CS47 - Lecture 12

Kaushik Patra
(kaushik.patra@sjsu.edu)

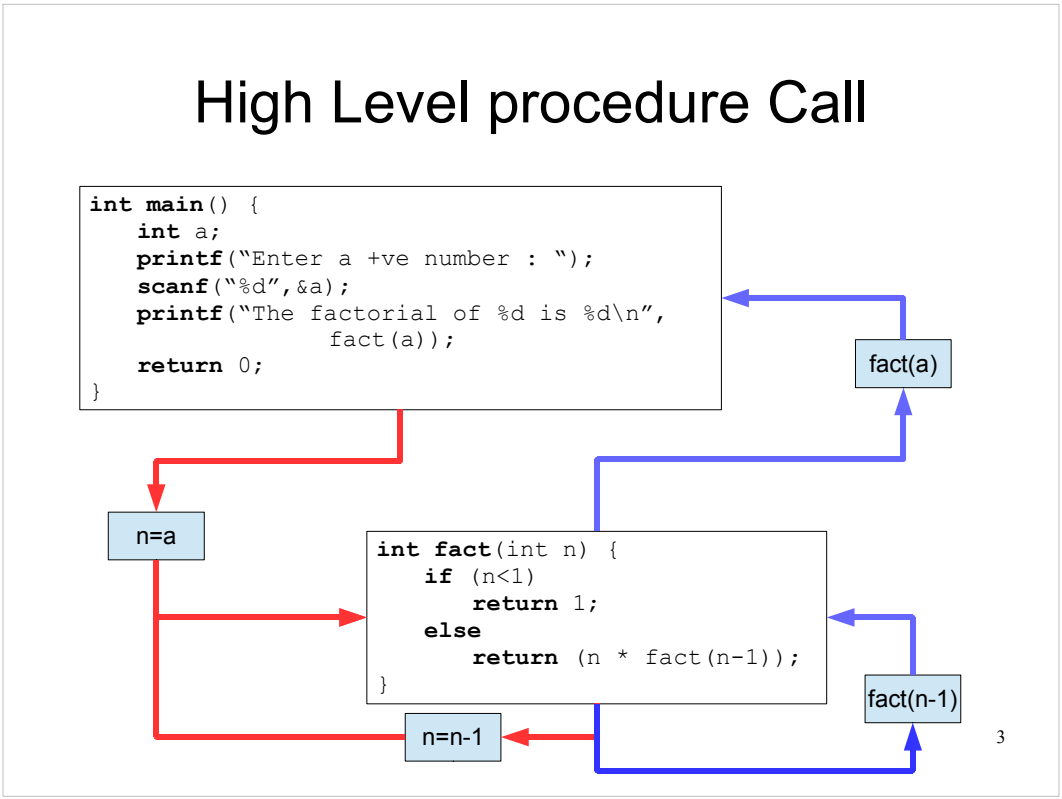
1

- Topics
  - Procedure Call
  - Low Level Procedure Call
  - Caller Run Time Environment
  - RTE Storage

[Chapter 2.7, Appendix A.6 of Computer Organization & Design by Patterson and Hennessy.]

Procedure Call ...

2



- The top level program calls the 'fact' functional routine to compute factorial of a given number a. It passes the parameter to the function and expect it return the result. The function does 'some' computation and returns the result.

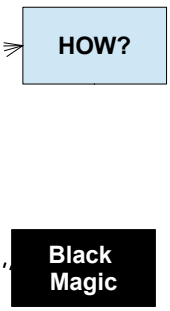
- From inside the function, it may call itself – it is called recursive function. Recursive function needs to have a base case to terminate recursion. The base case is the case where a function knows what is the result out of the given input.

- In this example, base case is n<1; where the result should be 1.

- The main function sets up the 'fact' parameter n to value a, and transfer the control to 'fact' function. The fact function returns 1 if the n < 1 otherwise it returns n * fact(n-1) [i.e. factorial of n is (n * factorial of n-1)]. This means, if n is 1 or more, this function calls itself with one less value of n hoping to reach a base case. Once the base case is hit, result is back to the previous step and multiplication of previous step's argument with the result is returned back from that stage. Finally the control is back to the main and result is 'returned' to it.

High Level procedure Call

- Function is like a 'black magic' to the caller. It passes the arguments and gets the result back.
  - Pass the arguments
  - Transfer the control to function
  - Perform the task
  - Return the result
  - Return the control to caller



- The expectation is that the caller's run time environment is not altered.

