CS 201

Digital Logic

procedures and the stack

Procedures

- Break your code into manageable pieces
  - aka subroutines, subprograms, functions, methods…
  - repeat common code without re-writing
- Jump to a procedure
  - Load the program counter with a new value
- Jump back to where you were before the procedure
  - restore the program counter's old value

`jal`

- This instruction is used to jump to a subroutine. Called "jump and link"
- `jal` is used to invoke the procedure
  - occurs in the parent procedure
  - `jal ProcedureLabel`
  - `jal` will first take the current value of the program counter and store it in `$ra`
  - then loads the PC with the new value
- Store `$ra` so we can get back to where we left off

$ra

- `$ra` is a special register
  - reserved for procedure calls
  - called “return address”
- jal stores PC+4 in $ra
- when the procedure is done, return to where we left off (instruction after the jal)
  - `jr $ra`
  - Jump to the address specified in a register

Procedure call example

```
add $a1, $a2, $a3
jal proc
add $a1, $a2, $a3
```

\$ra ← PC+4

PC ← proc

```
proc: add $v1, $a2, $a3
      add $t1, $v2, $v3
      jr $ra
```

PC ← \$ra

Transferring Arguments to Procedures

- Procedures often operate on *arguments*
  - abs(*n*) finds the absolute value of *n*
- two places to store information in MIPS
  - registers
  - memory
  - (so far - we'll see soon)
- Registers can hold information for a procedure call
  - between a *caller* and a *callee* procedure

Arguments and Procedure calls

- *caller* places information in a specific *register*
- *callee* assumes that register has the desired value
- callee procedure produces a *result*
- callee places result in an agreed-upon register
- Upon return, caller consults agreed register for result
- Problem: which registers?
  - recall: syscall

Register use during procedure calls

```
proc: add $v1, $a2, $a3
      add $t1, $v2, $v3
      jr  $ra
```

- During this procedure, `$v1` and `$t1` are modified
  - what if the caller had important information in one of these?
- Adopt a convention for register usage over procedure calls:
  - which to leave alone and which we can modify

variable and argument registers

- *variables* ($v) can be used *within* subroutines
  - caller *should not* assume these will stay the same
  - normally used for return values (v=value)
- *arguments* ($a) are stable between subroutines
  - caller *should* assume these will stay the same
  - callee *should not* modify these
  - normally used for arguments to functions

variable and argument registers

- `$t` are temporary registers
  - same as variable - callee can use
- `$s` are saved registers
  - same as arguments - callee should not use
- variables(`$v`)and temp(`$t`) are *caller-saved*
  - ***caller*** must actively save these if they are to be preserved ***across the procedure call***
- arguments(`$a`)and saved(`$s`) are *callee-saved*
  - ***callee*** must actively save and restore these if they are to be modified ***during the procedure***,

It’s on the sheet

this should be “yes”, I think…
see if you are paying attention!

REGISTERS

NAME	NMBR	USE	STORE?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes
$f0-$f31	0-31	Floating Point Registers	Yes

- Note: this is only a convention
  - merely an agreement between programmers
  - you can do whatever you like
  - if you want your code to be usable by others, or if you want to use another’s code
    - adhere to the convention.

Other callee-saved registers

- `$gp`
- `$sp`
- `$fp`
  - we don't know about the use of these yet
- `$ra`
  - return address
  - if this gets changed during the procedure, the procedure can't return to the original location

Nested subroutines

- Each subroutine needs a **`jal`** in the caller, and a matching **`jr $ra`** at the end of the callee
  - if the callee wants to call another procedure, must save the **`$ra`** somewhere
    - another register, a memory location…

```
addu $a0, $ra, $zero    # $ra -> $a0
jal  next                # call next
addu $ra, $a0, $zero    # $a0 -> $ra
jr   $ra                 # return
```

Procedure design

- Caller setup
  - *before calling a procedure*
    - put arguments into `$a` registers
    - save `$v` or `$t` register values that you may need after the call
- Callee save
  - *upon entering a procedure*
    - save and arguments (`$a`) and temp (`$t`)
      - only if they will be modified
    - save pointers (`$gp, $fp, $sp, $ra`)
      - if you will call another procedure

