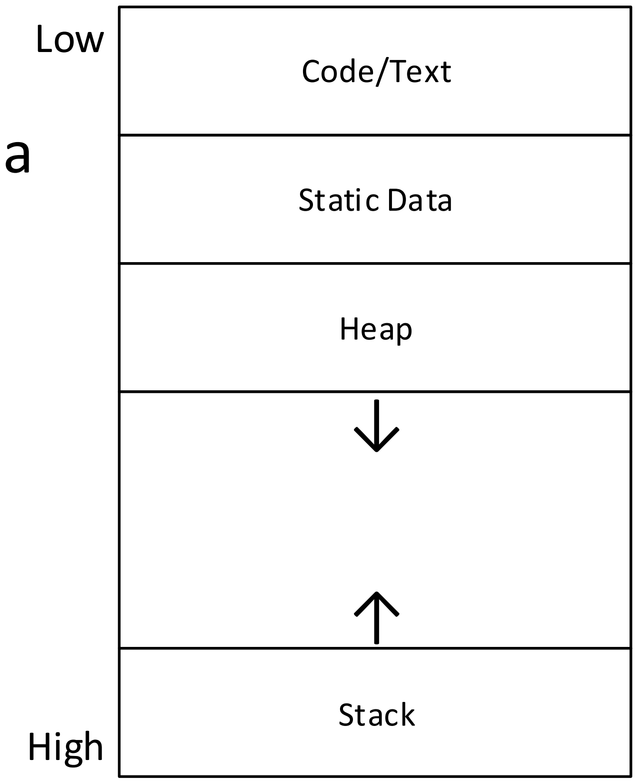
Run-time Environment

Prof. James L. Frankel
Harvard University

Version of 7:46 PM 11-Apr-2022

Storage Organization

- Automatic objects are stored on the stack
- Global/static objects are stored in the Static Data area
- malloc'ed objects are stored in the heap
- Unused heap storage is either Garbage Collected or explicitly freed – in C, the heap storage needs to be explicitly freed
- This format allows both the stack and the heap to grow



Global/Static Variables

- All global/static variables are stored in the Static Data area
  - The Static Data area is introduced in MIPS assembly code with the **.data** directive
  - In MIPS assembly code, all global variables (except for **main**) should be declared and referenced using the variable name declared by the programmer, but with a prefix of **_Global_**
    - For example, the global user variable **myVar** would be named **_Global_myVar** in MIPS assembly code
- There must be a function named **main** defined in the compilation module
  - The **main** function name is the only global symbol that cannot and should not have a prefix of **_Global_**

MIPS O32 ABI Calling Conventions (simplified)

- First four words of actual parameters (arguments) are passed in registers \$a0 through \$a3
    - Register \$a0 will contain all or part of the leftmost (first) actual parameter, if there are any parameters
- First two words of return value are returned in registers \$v0 and \$v1
    - If the return value fits in a single word, it is returned in register \$v0
- If the return value is either a structure or a union, then the caller must pass a pointer to a sufficiently large memory area for that return value in \$a0.
    - The callee places the return value into this area before it returns
    - If \$a0 is used for this purpose, then registers \$a1-\$a3 are used for the first three words of actual parameters

Temporary ($t) Registers

- Before calling a subroutine, the *caller* must push any temporary ($t) registers whose values it requires being maintained across the call
  - The stack is used to save the $t registers' values so that subroutines can be recursively nested
- After returning from the subroutine, the *caller* must pop any temporary ($t) registers that it pushed immediately before the call

Saved ($s) Registers

- On entry, the callee must push any saved ($s) registers that it may modify
    - The stack is used to save the $s registers' values so that subroutines can be recursively nested
- Just prior to returning, the callee must pop any saved ($s) registers that it saved immediately before the call
- Registers are saved in numerical order, with higher-numbered registers saved in higher memory addresses

Return Address Register

- The $ra register is loaded with the subroutine’s return address by the *And Link* instructions
  - This is the mechanism through which a subroutine is called
- However, if a nested subroutine call is executed, register $ra would be overwritten
- Therefore, if a subroutine is not a leaf subroutine, then, on entry, the callee must push the return address ($ra) register
  - A leaf subroutine is one that does not call any other subroutines
  - The stack is used to save the $ra register’s value so that subroutines can be recursively nested
- Just prior to returning, the callee must pop the return address ($ra) register if it saved it on entry

