Mips Stack Examples

Peter Rounce
P.Rounce@cs.ucl.ac.uk

MIPS Stack Example

The '*bottom*' of the stack is anchored in one place – top moves.

The '*top*' of the stack is defined by the contents of a register, *sp*.

accessing a data value within the stack

lw $7,8($sp)

sp Register 00EFFFD8

top of stack

Stack

00EFFFFC	97	B6	2A	78
00EFFFF8	08	E2	31	49
00EFFFF4	F3	7C	00	00
00EFFFF0	FF	FF	FF	FF
00EFFFEC	12	34	56	78
00EFFFE8	08	E2	56	56
00EFFFE4	F3	7C	56	56
00EFFFE0	FF	FF	FF	FF
00EFFFDC	57	A1	00	29
00EFFFD8	12	34	56	78
00EFFFD4	28	71	55	72
00EFFFD0	FF	FF	FF	F8

4 locations (4 bytes)

Where sp points is the top of the stack!

Memory at address 0xefffe0 (0xefffd8 + 8) is accessed and 4 bytes read from this address and the following 3 addresses into register $7

A ***C*** Function (method):

```
int fun1(int p1, double p2, int p3 ) {
int i , j, k, res;
double r1;
r1 =  fun2( p1*p2+k, j, &i) ;
…..
return res ;
}
```

A call to the function:

r1 = fun1(10, 11, 12) ;

MIPS assembler for the call (ignoring the parameters for now):-

```
jal     fun1    #   $31 <-- pc;   pc <- address of fun1
```

MIPS assembler for the return (ignoring the return value for now):-

```
jr      $31     # pc <-- $31
```

? what about the function call to fun2 within sub?

? what to do with return address?

Choices for dealing with *return address* for *nested* function calls

1) use a different register to hold return address at each level
e.g. jalr $30, fun2 # $30 <-- pc ; pc <- address of fun2
only useful in handwritten assembler – not for compiled code!

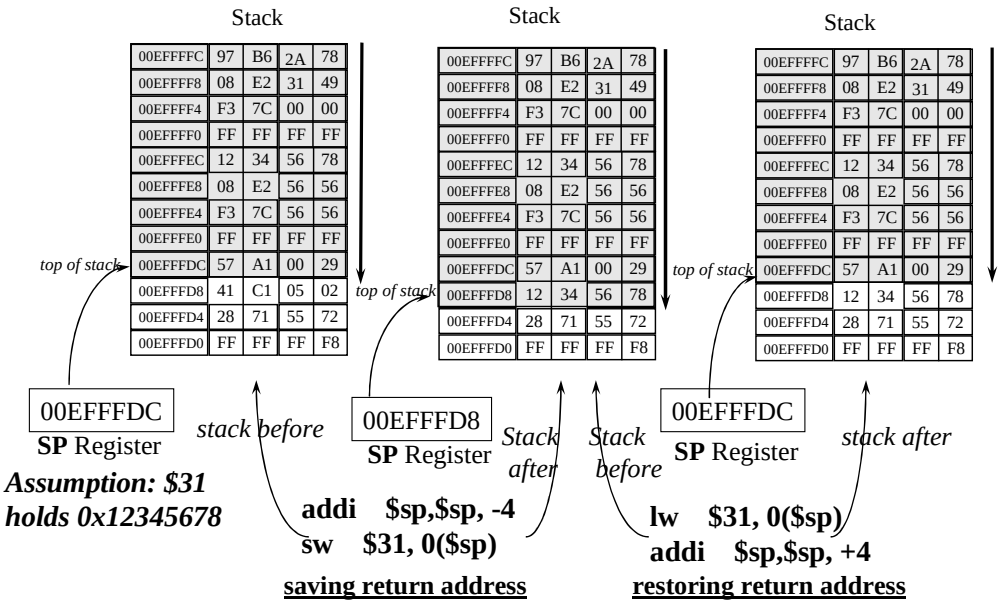
2) In each function, take return address from $31 into a defined memory location freeing $31 for use for calls within this function. Reload $31 just before return from function.
- no good for recursive function calls
-space is allocated even when function is not being used!

3) save return address into memory but use an extendable memory area
- a STACK

A Stack is a standard feature in computers. Supports recursion!

