Computer Architecture. Week 7

Alexander Tormasov

Innopolis University

a.tormasov@innopolis.ru

October 07, 2021

Topic of the lecture

- Functions and Function Calls

Content of the Class

- Supporting Functions in Computer Hardware
- Calling Conventions
- The Stack and Functions
- Recursive Function
- Structural Variations for Functions
- MIPS Conventions for Functions
- Nested Functions
- Inline Functions
- Macros
- Summary

Supporting Functions in Computer Hardware

- A function is a tool that programmers use to structure programs, to **make it easier to understand** and allow **code to be reused.**

- Functions allow the programmer to **concentrate on just one portion** of the task at a time

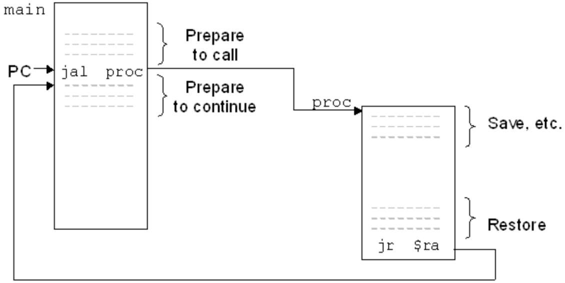
- **Parameters act as an interface between the function** and rest of the program.

- Functions are one way to implement **abstraction.**

Importance of Functions

- A set of instructions (a subroutine) performing a definite **task, which is completely independent from the main program flow** and communicates using input values and returning outputs.

- It helps to structure the program and **function can be reused.**



Calling Conventions (1/2)

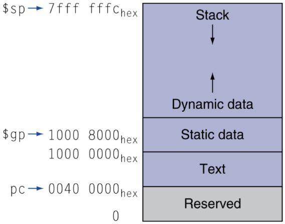
- Calling conventions are **standardized method for functions** to be implemented and called by the machine.
- It is a supportive way to make the common case fast.
- Caller and callee need to agree
- Enforced by compiler
- Important when using third party libraries
- Different styles ⟷ different advantages

NOTE: It is possible to have a single physical computer but different compilers can have different calling conventions.

Calling Conventions (2/2)

- Calling conventions specify:
  - how arguments are passed to a function,
  - how return values are passed back out of a function,
  - how the function is called, and
  - how the **function manage the stack and its stack frame.**
- Stack frame is not always required. An optimizing compiler might in some cases avoid using the call stack. Notably when it is able to inline a called function (Details later in this lecture)

Memory Layout – Stack



- **Text:** program code
- **Static data:** global variables
  - For Example: static variables in C, constant arrays and strings
  - $gp initialized to address allowing ± offsets into this segment
- **Dynamic data:** heap
  - E.g., malloc in C, new in Java
- **Stack:** automatic storage

Stack Frame Principles

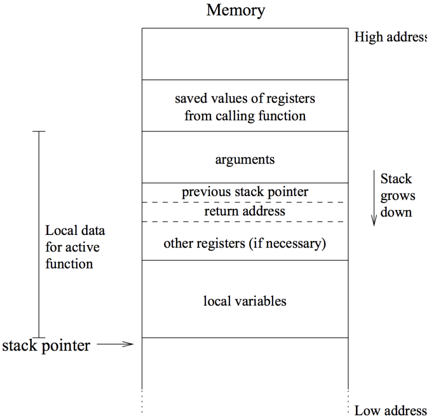
Stack frame supports function calls in a program, to control the execution flow of a program, including

- to pass input arguments to a called function;
- to specify the memory address of the first instruction of a function called;
- to store return value;
- to store the return address (the address of the instruction to be executed after the completion of a called function)
- other (e.g. to store values of local variables of a function, which called another function)

Frame - a collection of data associated to an individual function call.

Stack Frame Principles

- On entry to a function:
- 1. compute size of stack frame and new stack pointer
- 2. allocate memory for frame
- 3. save registers
- 4. store arguments
- 5. save previous stack pointer and return address
- 6. change stack pointer
- 7. pass control to new function



Spilling Registers

- The CPU has a **limited number of registers** for use by all functions, and it's possible that **several functions will need the same registers.**
- We can keep important registers from being overwritten by a function call, by **saving them before the function executes, and restoring them after the function completes**.

Register Conventions

- **What do these conventions mean?**

  A. If function A calls function B, then function A must save any temporary registers that it may be using onto the stack.