Argument ($a) Registers

- The $a registers are loaded with the actual parameters (arguments) passed to the subroutine by the caller
    - If more than four words are required for actual parameters, the remaining words are passed on the stack
- However, if a nested subroutine *which requires parameters* is called, some of the $a registers would be overwritten
- Therefore, if a subroutine is not a leaf subroutine, then, on entry, the callee must push the actual parameter ($a) registers that might be overwritten
    - A leaf subroutine is one that does not call any other subroutines
    - The stack is used to save the $a registers values' so that subroutines can be recursively nested
- Just prior to returning, the callee must pop the actual parameter ($a) registers if it saved them on entry

Local Variables

- Space for local (automatic) variables must be reserved on the stack
- This space is reserved by the callee at subroutine entry
- The space is released by the callee at subroutine return
- The stack is used for local variables so that subroutines can be recursively nested

Stack Frames

- A **stack frame** (sometimes referred to as an **activation record**) is created on the stack for each *invocation* of a subroutine (*i.e.,* each time any subroutine is called)
  - A new stack frame is created for each recursive subroutine call
- The format of the stack frame is the embodiment of the interface between the caller and the callee
- The stack frame contains the data in memory needed for a subroutine in an agreed representation
  - Passed actual parameters
  - Return value
  - Local variables
  - Locations in which saved registers ($s) can be saved
  - Return address
  - …

Program Execution/Stack Backtracing

- By following the conventions for use of the registers and the stack frame, a debugger is able to determine the state of the program at any point in time
- The $gp will point to the memory used for global variables (simplification)
- The program counter will point to the instruction that is about to be executed
    - The PC's value can be used to determine the currently executing subroutine
- $sp will point to the top of stack
- $fp will allow access to the current stack frame
    - The stack frame will contain all local variables
    - The $ra in the stack frame points to the caller's instruction to be executed upon return
    - The $fp in the stack frame points to the caller's stack frame

Initial Return Address Register Value

- The $ra register is initialized to zero by the operating system before starting any user program
- This allows a chain of return addresses found in stack frames to be followed through arbitrarily nested subroutine calls

Stack Frame Pointer

- We could address data on the stack using an offset off the $sp, but…
- Because data may be pushed on the stack and popped off the stack during the execution of a subroutine (thus changing where the $sp points), the offsets to access data on the stack would change
- We would like to address data on the stack using a stable offset during the execution duration of a subroutine
- Therefore, we dedicate another register, $fp, the **frame pointer**, to be a stable pointer to the current stack frame