Procedure design

- Callee restore
  - *before returning from a procedure*
    - place return values in `$v` registers
    - restore any arguments (`$a`), saved (`$s`) and pointers (`$gp`, `$fp`, `$sp`, `$ra`) that were modified during the procedure
- Caller continue
  - *after calling a procedure*
    - retrieve return values from $v registers
    - restore values to $v and $t registers, if used

Procedure example: Factorial

```
        .text
        .globl main
main:

        li      $a0, 10
        jal     fact

        move $a1, $v0
        la      $a0, answer
        jal     printf

        .data
answer:
        .asciiz         "The answer is %d\n"
```

Caller setup

Caller continue

Caller setup

```
        .text
fact:
        la      $t0, Storage
        sw      $a0, 0($t0)  # save arg

        lw      $v0, 0($t0)          # load n
        bgtz    $v0, L2
        li      $v0, 1               # return 1
        j       L1                   # to return code
L2:
        lw      $v1, 0($t0)     # load n
        subu    $v0, $v1, 1     # compute n-1
        move    $a0, $v0        # move arg to $a0
        jal     fact
        lw      $v1, 0($t0)     # load n
        mul     $v0, $v0, $v1   # return val in $v0
L1:
        lw      $a0, 0($t0)  # restore arg
        jr      $ra
        .data
Storage:
        .asciiz "00000000"
```

Callee save

Callee restore

More arguments and variables

- Callee saved arguments in an arbitrary place
  - would be good to have a standard place, since it's a common task
- More than 4 arguments will be a problem
  - could use saved temp, but they really are for temporary values of a higher scope
    - eg that other procedures can see
- Need a common area for additional arguments
  - must be common between procedures
- Called "Stack"

Stack

- Special area of memory used for temporary values
  - Push a value to the stack, pop from the stack
- `$sp` = stack pointer
  - points to the top of the stack
  - most recently pushed piece of data
- access the stack using `lw` and `sw`
  - load from stack, save to stack

Load from and save from stack

- `lw $t4, 0($sp)`
    - load the top word of the stack into $t4
- `sw $t4, 0($sp)`
    - save a word to the top of the stack
        - but this will over-write the data on the top of the stack
- Push: two instructions
    - `addi $sp,$sp,-4` # make room on the stack
    - `sw $t4, 0($sp)` # place data on the stack