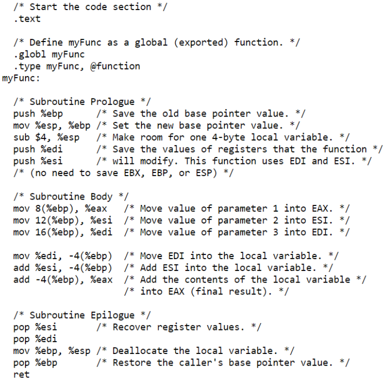
  B. Function B must save any s (saved) registers that intends to use before garbling up their values

  C. Remember: Caller/callee need to save only temporary/saved registers they are using, not all registers.

Reference: Previous lecture (Temporary and other registers)

Calling Conventions

- In practice, many calling conventions are possible.
- x86 Function calls that follow its convention rules.



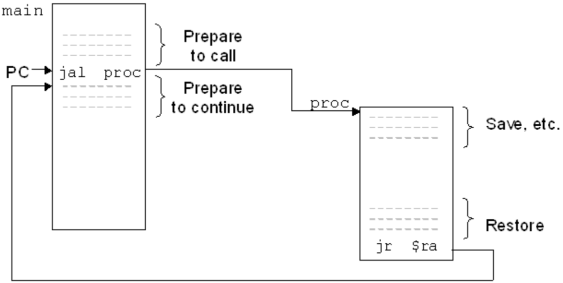
```
  /* Start the code section */
  .text

  /* Define myFunc as a global (exported) function. */
  .globl myFunc
  .type myFunc, @function
myFunc:

  /* Subroutine Prologue */
  push %ebp      /* Save the old base pointer value. */
  mov %esp, %ebp /* Set the new base pointer value. */
  sub $4, %esp   /* Make room for one 4-byte local variable. */
  push %edi      /* Save the values of registers that the function */
  push %esi      /* will modify. This function uses EDI and ESI. */
  /* (no need to save EBX, EBP, or ESP) */

  /* Subroutine Body */
  mov 8(%ebp), %eax   /* Move value of parameter 1 into EAX. */
  mov 12(%ebp), %esi  /* Move value of parameter 2 into ESI. */
  mov 16(%ebp), %edi  /* Move value of parameter 3 into EDI. */

  mov %edi, -4(%ebp)  /* Move EDI into the local variable. */
  add %esi, -4(%ebp)  /* Add ESI into the local variable. */
  add -4(%ebp), %eax  /* Add the contents of the local variable */
                      /* into EAX (final result). */

  /* Subroutine Epilogue */
  pop %esi       /* Recover register values. */
  pop %edi
  mov %ebp, %esp /* Deallocate the local variable. */
  pop %ebp       /* Restore the caller's base pointer value. */
  ret
```

More details:

http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html

Illustrating Function Calls

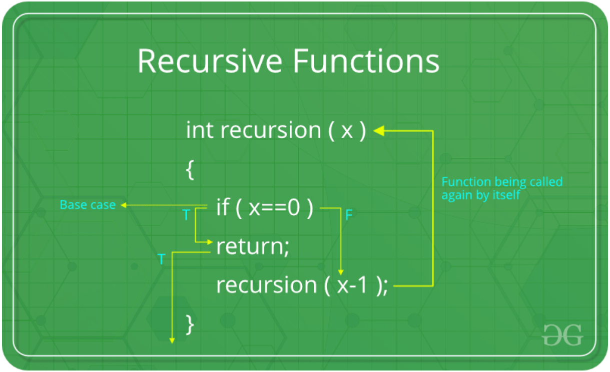
1. Place parameters in registers/memory
2. Transfer control to function
3. Acquire storage for local variables used in function
4. Perform function's operations
5. Place result in register/memory for caller
6. Return to place of call



Recursive Function

A recursive function – a function that calls itself:

- A function calls itself, with different arguments, until a terminating condition is met;
- Similar to a loop;
- The function call address remains unchanged



x86 Structural Variations for Functions

- There are a lot of different calling conventions: cdecl, syscall, fastcall, pascal, thiscall... Caller or callee can cleanup the stack.
- Due to the small number of architectural registers, the x86 calling conventions **mostly pass arguments on the stack**, while the return value (or a pointer to it) is passed in a register.
- Some conventions use registers for the first few parameters, which may **improve performance for short and simple leaf-routines** very frequently invoked.
- Registers are guaranteed to retain their values after a subroutine call. According to the Intel ABI to which the vast majority of compilers conform, the **EAX, EDX, and ECX are to be free for use** within a function, and do not need to preserve.

x86 Structural Variations for Functions

- **Example: Function Call**

```
push EAX            ; pass some register result
push byte[EBP+20]   ; pass some memory variable (FASM/
    TASM syntax)
push 3              ; pass some constant
call calc           ; the returned result is now in EAX
```

