Determining Stack Offsets

Prof. James L. Frankel
Harvard University

Version of 6:44 PM 10-Apr-2018

Stack Frame Offsets

- Lowest offsets (Top of Stack): Callee's argument build area
  - For excess arguments to be passed to subroutines we call
- Possible unused word for double-word alignment
- General register save area
  - Return address ($ra)
  - Old frame pointer ($fp)
  - $s0 through $s7
  - $a0 through $a3
- Temps
  - Used by compiler-generated code to save values in $t registers or when not all data fits in registers (*i.e.,* when any registers are spilled)
- Highest offsets: Local (automatic) variables

Callee's argument build area

- Offsets start at +0
- The argument build area must be at the lowest addresses in the caller's stack frame
  - This means that these arguments will be just past the stack frame allocated for the callee
- If we need to access arguments on the stack that have been passed to us, they have offsets just greater than the offsets for any data in our stack frame
- The argument build area is used only when there are more than four words of parameters to be passed
  - The first four words of parameters are passed in $a0 through $a3 (rather than in the argument build area)

Unused word for double-word alignment

- If the stack frame size is not a multiple of eight bytes, this word is reserved in the stack frame
  - It guarantees that all stack frames are double-word aligned

- If *addr* is an integral value to be rounded up to a multiple of eight, use the following expression
  - ((addr+7)/8)*8

    or equivalently
  - ((addr+7) >> 3) << 3

General register save area

- Always reserve space for $a0 through $a3, the return address ($ra), and old frame pointer ($fp)
- The return address and old frame pointer are always saved into the stack frame
- The MIPS calling conventions call for argument registers to be saved into the stack frame only if
  - A particular argument register is used for a passed parameter

    AND
  - Our subroutine is not a leaf subroutine
    - That is, it calls at least one subroutine
  - However, in our compilers, we will always save all argument registers into the stack frame if they are used for a passed parameter (even if the subroutine is a leaf subroutine)
    - This allows the saved argument registers in the stack frame to be accessed in exactly the same way that a local variable would be accesses
- All $s registers (that are modified) are saved into the stack frame
  - In our compilers, we will always save all of the $s registers into the stack frame

Additional Simplifications for Your Initial Compiler

- Always leave room in the stack frame for all $t registers ($t0 through $t9)
- Before each function call, save all $t registers
- After each function call, restore all $t registers
- Use all $s and then all $t registers when you assign registers

- Don’t try to implement any functions that require more than four words for parameters

- At the start of each statement, reset the register number to which temporaries are assigned

Possible Later Optimizations

- Analyze which $t registers are live across calls and
  - Allocate space in the stack frame for only those $t registers that are live across any calls
  - For each call, only save and then restore those $t registers that are live across that call
- Allow more than four words of arguments to functions
  - In each function, determine the maximum number of words required for function calls
  - Allocate space in the Argument Build Area for the maximum number of words in excess of four
  - For each function call, store all arguments in excess of four in the Argument Build Area
- Implement a better algorithm for temporary to register assignment – perhaps graph coloring

Stack Frame Limitation

- Because we're accessing variables in the stack frame using an offset off the frame pointer ($fp), our address range is limited
  - The offset field is sixteen bits in length

Temps

- If the compiler does not have sufficient registers for its code generation, space is allocated here for registers contents to be saved and restored
- This use of memory to save and restore values in registers is called *spilling*

Local (automatic) variables

- Offsets for local (automatic) variables will be assigned in the order in which variables are defined
- Variables in blocks at the same nesting level inside functions will share memory
- Ensure word alignment for word-size variables
- Ensure half-word alignment for half-word-size variables
- There is no required alignment for byte-size variable

Local (automatic) variables example

```
int main(void) {  /* If n is the beginning offset for local variables, … */
  int l0i, l0j, l0k;  /* l0i @ offset +n; l0j @ offset +n+4; l0k @ offset +n+8 */
  char l0c1;          /* l0c1 @ offset +n+12 */
  short int l0s1;  /* l0s1 @ offset +n+14 – NOTE: this offset is not +n+13 */
  short int l0s2;  /* l0s2 @ offset +n+16 */
  {
  int l1i;            /* l1i @ offset +n+20  – NOTE: this offset is not +n+18 */
  }
  {
  int l1j;            /* l1j @ offset +n+20  – NOTE: this offset is not +n+18 */
    {
    int l2i;                  /* l2i @ offset +n+24 */
    }
    {
    int l2j;                  /* l2j @ offset +n+24 */
    }
  }
}                             /* Total bytes for local variables for main: 28 */
```

Forcing word or half-word alignment

- If *addr* is an integral value to be rounded up to a multiple of four, use the following expression
  - ((addr+3)/4)*4

    or equivalently
  - ((addr+3) >> 2) << 2

- If *addr* is an integral value to be rounded up to a multiple of two, use the following expression
  - ((addr+1)/2)*2

    or equivalently
  - ((addr+1) >> 1) << 1

Forcing arbitrary alignment

- If *addr* is an integral value to be rounded up to a multiple of *multiple*, use the following expression
  - ((addr+multiple-1)/multiple)*multiple