4

- From caller perspective function is like a 'black magic' – it does not need to know what it performs inside. The only expectation is that while doing its task the function should not alter any state of the caller. Callers passes the argument to function and expects the result returned from function without any modification to caller's run time environment. The analogy can be drawn from a home delivery pizza at movie night with friend. We call pizza house and tell them our order and then back to movie watching while waiting for them to deliver the pizza. Nothing is altered at home environment while pizza house does many steps to get the pizza out of oven and deliver to the door step. The result back from 'pizza house' function is the warm pizza – we are least bothered about that 'black magic' of creating good pizza.

- There are many questions on how the arguments are passed, how the control is passed to function. There are question regarding the process of returning the result and how the function preserves run time environment of caller.

Low Level Procedure Call ...

5

Processor Support for Function Call

- Function / Procedure call support in HW
  - Pass the arguments
    - Use **$a0-$a3** to pass the arguments
    - Use **stack** to store any additional arguments
  - Transfer the control to function
    - Use '**jal**' (Jump and link), stores the next instruction address in caller in **$ra**.
  - Perform the task.

6

- Function's arguments are passed from caller through four internal registers $a0-$a3. So four arguments are passed through the registers. If there is need to pass additional arguments, they are pushed into the stack. The argument order is predefined. The function assumes its argument in right place. This is the reason function needs to stick to its signature (the order of arguments in the high level function.)

- Next step from caller is to use jump & link (jal) instruction to jump to the start of the function code. This 'jal' instruction also records the next instruction address in caller to register $ra. The function uses this value to return to the caller at the next instruction position.

- Once in the function code, function performs whatever it needs to perform – caller does not need to know what it does. The only expectation is that caller environment is not changed. We'll discuss about this environment later.

Processor Support for Function Call

- Function / Procedure call support in HW
  - Return the result
    - Use **$v0** and **$v1** to return the result
    - It is usually is not desired to return more than two values from a function (usually it is one).
  - Return the control to caller
    - Use '**jr $ra**' to return to caller's next instruction.

7

- Once done, function places the result in $v0. In some extreme cases, if function needs to return a 2nd result it places in $v1. It is not desired to get back more than two results out of a function. If so, it is software engineering disaster and needs re-factoring of code. However high level function supports more than one return by using reference variable (or pointers in C). This is obtained by directly writing the results from function into the memory location specified by caller (passed as reference or pointer.)

- Once the result is placed into desired place, function returns control to caller using jump register (jr) instruction. This instruction takes register as argument and jumps to the location pointed by that register. Since at the beginning of the call, 'jal' instruction already records the return location, function uses 'jr $ra' to return the control to caller.

Caller Run Time Environment ...

8

Run time environment

- Caller's run time environment consists of
  - All the registers
  - Program counter (PC) pointing to next instruction in caller.
- To preserve caller's run time environment
  - Save all the register values and PC at beginning
  - Restore all the register values and PC before return
- It is too much to store all 32 registers

9

- At any moment of time a caller (or a program's) run time environment is the current value of all the registers in the processor and program counter (PC) pointing to the next instruction. At any point of time during program execution, we can start doing some unrelated task, as long as we save the run time environment. Once the task at hand is done, we can always comeback to the point by restoring the saved the run time environment without altering result of the original program. This concept is used at the procedure call mechanism and also at OS context switching to support time shared multiple process execution.
- At the beginning any function needs to save the caller RTE and at the end it needs to restore the caller RTE before the control is returned to caller.
- However, if each procedure call needs to save all the 32 registers and PC, it will be a performance hit. Each time it is involving 32 memory store and 32 memory load process which is very slow.

Run time environment

- It is sufficient to save the following registers
  - $a0-$a3 : The arguments, if used
  - $s0-$s7 : The saved registers, if used
  - $sp, $fp, $ra
- Any code should not use $gp, $k0, $k1, $at
- $zero is constant
- $v0, $v1 is used for returning result
- $t0-$t9 are not saved as convention