- **Typical callee structure**

```
calc:
     push EBP           ; save old frame pointer
     mov EBP,ESP        ; get new frame pointer
     sub ESP,localsize  ; reserve place for locals
     .
     .    ; perform calculations,leave result in EAX
     .
     mov ESP,EBP      ; free space for locals
     pop EBP          ; restore old frame pointer
     ret paramsize    ; free parameter space and return
```

Register Usage for the Functions (MIPS)

- **Remember! Registers used:**
  - \$a0 – \$a3: arguments (registers 4 – 7)
  - \$v0, \$v1: result values (registers 2 and 3)
  - \$t0 – \$t9: temporaries (it can be overwritten by callee)
  - \$s0 – \$s7: saved (Must be saved/restored by callee)
  - \$gp: global pointer for static data (register 28)
  - \$sp: stack pointer (register 29)
  - \$fp: frame pointer (register 30)
  - \$ra: return address (register 31)
- NOTE: In case of more arguments, then we can move them to stack but do not heavily loaded the function.

MIPS64 Instruction set format

I - type instruction

6	5	5	16
Opcode	**rs**	**rt**	**Immediate**
0 5	6 10	11 15	16 31

Encodes: Loads and stores of bytes, words, half words. All immediates (rd ← rs op immediate)
Conditional branch instructions (rs1 is register, rd unused)
Jump register, jump and link register (rd = 0, rs = destination, immediate = 0)

R - type instruction

6	5	5	5	5	6
Opcode	**rs**	**rt**	**rd**	**shamt**	**func**
0 5	6 10	11 15	16 20	21 25	26 31

Register-register ALU operations: rd ← rs func rt Function encodes the data path operation: Add, Sub .. Read/write special registers and moves.

J - Type instruction

6	26
Opcode	**Offset added to PC**
0 5	6 31

Jump and jump and link. Trap and return from exception

MIPS Convention for Functions

- Registers are the **fastest place to hold data** in a computer, so we want to use them as much as possible.

- MIPS follows the following convention for function calling in allocating its 32 registers

Return address	\$ra
Arguments	\$a0, \$a1, \$a2, \$a3
Return values	\$v0, \$v1
Local variables	\$s0, \$s1, . . . , \$s7

Function Call

- MIPS includes **Jump and Link (jal)** instruction for function call.
- It jumps to an address and **simultaneously saves the address of the following instruction** in register \$ra.
- Syntax:

$$jal \ \ FunctionAddress$$

- The "link" is stored in **register \$ra** known as return address.
- The **return address is needed** because the same function could be called from several parts of the program.

Function Return

- MIPS use jump register (jr) instruction for **unconditional jump to the address specified in a register.**

- Syntax:

$$jr \quad \$ra$$

- Thus, the **caller** puts the parameter values in $a0 – $a3 and uses $jal\ X$ to jump to function X

- The **callee** then performs the calculations, places the results in $v0 and $v1, and returns control to the caller using jr $ra

Function Example (1/3)

- C code

```
int leaf_example (int g, int h, int i, int j)
{
int f;
f =      (g + h) - (i + j);
return f;
}
```

```
 Arguments g,..., j in $a0, ... , $a3
 f in $s0 (hence, need to save $s0 on stack)
 Result in $v0
```

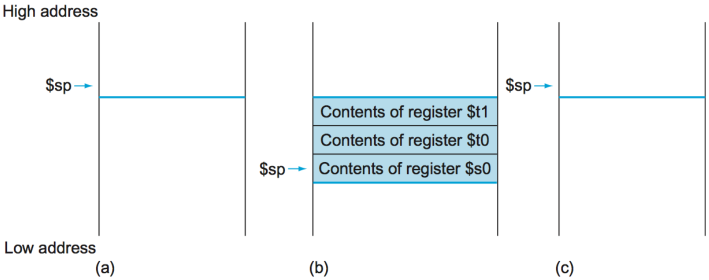
Function Example (2/3)

- MIPS Assembly Code (Solution I)
- Pay attention: violation of standard calling convention!

```
leaf_example:
    addi $sp, $sp,-12 # adjust stack to make room for 3 items
    sw $t1, 8($sp)    # save register $t1 for use afterwards
    sw $t0, 4($sp)    # save register $t0 for use afterwards
    sw $s0, 0($sp)    # save register $s0 for use afterwards
    add $t0,$a0,$a1   # register $t0 contains g + h
    add $t1,$a2,$a3   # register $t1 contains i + j
    sub $s0,$t0,$t1   # f = $t0 - $t1, which is (g + h)-(i + j)
    add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)
    lw $s0, 0($sp)    # restore register $s0 for caller
    lw $t0, 4($sp)    # restore register $t0 for caller
    lw $t1, 8($sp)    # restore register $t1 for caller
    addi $sp,$sp,12   # adjust stack to delete 3 items
    jr   $ra          # jump back to calling routine
```

NOTE: We are restoring the temporary register. While it can be overwritten by callee.

