Almost all operand storage used by programs is provided by memory. Even though registers are more efficiently accessed by instructions, there are too few registers to hold the stored values or working set of a program during execution. Memory appears as a large, linear addressed array of storage locations. However the storage needs of a program are varied; so this large array is broken into regions that are managed differently. Figure 1 gives a typical organization of a program's storage in memory.

Global variables are statically defined and accessible by all procedures. Since their number, size, and type are known at compile time, the compiler can generate extremely efficient code to access them. Unfortunately, heavy use of global variables leads to non-modular code that is difficult to maintain. Globals also lack the flexibility to support program mechanisms where the amount and type of storage required is not known at compile time.

Dynamic (run-time) storage management allows proper allocation and deallocation of needed operand storage resulting from unpredictable execution behavior (e.g., calling patterns of procedures), varying input data set sizes, and dynamic storage needs not tied to procedure invocations.

A program's storage map begins with code storage followed by statically allocated variables (typically globals) which are fully defined, allocated, and possibly initialized by the compiler. The remaining space is divided between the heap which grows towards larger addresses and the stack which begins at the largest address and grows towards smaller addresses. Although large virtual address spaces obviate concerns over sharing available space between these two dynamically allocated spaces, most embedded platforms face this finite storage resource.

Since much of data storage is associated with procedure invocations (procedure calling), a natural place to allocate this storage in on the stack. The stack's Last-In,
First-Out (LIFO) behavior perfectly accommodates the needs of nested procedure calls. Consider a person working on a problem that contains many parts where the information needed to perform each part is placed on a separate piece of paper. The first page contains information for the highest level procedure in the problem. When a subproblem is reached, a new page is placed on top that contains information for that task. This process continues as lower level tasks are undertaken and more pages are added to the stack. When the top task is completed, the associated page is removed from the stack and work on the new top page is resumed. Additional subtasks (and pages) may be added and then removed until all work is completed and the initial page is removed. At all times, the information associated with the current task is on the page on top of the stack. This is exactly how storage is managed in procedural programming environments.

In programs, storage required by a procedure is created as a block on the stack called an activation frame. This block includes storage for input parameters, the return value, and local variables. The functions Bar and Foo, shown in Figure 2, will illustrate how activation frames are created and used. The function Bar requires an activation frame that contains two input parameters I and J, its return value, and three local variables M, N, and P. Function Foo's activation frame contains two input parameters X and Y, a return value, and two locals A and B.

```
int Bar(int I, int J) {
    int   M, N, P;
    int   Foo(int, int *);
    ...
    P = Foo(M, &N);
    ...
    return (M);
}

int Foo(int X, int *Y) {
    int   A, B = 15;
    ...
    A = X;
    ...
    *Y = B;
    ...
    return (A);
}
```

Figure 2: C functions Bar and Foo

Activation frames are created on the stack in the order of procedure calls. The current procedure's activation frame is accessible by the frame pointer, typically a dedicated register in the processor. In MIPS convention, the frame pointer is $30, while the stack pointer is $29. The use of a frame pointer allows compiler generated code to describe an unambiguous location for variables, regardless of how many calls to the procedure may be active at one time. For example, in recursive procedures, the stack may contain many activation frames for the same procedure; each containing a complete set of required storage locations and a corresponding frame pointer. Figure 3 shows the state of the stack just before Bar initiates the call of Foo.
At this point, it would be possible to use the stack pointer access all activation frame slots. However there are several circumstances in which additional values are stored on the stack during Bar's execution, which would change stack offsets. Since the position of the frame pointer relative to the activation frame does not change, procedure variables can always be easily accessed relative to it.