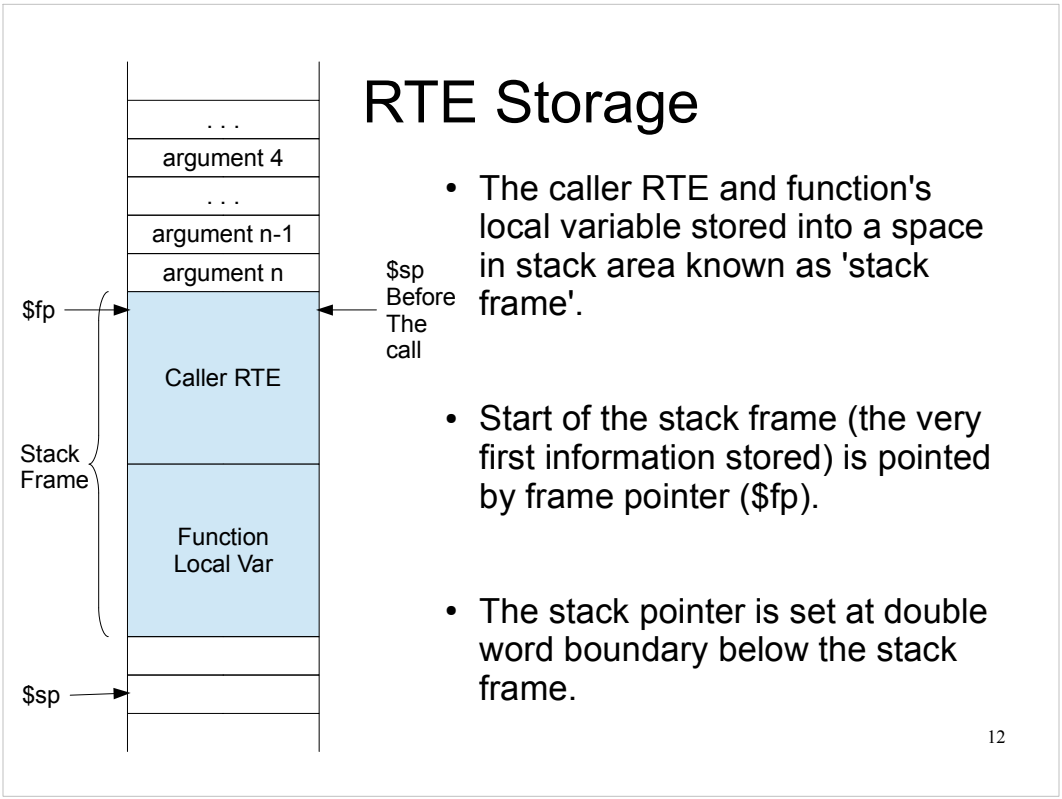
10

- Total 17 out of 32 registers does not need saving. Temporary registers ($t0-$t1) are not saved during function call by convention. We do not need to store $v0 and $v1 since they are used to store the return value, so their values can not be preserved. No program can alter the constant $zero register's value. On top of this, any user program should not alter values in $gp, $k0, $k1 and $at as programming convention ($gp should be pointing to middle of the static global data, $k0-$k1 to be used by OS kernel and $at to be used by assembler).

- We are down to total 15 $a0-$a3, $s0-$s7, $sp, $fp and $ra to be stored. If a function does not uses $a0-$a3 and $s0-$s7 it does not even needs to save them (even if calls other function – the called function guarantees to preserve these values across the boundary). A function always uses $sp (stack pointer) and $fp (frame pointer) to setup temporary storage area for the caller's run time environment and it's own local storage. Therefore they needed to be saved. Also a function can call another function. This means the function is using 'jal' instruction which will write latest return address in $ra. Therefore, $ra needs to be saved before any other function call. However, if a function knows that it is not going to call another function, it can skip storing $ra.

Run Time Environment (RTE) Storage ...