Function and Memory



- The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the “top” of the stack, or the last word in the stack in this drawing.

Function Example (3/3)

- MIPS Assembly Code (Solution II)

```
leaf_example:
     addi $sp, $sp, -4
     sw $s0, 0($sp)           # Storing $s0
     add $t0, $a0, $a1
     add $t1, $a2, $a3
     sub $s0, $t0, $t1
     add $v0, $s0, $zero     # Body of the function
     lw $s0, 0($sp)          # Restore $s0
     addi $sp, $sp, 4        # set back the pointer
     jr $ra
```

- NOTE: We are overwriting the temporary register.

Passing Arguments

- Recall: arguments to a function passed through $a0-$a3
- **What if the function has > 4 arguments?**
  - First four arguments are put in $a0-$a3
  - Remaining arguments are **put on stack by the caller**
- **Example:** myFunction(int x0, int x1, . . . , int x7)
  - Caller places arguments x0-x3 in $a0-$a3
  - Caller places arguments x4-x7 on stack

Return Values

- Recall: return values from a function passed through \$v0-\$v1
- **What if the function has > 2 return values?**
  - First two return values are put in \$v0-\$v1
  - Remaining return values are put on stack by the function
  - The remaining return values are popped from the stack by the caller

The MIPS stack

- In MIPS machines, **part of main memory is reserved for stack**
    - The stack grows downward in terms of memory addresses.
    - The address of the top element of the stack is stored (by convention) in the "stack pointer" register $sp.
- MIPS does not provide **"push" and "pop"** instructions.
- Instead, they must be done **explicitly by the programmer**.
- The stack "grows" from **higher addresses to lower addresses**.

MIPS Calling Convention (1/2)

- **Stack Management**
  - The stack grows down.
    - Subtract from $sp to allocate local storage space.
    - Restore $sp by adding the same amount at function exit.
  - The stack must be 4-byte aligned.
    - Modify $sp only in multiples of four.

MIPS Calling Convention (2/2)

- **Function Parameters**
  - Every parameter smaller than 32 bits is promoted to 32 bits.
  - First four parameters are passed in registers $a0−$a3.
    - 64-bit parameters are passed in register pairs.
  - Every subsequent parameter is passed through the stack.
    - First 16 bytes on the stack are not used.
    - Assuming $sp was not modified at function entry.
    - The 1st stack parameter is located at 16($sp).
    - The 2nd stack parameter is located at 20($sp), etc.
    - 64-bit parameters are 8-byte aligned
- **Return Values**
  - 32-bit and smaller values are returned in register $v0.
  - 64-bit values are returned in registers $v0 and $v1

MIPS Register Conventions (1/3)

- When callee returns from execution, the caller needs to know which **registers may have changed and which are guaranteed to be unchanged.**

- **Register Conventions:** A set of generally accepted rules as to which registers will be unchanged after a procedure call (jal) and which may be changed.

Register Conventions (2/3)

- **$0: No Change.** Always 0.
- **$v0-$v1: Change.** These are expected to contain new values.
- **$a0-$a3: Change.** These are volatile argument registers.
- **$t0-$t9: Change.** That's why they're called temporary: any procedure may change them at any time
- **$s0-$s7: No Change.** Very important, that's why they're called saved registers. If the callee changes these in any way, it must restore the original values before returning.
- **$sp: No Change.** The stack pointer must point to the same place before and after the jal call, or else the caller won't be able to restore values from the stack.
- **$ra: Change.** The jal call itself will change this register.

Register Conventions (3/3)

- Note that, if the **callee is going to use some s registers**, it must:
  - save those s registers on the stack
  - use the registers
  - restore s registers from the stack
  - jr $ra
- With the **temp registers**, the callee doesn't need to save onto the stack.
- Therefore the caller must save those temp registers that it would like to **preserve through the call.**

Other Registers

- Note that, if the **callee is going to use some s registers**, it must:
  - $at: may be used by the assembler at any time; **unsafe to use**
  - $k0-$k1: may be used by the kernel at any time; **unsafe to use**
  - $gp: don't worry about it
  - $fp: we have seen

Things to Remember

- Functions are called with jal, and return with jr $ra.
- The **stack is your friend**: Use it to save anything you need.
- The **stack is your enemy**: Just be sure to leave it the way you found it.
- Instructions we know so far

```
Arithmetic: add, addi, sub, addu, addiu, subu, sll
Memory: lw, sw
Decision: beq, bne, slti
Unconditional Branches (Jumps): j, jal, jr
```

- Registers we know so far
  - All of them!