When Bar calls Foo, an activation frame must be allocated and initialized for Foo's input parameters, return value, and local variables. Since the compiler does not always have a called procedure's code when the calling procedure is processed, the natural place to put code to generate the activation frame is at the beginning of the procedure that will use the new frame. In this example, one might assume that Foo will create its own activation frame. This approach presents problems since Foo's input parameters must be initialized by Bar before Foo is called. If Foo deallocates its activation frame at its completion (when control is returned to Bar), Foo's return value will not persist for Bar to access it. For this reason, activation frames are created in two parts: a caller generated part and a callee generate part. In this example, before Bar calls Foo, it allocates and initializes the caller portion of Foo's activation frame containing copies of passed parameters and a slot for the return value. Then Foo is called. When Foo begins execution, it creates the callee generation portion of the activation frame which contains room for local variables. Since the calling of Foo will overwrite Bar's frame pointer ($30) and return address ($31), a copy of these values must be preserved on the stack before Foo's activation frame is created. Figure 4 shows the stack just before Foo is called by Bar and after Foo is called and has generated the complete activation frame. If Foo should call another procedure, for example Bar, then another activation frame would be created on the stack below Foo's.

Once Foo completes its execution, it places its return value in the last slot of the caller generated part of its activation frame. Then it deallocates the callee generate part which it created and returns to Bar. After restoring its frame pointer and return address, Bar can access Foo's returned value and deallocate the caller generate part of Foo's activation frame.
MIPS assembly code for the calling of Foo by Bar is shown in Figure 5. It begins by allocating space for copies of Bar’s frame pointer and return address plus storage for Foo’s caller generate activation frame. As integers and pointers are four bytes long, this requires $5 \times 4 = 20$ bytes. Then Bar’s frame pointer and return address are saved.

```
statements in Bar
  addi $29, $29, -20  # allocate stack space
  sw $30, 16($29)  # preserve Bar’s FP
  sw $31, 12($29)  # preserve Bar’s RA
  lw $1, -4($30)   # copy M into temp
  sw $1, 8($29)   # store in Foo’s AF
  addi $1, $30, -8  # compute EA of N
  sw $1, 4($29)   # store in Foo’s AF
  jal Foo          # call Foo
  lw $30, 16($29)  # restore Bar’s FP
  lw $31, 12($29)  # restore Bar’s RA
  lw $1, 0($29)   # load Foo’s RV
  sw $1, -12($30) # store in P
```

Figure 5: MIPS code for Bar’s call of Foo

In the C language, parameter passing is normally call-by-value. So a copy of M is stored in the X slot of Foo’s activation frame. Since Y is a pointer to support call
by reference, the effective address of N is computed and stored in the Y slot. This pointer can be used by Foo to directly affect the storage in Bar’s activation frame. Finally Foo is called using a jump and link instruction. When control returns from Foo, Bar restores its frame pointer and return address. Then it copies Foo’s return value from Foo’s activation frame into its own as P. Finally, Foo’s caller generated activation frame and preservation slots are deallocated. Note that when Bar is accessing its own storage, it uses its frame pointer. But when Bar is building Foo’s activation frame, it uses the stack pointer.

MIPS code for Foo is shown in Figure 6. It begins by setting Foo’s frame pointer to the current stack pointer. This is the bottom slot of the activation frame created by Bar (the return value). Foo then extends the frame with space for its local variables, in this example A and B. The remainder of Foo’s code references operands using the frame pointer. B is initialized to 15 and then the statements of Foo are executed. When \( A = X; \) is executed, these variables are accessed in the activation frame. Caller accessible variables have positive offsets while local variables have negative offsets. When \( *Y = B; \) is executed, a value is stored using the pointer Y which points to N in Bar’s frame. This mechanism supports call-by-reference parameter passing in the C language. At Foo’s end, the return value slot is set and the stack space is deallocated.

```mips
Foo:  add   $30, $29, $0  # set Foo’s FP
      addi  $29, $29, -8  # allocate stack space
      addi  $1, $0, 15    # load init constant
      sw    $1, -8($30)   # B = 15
statements in Foo
      lw    $1, 8($30)    # load X
      sw    $1, -4($30)   # A = X
statements in Foo
      lw    $1, -8($30)   # load B
      lw    $2, 4($30)    # load Y
      sw    $1, 0($2)     # store B in Bar’s N
statements in Foo
      lw    $1, -4($30)   # load A
      sw    $1, 0($30)    # return (A)
      add   $29, $30, $0  # deallocate stack space
      jr     $31           # return to caller
```

**Figure 6:** MIPS code for Foo