11

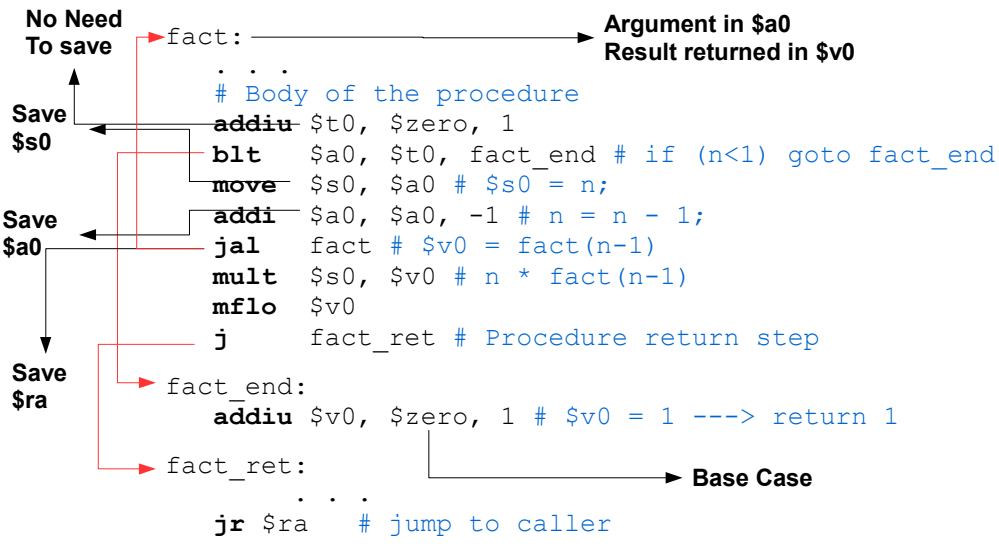


- Caller can use the stack to pass more than four arguments (e.g. printf many time needs to have more than four arguments). It pushes the arguments into stack in sequence. In some cases, function may needs caller to put all the arguments into the stack instead of $v0-$v3. In this scenario, function known exactly how to access which argument value. Once the function is done, it is caller's responsibility to pop out those arguments.

- The total amount of space needed for a function to store RTE and local variables is known by programmer. Programmer knows how many registers to save (at least the $fp and in most cases $ra) and how much local variable / storage it needs in the memory. If the amount is x byte then the $sp is moved by x + 8 byte to point to a free space in stack which can be used to push other data by other functions. First the $sp is moved down by (x+8) byte. Then all the caller RTE stored w.r.t. $sp. At last the $fp is set to the old value of $sp. The $sp is saved implicitly – at the end of the function, previous $sp value is restored by ($sp – x – 8). The new $sp points to a double word aligned space (that +8 in the equation) to accommodated double word push if needed.

- The function can use $fp as the base address to access arguments stored in stack.

RTE Storage – real example

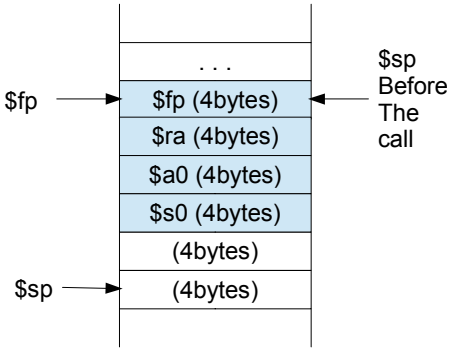


13

- The register $fp always needs to be stored. It is always a good idea to store $ra even if the function does not call another function from inside.

- Once the function is written, programmer needs to examine if it used $a0-$a3 and $s0-$s7 to store temporary result. If any one of them is used to store temporary results, it must be stored as a part for caller RTE.

RTE Storage – real example

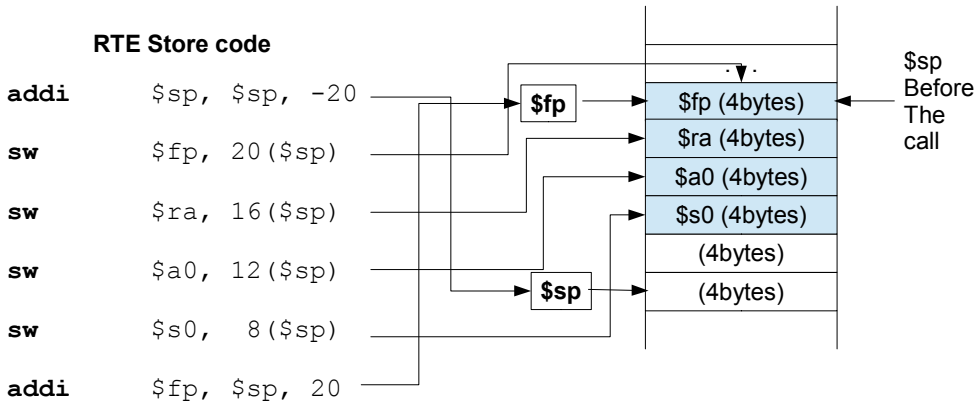


- Need to store $s0, $a0, $ra, $fp in stack frame (total 16-byte).
- $sp needs to point to free space with double word (8-byte) push capability.
- Therefore, first $sp is moved down by 20 bytes.
- Then we store $s0, $a0, $ra and current $fp. At last, we move the $fp to location that $sp was pointing previously.

14

- By inspection of our implementation of factorial function it is revealed that we need to store $s0, $a0, $fp and $ra. This is total of 16-byte. Therefore the frame size is 16 bytes. The $sp needs to point to a space double word push capable. It is 8 byte. This means the function needs to keep aside total (16+8) 24 bytes of space and the $sp needs to point to the lowest address position of this space. Since the $sp, when the call is made, is already pointing to a 4-byte free space, we need to move the $sp by 20 bytes only (not 24 bytes).