Nested Functions

- Functions that do not call others are called leaf functions.
- Nested functions are functions invoking other functions before the return.
- Nested functions are also known as non-leaf functions.

Example of Nesting (1/3)

```
int sumSquare(int x, int y)
{
     return mult (x,x) + y;
}
```

- Something called sumSquare, now sumSquare is calling mult.
- So there is a value in $ra that sumSquare wants to jump back to, but this will be **overwritten by the call to mult.**
- Need to save sumSquare return address before call to mult.

Example of Nesting (2/3)

- We have a register $sp which always **points to the last used space in the stack.**
- As we said, to use stack, we **decrement this pointer** by the amount of space we need and then fill it with info.
- **So, how do we compile this?**

```
int sumSquare(int x, int y)
{
     return mult (x,x) + y;
}
```

Example of Nesting (3/3)

```
sumSquare:
     addi $sp, $sp, -8        #space on stack
     sw   $ra, 4($sp)         # save ret addr
     sw   $a1, 0($sp)         # save y

     add  $a1, $a0, $zero     # mult(x,x)
     jal  mult                # call mult

     lw   $a1, 0($sp)         # restore y
     add  $v0, $v0, $a1       # mult() + y

     lw   $ra, 4($sp)         # get ret addr
     addi $sp, $sp, 8         # restore stack
     jr   $ra
```

Another Example (1/2)

```
int fact (int n)
{
      if (n < 1) return 1;
      else   return n * fact(n - 1);
 }
```

- Argument n in \$a0
- Result in \$v0

A Look at the Code (2/2)

```
fact:
  addi $sp, $sp, -8      # adjust stack for 2 items
  sw   $ra, 4($sp)       # save return address
  sw   $a0, 0($sp)       # save argument
  slti $t0, $a0, 1       # test for n < 1
  beq  $t0, $zero, L1
  addi $v0, $zero, 1     # if so, result is 1
  addi $sp, $sp, 8       #   pop 2 items from stack
  jr   $ra               #   and return
  L1: addi $a0, $a0, -1 # else decrement n
  jal  fact              # recursive call
  lw   $a0, 0($sp)       # restore original n
  lw   $ra, 4($sp)       #   and return address
  addi $sp, $sp, 8       # pop 2 items from stack
  mul  $v0, $a0, $v0     # multiply to get result
  jr   $ra               # and return
```

Summary of the Steps for a Call

1. Save necessary values onto stack.
2. Assign argument(s), if any.
3. jal call
4. Restore values from stack.

Summary of the Rules for Calls

- Called with a jal instruction, returns with a jr $ra

- Accepts up to 4 arguments in $a0, $a1, $a2, and $a3

- Return value is always in $v0 (and if necessary in $v1)

- Must follow **register conventions**, even in functions that only you will call!

Side effects of Functions

- Side effect = changing something somewhere.
- A function is said to have a side effect if it modifies some variables outside its local environment.
- Example: A particular function modify a non-local variable.

Macros

- It is like a function but operate differently
- It does not require the protocols and execution overhead of functions.
- For Example
  - System call 10 for exit (MARS IDE)

```
        li $v0,10
                syscall
```

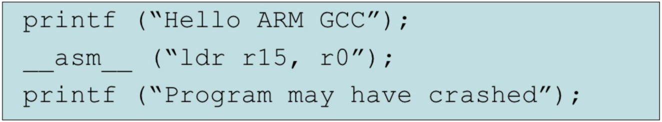
  - A macro "done"

```
.macro done
li $v0,10
syscall
.end_macro
```

  - Usage: invoke it whenever you wish with the statement "done"

Inline Functions

- GNU C compiler for ARM RISC processors offers, to embed assembly language code into C programs.
- This cool feature may be used for manually optimizing time critical parts of the software or to use specific processor instruction, which are not available in the C language.
- For Example:



```
printf (“Hello ARM GCC”);
__asm__ (“ldr r15, r0”);
printf (“Program may have crashed”);
```

NOTE:

Above code will corrupt the program counter, So use inline assembly carefully. It may lead to non portable code.

Summary

- Supporting Functions in Computer Hardware
- Using Functions (leaf functions)
- The Stack and Functions
- Calling Conventions
- Nested Functions (non-leaf functions)