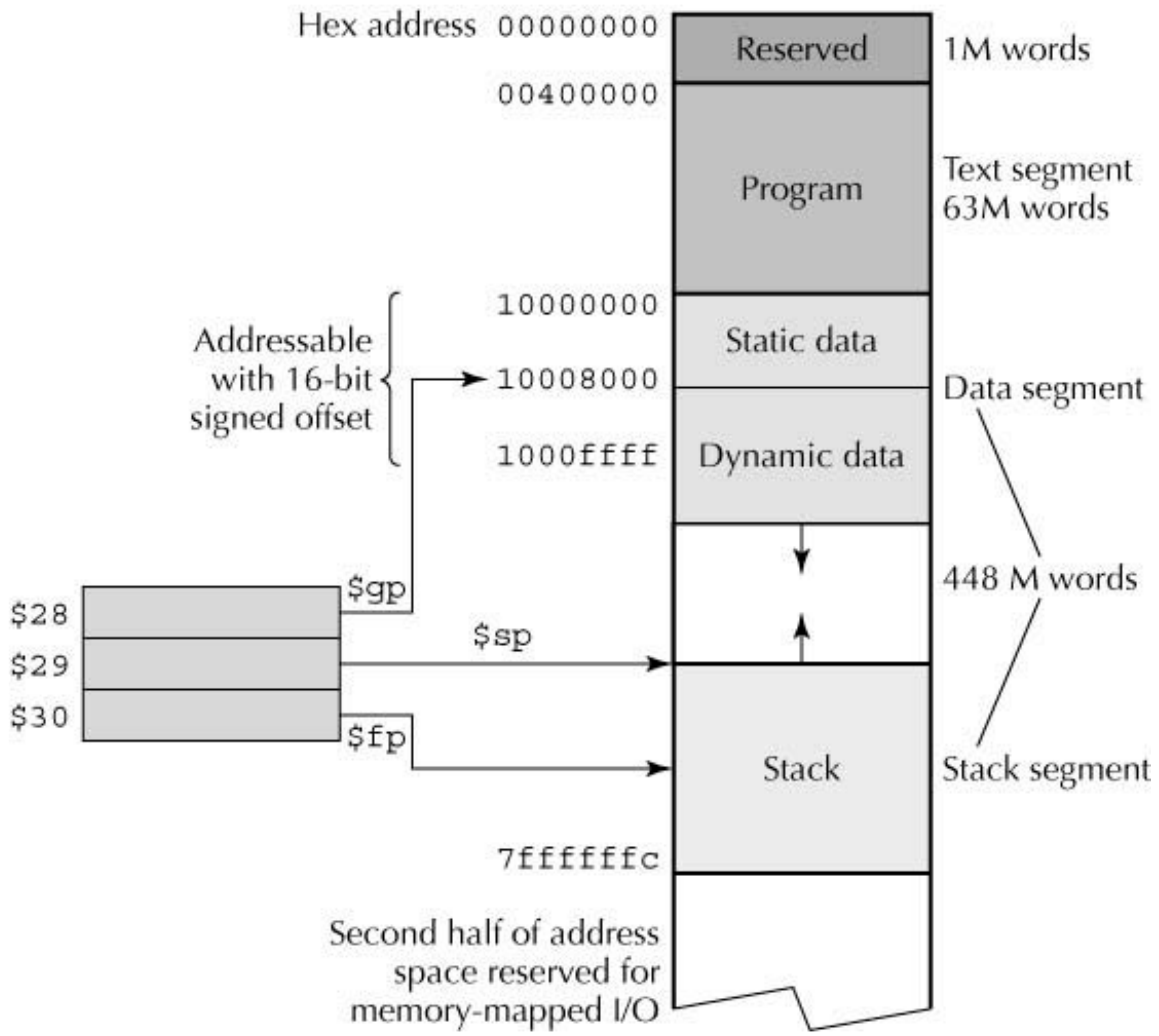
Load from and save to the stack

- Pop: retrieve from the stack
  - also two instructions:

```
lw  $t4, 0($sp)   # get data from the stack
addi $sp,$sp,4    # reduce the stack
```

- Note: data doesn't go away, but isn't "on the stack" anymore.
- Stack grows from high address toward low addresses
  - `$sp` starts toward the bottom of the memory

MIPS Memory address space



MIPS Memory address space

- `$gp` = global pointer
  - points to the beginning of the dynamic data segment of memory
    - more later
- `$sp` = stack pointer
  - points to the top of the stack
- `$fp` = frame pointer
  - used for delimiting part of the stack for procedure use
    - more later

Memory address space

- 1M words reserved for system
- 63M words for the text segment
  - when you write .text, this is where it goes
- Rest of the top half of the memory is for data
  - static data, pre-defined with .data
  - dynamic data, allocated at run-time
  - stack
    - stack and dynamic data grow toward each other

Back to the stack

- Data at `$sp` is accessible
  - using `0($sp)`
- Data below `$sp` ("under" the top of the stack) is accessible
  - using, say, `4($sp)`
- Data above `$sp` is still accessible
  - using, say, `-4($sp)`
  - but is not considered valid data, since it already has been popped.