RTE Storage – real example

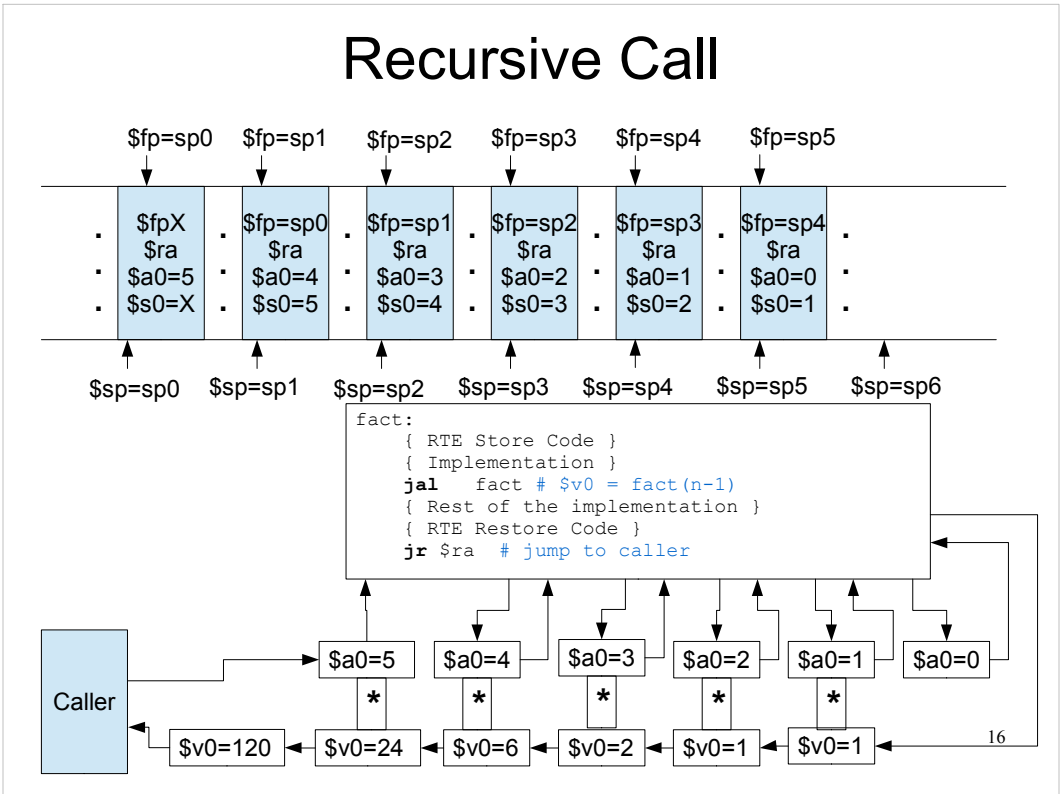


15

- Once $sp is moved to new position $fp, $ra, $a0 and $s0 are copied into stack memory using 'sw' instruction relative to $sp. After the store is done, $fp is set to the old position of the $sp (i.e. $sp+20 in this case).

- This RTE storing code to be executed at the entry point of any function implementation. On exit, it needs to restore caller RTE from this storage part. For our example the code looks like the following.

```
lw      $fp, 20($sp)
lw      $ra, 16($sp)
lw      $a0, 12($sp)
lw      $s0,  8($sp)
addi    $sp, $sp, 20
```



- The stack frame is created for each call for the function, even if it calls itself (for recursive function implementation). Please note the frame is not 'wiped' out from the stack memory upon exiting. It is only freed up for override later by moving the stack pointer to the previous position.

- In this example, caller calls the factorial function with argument value 5 at $a0. Upon entering the function, it sets up the stack frame and move the stack pointer down. Since 5 is not the base condition, it calls it self, passing the argument value as 4 (one less to 5) on $a0. Upon entering for the second call, it again sets up the next frame. This process continues till it reaches base condition (i.e. $a0 has value 0). At this point it returns the caller a value 1 at $v0. Before returning, it restore the caller (in this case it's own earlier call) RTE. This return value is used to compute factorial value at that stage and the value is returned in similar fashion. Finally when it reaches the point when it knows value of factorial 4, it computes value of factorial 5 and returns to original caller.

- Since recursive function involves setting up stack frame at each call, it has performance issue. Usually recursive functions are converted into iterative implementation to avoid deep recursion.

CS47 - Lecture 12

Kaushik Patra
(kaushik.patra@sjsu.edu)

17

- Topics
  - Procedure Call
  - Low Level Procedure Call
  - Caller Run Time Environment
  - RTE Storage

[Chapter 2.7, Appendix A.6 of Computer Organization & Design by Patterson and Hennessy.]