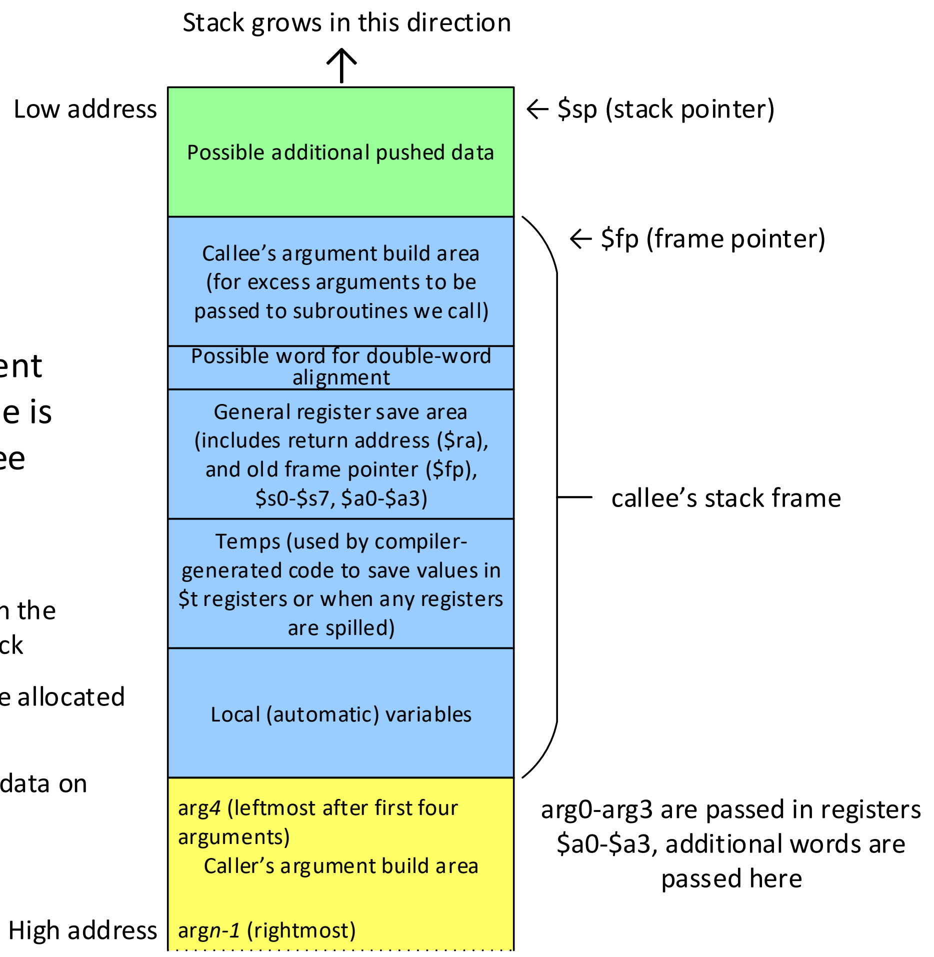
Stack Frame Constraints

- Stack frame must be on a double-word boundary
  - Enforces alignment for the largest MIPS data type
- *Always* leave space in the stack frame for $a0-$a3 and $ra in case a subroutine is called
- The old frame pointer is stored in the stack frame as a dynamic stack frame back link
  - *Always* leave space in the stack frame for the old $fp
- $ra and $fp are stored in the general register save area
- The minimum size for a stack frame is 24 bytes

Stack Frame Argument Build Area

- Before calling a subroutine, any additional data pushed on the stack must be popped
  - After doing so, the Argument Build Area will be on the top of stack
- The Argument Build Area will consist of all words needed for arguments after the first four words are passed in $a0 through $a3
- At the time the stack frame is allocated on subroutine entry, the maximum size required for arguments to be passed to called subroutines must be reserved

Stack Frame Format


Stack grows in this direction
Low address
High address
$sp (stack pointer)
$fp (frame pointer)
Possible additional pushed data
Callee's argument build area (for excess arguments to be passed to subroutines we call)
Possible word for double-word alignment
General register save area (includes return address ($ra), and old frame pointer ($fp), $s0-$s7, $a0-$a3)
Temps (used by compiler-generated code to save values in $t registers or when any registers are spilled)
Local (automatic) variables
arg4 (leftmost after first four arguments)
Caller's argument build area
argn-1 (rightmost)
callee's stack frame
arg0-arg3 are passed in registers $a0-$a3, additional words are passed here

The current subroutine is the callee

🟨 allocated in the caller's stack

🟦 stack frame allocated by callee

🟩 additional data on stack

Our Stack Frame Implementation

- In our implementation, on entry to a function we will always save:
    - The $fp, caller's frame pointer
    - The $ra, return address
    - All $a registers that are used to pass parameters to us
    - All of the $s registers
- The $fp, $ra, and the $s registers will be restored just before we return to our caller

- Before calling a function, we will save:
    - Any $t registers whose values we need after that function returns
    - We save the values of those $t registers in the **Temps** portion of the stack frame
- The $t registers will be restored immediately after that function returns

Dynamic vs. Static Old Frame Pointer Links

- In our stack frame, we store the old frame pointer (*i.e.,* the frame pointer of the caller subroutine – that is, the subroutine that called us)
- Some languages allow definition of functions within other functions
  - These languages allow access to the local variables of the functions in which they're nested
- Implementation of this concept requires some means to access all of the enclosing function's local variables
  - A static link – that is, a pointer to the stack frame of the enclosing function's most recent invocation
  - A display – an array of pointers to the stack frames of all of the enclosing function's most recent invocations