Stack manipulation

- add 5 empty spaces to the top of the slot

```
addi $sp, $sp, -20
```

- discard 10 words off the stack at once

```
addi $sp, $sp, 40
```

- Access the 15th stack element

```
lw $t4, 56($sp)    #(first element is 0($sp))
```

- Save the return address (at the beginning of a procedure)

```
addi $sp,$sp,-4
sw     $ra, 0($sp)
```

Procedures and the stack

- Procedure calling can benefit from the use of the stack in 2 ways
  - Callee saving register values it will use
    - e.g. saving `$ra` for nested procedures
  - caller passing more than 4 arguments to a procedure
- Each procedure can also access data on the stack for other purposes
  - each procedure has a *stack frame*.

subroutine call and return

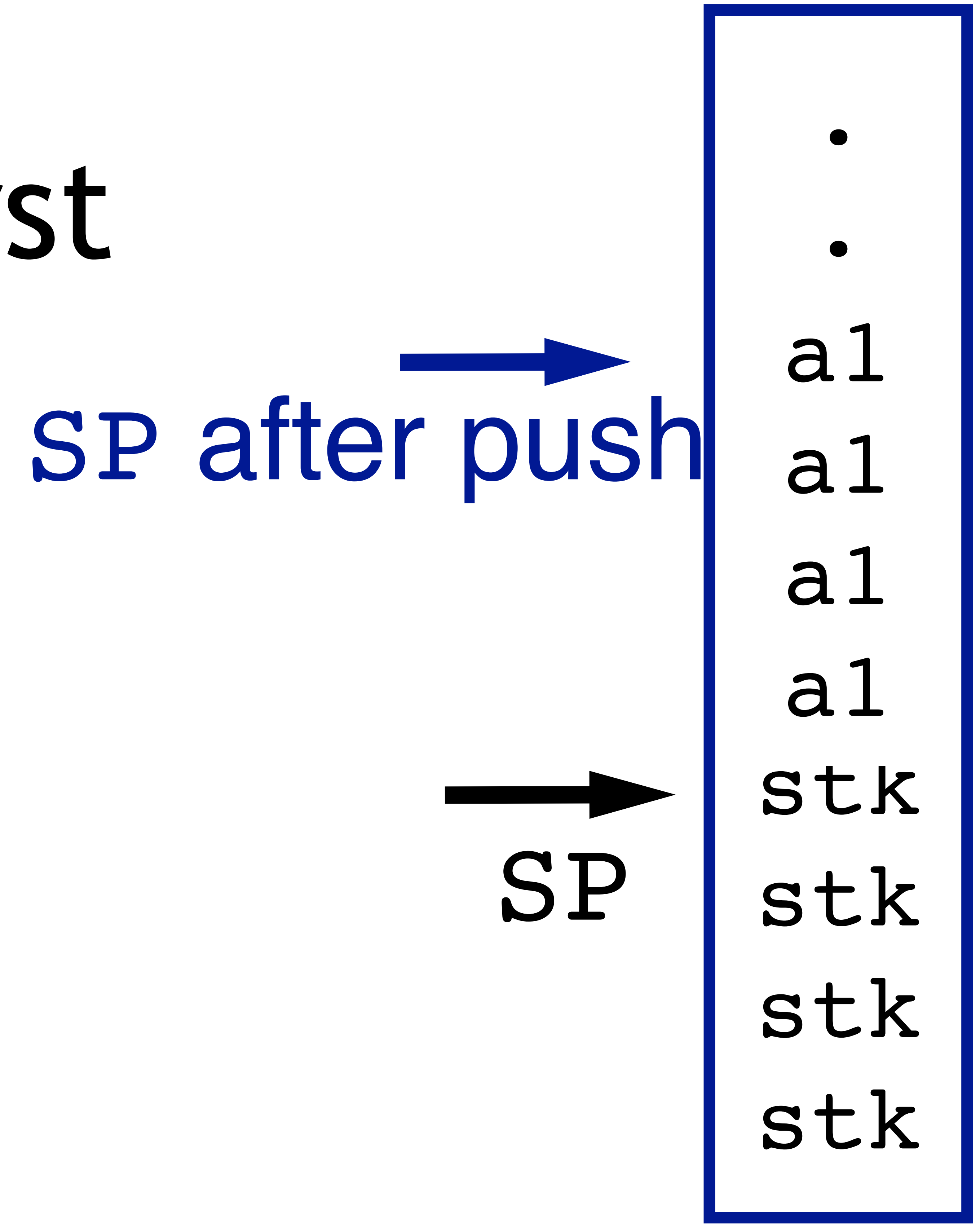
- Assumption: the stack is in the same state just before `jr $ra` as it was just after the `jal`
  - Any stack changes must be undone before `jr $ra`
- Stack is often used during the procedure
  - Preserving registers that the subroutine needs
  - Local variables, scratch area

