`

- Registradores *callee-saved* que são modificados pela rotina devem ser salvos no quadro de pilha no início da rotina e recuperados do quadro de pilha no final da rotina.
- A rotina pode usar registradores *caller-saved* sem salvá-los, entretanto, se precisar preservar seus valores após chamada de outras rotinas, estes devem ser salvos no quadro de pilha antes da chamada da rotina e recuperados do quadro de pilha após o retorno da rotina.
- Caso um quadro de pilha (*stack frame*) seja alocado, o código da rotina deve ajustar o registrador `fp` (*frame pointer*) para apontar para o quadro da rotina atual. Neste caso:
  - o valor antigo do registrador `fp` (*frame pointer*) deve ser salvo e recuperado do quadro de pilha antes do retorno da rotina.

# Agenda

- Chamada e retorno de rotinas
- A pilha do programa
- ABI: *Application Binary Interface*
- Passagem de parâmetros e retorno de valores
- Variáveis locais
- Políticas de uso de registradores
- Quadro de pilha
- Implementação de rotinas compatíveis com a `ilp32`
- **Chamadas de sistema (syscalls)**

# Chamadas de sistema

- Programas de usuário geralmente operam com dados armazenados na memória e nos registradores.
- Entrada e saída de dados são realizadas com o auxílio de dispositivos de entrada e saída:
  - Teclado, monitor, impressora, rede, etc...
  - Estes dispositivos são gerenciados pelo sistema operacional e a entrada e saída é feita através de requisições ao sistema operacional.
- Requisições ao sistema operacional => Chamadas de sistemas (*system call* ou *syscall*)

# Chamadas de sistema

Chamada de sistema = invocar serviços (rotinas) do sistema operacional.

- São realizadas com a instrução `ecall`
- A convenção usada na chamada depende do sistema operacional!

# Chamadas de sistema

## Exemplo no Linux: escrita em arquivo

```
char* msg = "My message”;
char* filename = "my_file.txt”;
...
/* Set the flags. */
int flags = O_WRONLY | O_CREAT | O_TRUNC;
/* Open the file. */
int fd = open (filename,flags);
/* Write the first 5 bytes pointed by msg into the file. */
write(fd, msg, 5);
/* Close the file. */
close(fd);
```

# Chamadas de sistema

## Exemplo no Linux: Chamando a *syscall write*

```
# Syscall write:
# a0: File descriptor
# a1: Apontador para o buffer
# a2: Número de bytes a serem escritos

# Ajustar os parâmetros
li a0, fd      # a0: Valor do file descriptor
la a1, msg     # a1: Apontador para o buffer
li a2, 5       # a2: Número de bytes a serem escritos
```

```
write(fd, msg, 5);
```

# Chamadas de sistema

## Exemplo no Linux: Chamando a *syscall write*

```
# Syscall write:
# a0: File descriptor
# a1: Apontador para o buffer
# a2: Número de bytes a serem escritos

# Ajustar os parâmetros
li a0, fd     # a0: Valor do file descriptor
la a1, msg    # a1: Apontador para o buffer
li a2, 5      # a2: Número de bytes a serem escritos
# Chamar a função write
li a7, 64     # Código da syscall: 64 == write
ecall         # Invoca o sistema operacional
```

```
write(fd, msg, 5);
```

# Chamadas de sistema

Podemos implementar uma função *write* que encapsula a chamada à *syscall write*:

```
# Entrada: a0: descritor do arquivo (fd)
#          a1: apontador para o buffer
#          a2: número de bytes a ser escrito
# Saída:   a0: número de bytes escrito pela write
write:
  addi sp, sp, -16 # Aloca o quadro de pilha
  sw ra, 0(sp)     # Salva ra
  li  a7, 64       # Código da syscall: 64 == write
  ecall            # Invoca o sistema operacional
  lw ra, 0(sp)     # Restaura ra
  addi sp, sp, 16  # Desaloca o quadro de pilha
  ret              # Retorna
```

# Chamadas de sistema

Podemos implementar uma função *read* que encapsula a chamada à *syscall read*:

```
# Entrada: a0: descritor do arquivo (fd)
#          a1: apontador para o buffer
#          a2: número de bytes a ser lido
# Saída:   a0: número de bytes lidos pela syscall read
read:
  addi sp, sp, -16 # Aloca o quadro de pilha
  sw ra, 0(sp)     # Salva ra
  li  a7, 63       # Código da syscall: 63 == read
  ecall            # Invoca o sistema operacional
  lw ra, 0(sp)     # Restaura ra
  addi sp, sp, 16  # Desaloca o quadro de pilha
  ret              # Retorna
```

# Agenda

- Chamada e retorno de rotinas
- A pilha do programa
- ABI: *Application Binary Interface*
- Passagem de parâmetros e retorno de valores
- Variáveis locais
- Políticas de uso de registradores
- Quadro de pilha
- Implementação de rotinas compatíveis com a `ilp32`
- Chamadas de sistema (syscalls)