Item 1) Using a different register to hold return address at each level; there are difficulties in implementation : can run out of registers if there are many calls
Need to make sure that calling function uses the correct register for the called function.
If there are several calls to a function from different places in the code: all have to use the same register for the return address! Could be at different code levels – program writer might have to know how the hardware works!!!!!!
Cannot make recursive function calls if do this!

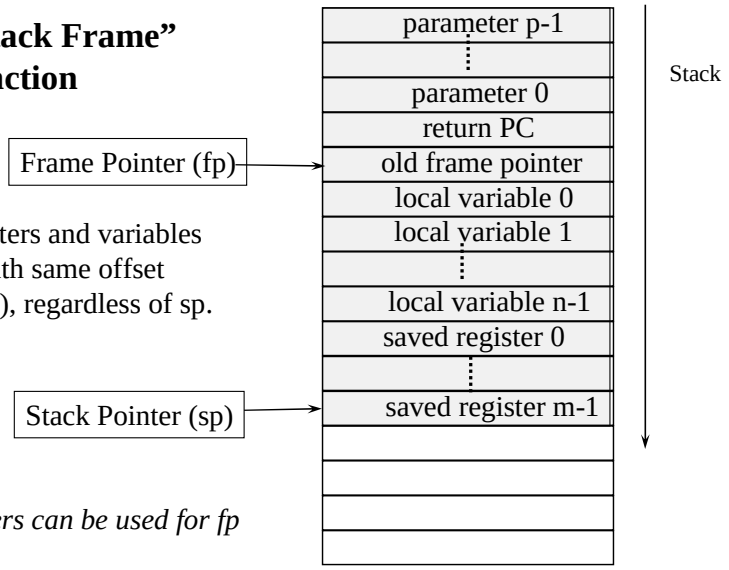
MIPS Stack Example: saving return/returning address in $31 to/from stack



Note: can save any register value on to stack - can use stack as a temporary store

Stack is used in C to store function parameters and local variables plus…...

This is the "Stack Frame" of a function



fp allows parameters and variables to be accessed with same offset from fp, e.g. 8(fp), regardless of sp.

Any 2 MIPS registers can be used for fp and sp

Return PC, old frame pointer, local variables etc place on stack by called function!

10/13/2011 09-GC03 Stacks 6

Parameters are pushed on the stack by the calling subroutine.
Parameters are accessed by called subroutine by displacement mode addressing via FP, e.g +4(FP).

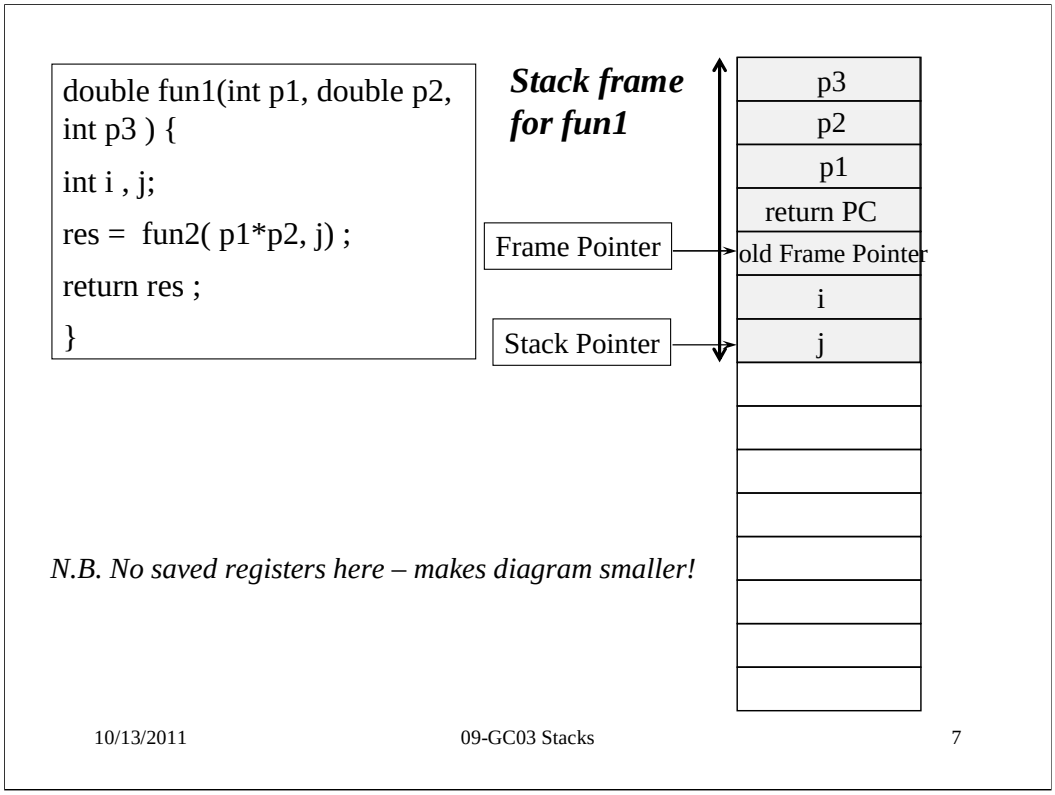
Space is created on the stack for Local Variables at the start of the function.
Local Variables are accessed like parameters via fp but using negative displacements, e.g. -4(fp).