Preserving registers using the stack

Subroutines must assume that all `$a` and `$s` registers contain important data

- To use an `$a` or `$s` register, first back it up
- Push it onto the stack

push `$a1`




```
addi $sp,$sp,-4
sw  $a1, 0($sp)
```

Preserving registers using the stack

- pop in the reverse order pushed

```
sub:   addi    $sp, $sp, -4
       sw      $ra, 0($sp)
       addi    $sp, $sp, -4
       sw      $a0, 0($sp)
       ...
       lw      $a0, 0($sp)
       addi    $sp, $sp, 4
       lw      $ra, 0($sp)
       addi    $sp, $sp, 4
       jr      $ra
```

Optimizing multiple stack operations

- make room for all stack elements first, then place using offsets (with `$sp` as base)

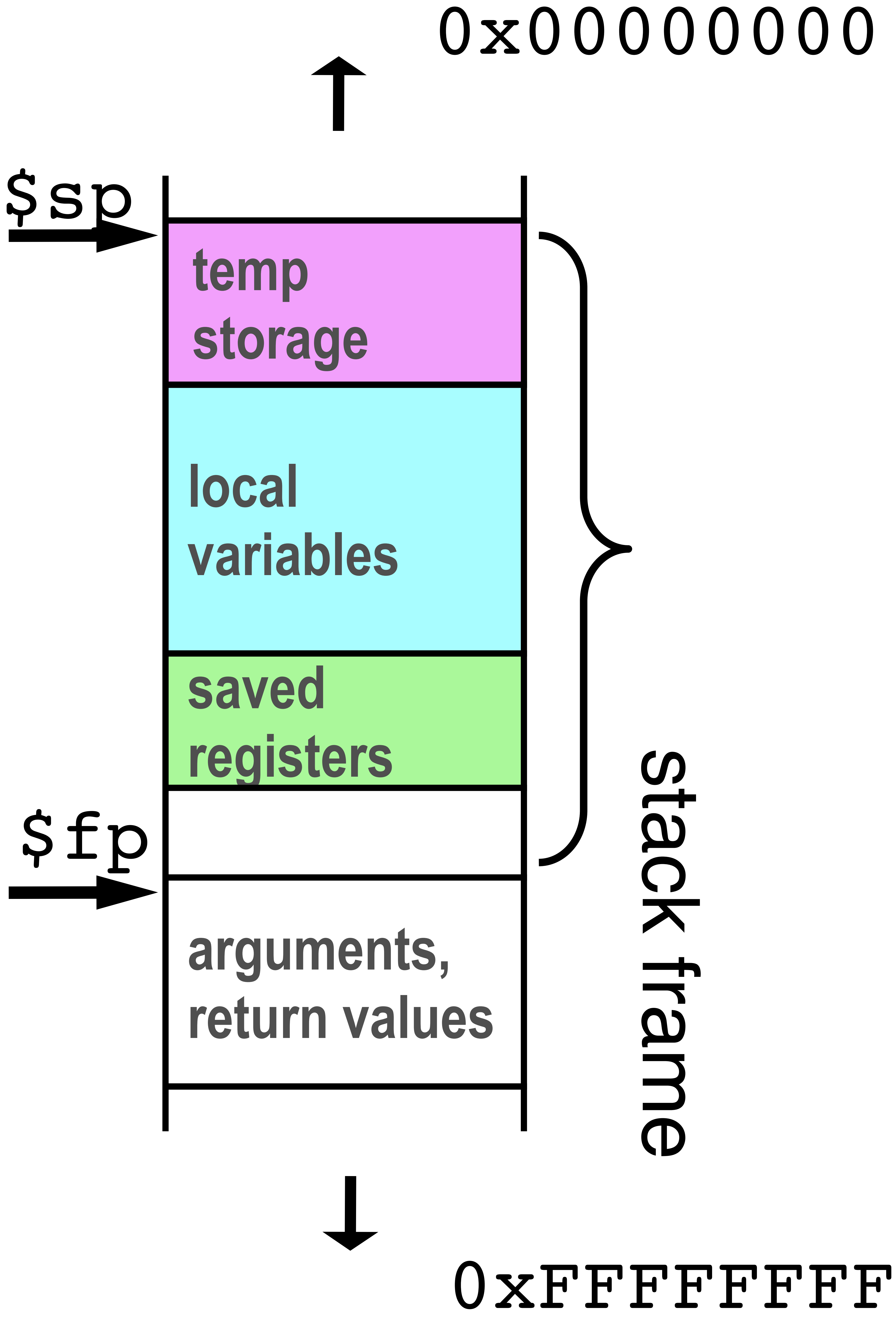
```
subprog:   addi $sp, $sp, -8
        sw  $ra, 4($sp)
        sw  $a0, 0($sp)
       ...
        lw $a0, 0($sp)
        lw $ra, 4($sp)
        addi $sp, $sp, 8
         jr     $ra
```