Sometime CPUs registers are saved the stack to leave registers free for usage by subroutine. In particular, it is usual in a subroutine to have some local variables stored in registers, while the rest are stored on the stack. It is often necessary to keep the values of local variable across function calls, so the values of those stored in registers must be saved on the stack across function calls.
The choice for local variable storage is to store them in registers first and only then to use the stack: registers are quicker to access.
When there is a choice to be made as to which local variables go into registers: it should be the most used ones, although often it is the first declared in the function. If registers are used for local variables, then the same registers will be used by all functions; when a function is entered it must save contents of those registers it uses for local variables on the stack as their contents must be preserved in case the calling function or one of the ones that called it is using these registers for its local variables.

Within a function, the top of the stack is free for use for temporary storage of values and by further function calls.

Some parameters may go into registers for faster access, while others go on the stack. The stack allows for very large numbers of parameters to be used without running into resource problems.

Typically in a Mips system, there might be up to 4 local variables in registers and 2 parameters in registers. Although faster to access these registers are a liability to some degree because on a function call, they have to be saved on the stack in the called function if the registers are to be used again in the called function.

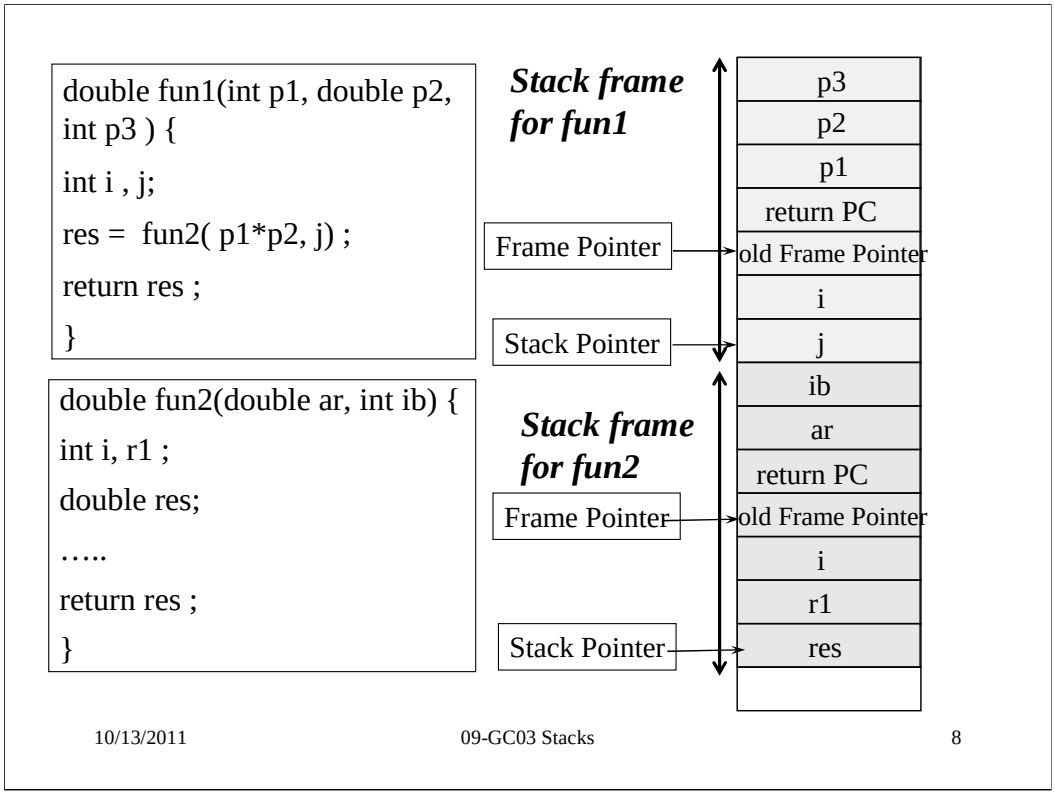


Frame pointer register points at a fixed point within the stack frame.

This allows parameters to be referenced with the same offset throughout function execution, i.e, +8($fp) accesses p1.

Similarly, local variables are referenced with the same offset throughout function, -4($fp) for ‘i’.

The value in $sp in the ‘calling function’ is remembered within the ‘called’ function by storing it on the stack as ‘Old Frame Pointer’ – this value will be restored to $fp at the end of the’called’ function.

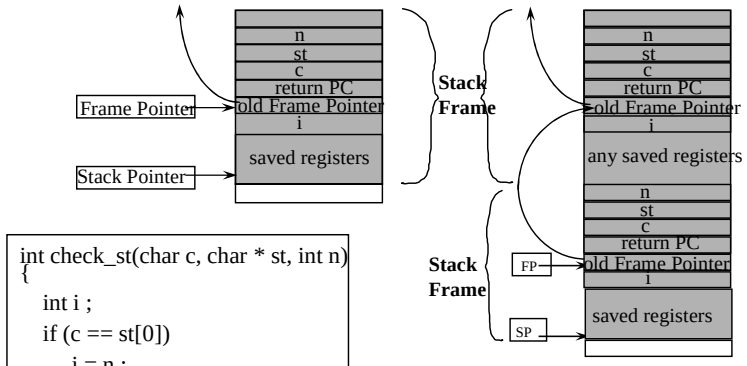


1) Can re-use names for local variables without difficulty, since get new space for local variables on each call!
2) Before sub 2 returns, its stack frame is collapsed:
   - the old frame pointer is placed back into the Frame Pointer, pointing it back into the correct place within the stack frame for sub1;
   - the stack is shrunk so that sub2 local variables are no longer in the stack (their values must be considered lost, since later calls and other uses of the stack will overwrite their locations).
   - The return pc is popped from the stack and the return is made.
3) On re-entering the calling function, the parameters are still on the stack to be removed by the calling function, and the stack pointer should be pointing at the same stack location as when the call was made.

A stack uses space effectively since the same space is re-used over and over again by stack frames for different functions.

Recursive function calls are possible – functions that call themselves – because there is a new stack frame on every call, so that there are a new set of parameters, and a new set of local variables. For a useful recursive function see the Java Linked List code of earlier notes.

Recursive Function calls: function calls itself




```
int check_st(char c, char * st, int n)
{
   int i ;
   if (c == st[0])
      i = n ;
   else
      i = check_st(c, &st[1], n+1) ;
   return i ;
}
```

Note: new storage for local variables is allocated on each call.

Outline code for function call check_st(c, &st[1], n+1):-

in calling function
- push value of ‘c’ on to stack
- push address of location of array element st[1] on to stack
- push value of n+1 on to stack
- jump&link (jal) to function check_st

in called function
- push return pc to stack from register where it is saved
- push current value of FP register to stack (becomes Old FP)
- copy current address of top of stack from sp register to FP register
- create space on top of stack for local variables, e.g. add $sp, $sp, -12
- push the contents of any registers needing saving to stack
- store any initial values for local variables on stack or in registers

Outline code for function return:-

in called function
- place any return value into the appropriate register
- reload contents of any saved registers from stack
- copy contents of $fp to $sp – effectively removes local variables from stack
- pop ‘Old FP’ from stack into $fp – set $fp t opoint back into calling stack frame
- pop ‘return PC’ into register
- return to calling function

in calling function
- pop parameters from stack – called function knows exactly how many there are.

On each recursive call, a new stack frame is created with a whole new set of local variables and parameters, which hide the previous sets in early stack frames.