Formalizing stack use by procedure calls

- Stack Frame: an organized section of the stack
- contains registers to be preserved
- also contains
  - Arguments passed to the procedure
  - Space for return values from the procedure
  - Space for local variables and scratch space
  - `$fp` - frame pointer. Used to access data in the stack frame

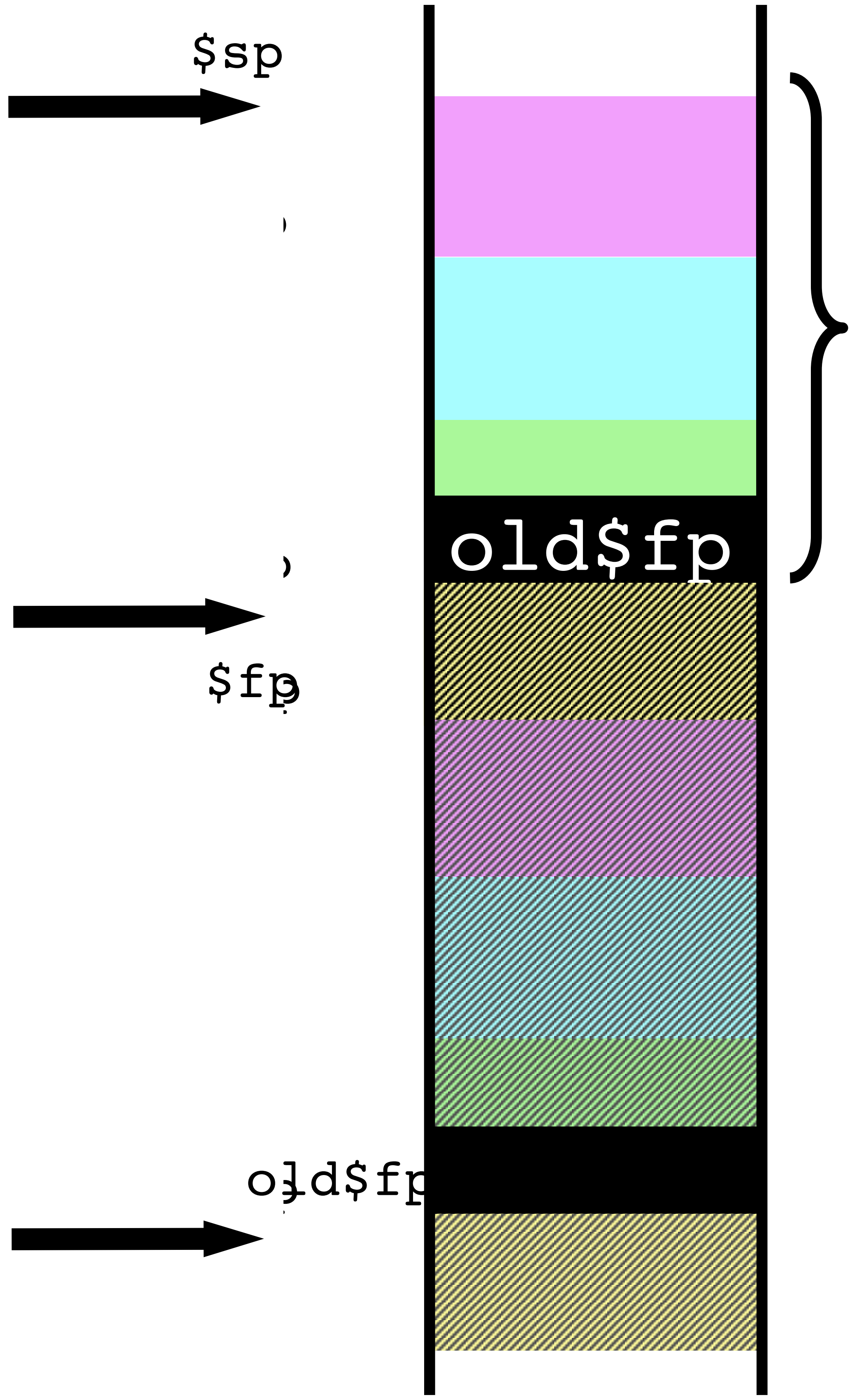
Stack Frame

- `$sp` points to the top of the stack (as always)
- `$fp` points to the bottom of the current stack frame
- data in the stack frame can be accessed relative to `$fp` or `$sp`
- Each procedure has a separate stack frame



Building the stack frame

- caller pushes additional arguments and space for return values (as needed)
- caller calls callee (no stack change)
- callee sets up new stack frame
  - store caller's `$fp` `(old$fp)`
  - set `$fp` to current `$sp`
- callee preserves registers and allocates space for local variables
- callee can then use stack for temporary storage



Code for building the stack frame

- caller

```
la   $t1,data
addi $sp,$sp,-8
sw   $t1, 4($sp)
jal   proc
```

Stack space for 1 argum
and 1 return value

place argument
(address of "data")
at 4($fp)

Jump to subroutine

Code for building the stack frame

- callee beginning

```
proc:
  sw   $fp,-4($sp)
  addi $fp,$sp,0
  addi $sp,$sp,-12
  sw  $ra,4($sp)
  sw  $s0,0($sp)
  ...
 lw  $t1,4($fp)
```

old $fp to stack

set $fp to $sp

space for saved

registers (and $fp)

saved registers

accessing the argum

Code for un-building the stack frame

- callee ending

```
 sw   $t1,0($fp)
...
 lw    $s0,0($sp)
lw    $ra,4($sp)
addi $sp,$fp,0
lw    $fp,-4($sp)
jr    $ra
```

save a return value

retrieve saved register

restore $sp to $fp

retrieve old $fp

return

Code for un-building the stack frame

- caller unpacking

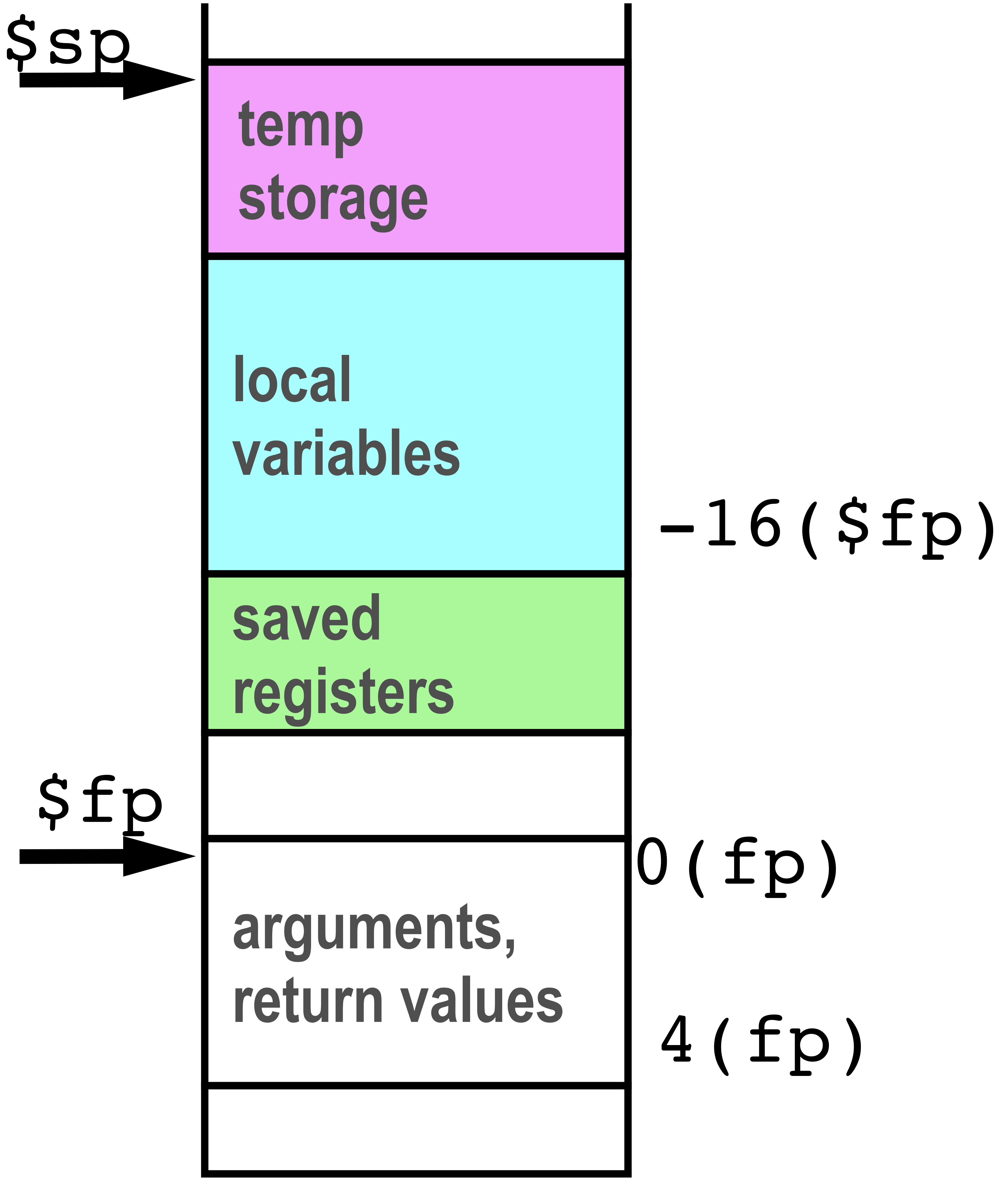
```
la   $t1,data
addi $sp,$sp,-8
sw   $t1, 4($sp)
jal   proc
lw   $t2, 0($sp)
addi $sp,$sp,8
```

retrieve return value

restore stack

Accessing data in the stack frame

- Use `$fp`:
  - `$fp` is static for the frame
    - $sp may change
  - can hard-code offsets from `$fp` to data in the frame
    - `$fp−k` for local variables
    - `$fp+k` for arguments or return values



Stack Frame

- Note that the stack frame is *not required*
  - if a procedure doesn't alter many registers or call another procedure
    - eg: only needs to save $s0, never alters stack:

```
proc:  sw  $s0,-4($sp)
       ...
       lw  $s0,-4($sp)
       jr  $ra
```

- Also, stack frame can look different
  - depending on the compiler or programmer
  - this method is a standard one