Execution Flow, Branches, Function Calls

1 Introduction

- This week, we come back up from the internals of the CPU and return to the investigation of the design of the Instruction Set Architecture (ISA).
- We are going to look at the instructions that control the flow of execution in the CPU, i.e. those that affect the value of the program counter (PC).
- We will see how a few primitive instructions can be used to implement high–level control constructs such as decision–making, loops and function calls.

2 Relative Branches

- In the addressing modes section, we saw the idea of relative branching:

  PC <– old PC + literal value in instruction

- And this could be combined with a comparison, so that the branch was only taken if the comparison was true. If false, the PC increments as usual and the CPU executes the following instruction as usual.
- An example MIPS instruction which performs a comparison and a branch is:

```
beq $t1, $t2, 100     # PC = PC + 100 if $t1 == $t2
```

- The immediate value specified in the branch instruction can be positive or negative.
  - Positive values allow the CPU to skip past sections of code, useful for implementing decision constructs like IF ... THEN ... ELSE.
  - Negative values allow the CPU to go back and repeat code, useful for implementing loop constructs.
- In the lab, we have already seen how to do this, so here is a quick recap.

2.1 IF ... THEN ... ELSE

- High–level code:

```
if (condition is true) {
  do this code;
} else {
  do that code;

}
```

- Low–level equivalent code:

```
if_start:    branch to else_clause if (condition is false)
             do this code
             . . .
```

```
                branch always to if_end

else_clause:    do that code
                . . .
if_end:         . . .
```

- To implement IF ... THEN only (no ELSE), just branch to *if_end* if the condition is false.

2.2 WHILE Loop

- High-level code:

```
loop initialisation;
while (loop condition is true) {
  do this code;
  update the loop condition;
}
```

- Note, as always, the loop may never start, and the loop condition must change for it to end.
- Low-level equivalent code:

```
                loop initialisation
loop_start:     branch to loop_end if (loop condition is false)
                do this code
                . . .
                update the loop condition
                branch always to loop_start
loop_end:       . . .
```

2.3 DO ... WHILE Loop

- High-level code:

```
do {
  do this code;
  update the loop condition;
} while (loop condition is true);
```

- Note that the loop always starts, and the loop condition must change for it to end.
- Low-level equivalent code:

```
loop_start:     do this code
                . . .
                update the loop condition
                branch to loop_start if (loop condition is true)
loop_end:       . . .
```

2.4 FOR Loops

- High-level code:

```
for (loop initialisation; loop condition is true; update loop condition) {
  do this code;
}
```

- Low–level code is the same as that of a WHILE loop:

```
            loop initialisation
loop_start: branch to loop_end if (loop condition is false)
            do this code
            . . .
            update the loop condition
            branch always to loop_start
loop_end:   . . .
```

2.5 SWITCH Statements

- One high–level construct that we have not yet looked at is the SWITCH statement, e.g.

```
char gender;

System.out.print("Enter your gender (M or F): ");
gender= scan.nextLine(System.in).charAt(0);

switch (gender) {
   case 'M':
   case 'm': System.out.println("You are male");
             break;
   case 'F':
   case 'f': System.out.println("You are female");
             break;
   default:  System.out.println("I don't know what you are!");
}
```

- This could always be rewritten using consecutive IF ... THEN ... ELSE constructs, e.g.

```
if (gender == 'M')
  System.out.println("You are male");
else if (gender == 'm')
  System.out.println("You are male");
else if (gender == 'F')
  System.out.println("You are female");
else if (gender == 'f')
  System.out.println("You are female");
else
  System.out.println("I don't know what you are!");
```

  and hence could be translated to machine–code as a sequence of different comparisons and branches.

- But this is quite inefficient, as it requires several tests to determine to print out the last option.
- An alternative is to use a jump table, so that each option takes the same number of instructions.
- Let's assume that `char gender` is stored as an 8–bit ASCII value; in fact, this means only 7 bits are used in the value.
- Let's also write out the machine code to do all the printing as follows:

```
M_option:   nop            # An instruction that does nothing
m_option:   print "You are male"
            branch always to switch_end

F_option:   nop
f_option:   print "You are female"
```

```
                branch always to switch_end

default:        print "I don't know what you are!"

switch_end: . . .
```

- Now consider, in the program's data section, an array of 128 words which hold instruction addresses.
- Most of these hold the address of the instruction at the `default` label.

  ```
  jump_table: .word  default, default, default, default, default, ...
  ```

  but at position 77 (ASCII 'M') in the table, the word holds the value `M_option`:

  ```
                      default, default, M_option, default, default, ...
  ```

- Similarly, at positions 109, 70 and 102 ('m', 'F' and 'f'), we have `m_option`, `F_option` and `f_option`, respectively.
- Now, before the code to print out all the message, we can have this sort of machine code:

  ```
  do the code to get the gender
  lb $t1, gender                  # Load the gender into a register
  mul $t1, $t1, 4                 # Multiply $t1 by size of word
  lw $t2, jump_table($t1)         # Get the instruction address at index position
                                  # $t1 from the jump_table array, store in $t2
  jr $t2                          # Jump to the instruction at the address in $t2
  ```

- If the user enters an 'M', this is treated as ASCII 77.
- We load the word at index position 77 from the jump_table, which holds the address of the `nop` instruction at the `M_option` label.
- The CPU then performs jr $t2, which is a *register indirect jump*: PC <– value in $t2.
- The next instruction performed is the `nop`, and any code between the `jr` and the `nop` is skipped.
- What have we done? We have traded off a large amount of data (128 jump addresses) for a number of consecutive comparisons and possible branches.
- Regardless of the character entered by the user, the jump to the matching instruction takes the same amount of time.
- Also note the "branch always" at the end of each option in the machine code.
  - Can you see that this is equivalent to the `break;` command in Java and C?
  - This explains why we have to put `break;` in: it made writing the early C compilers easier.
  - Unfortunately, we are now stuck with having to write `break;` all the time.

2.6 GOTO Considered Harmful

- Back before high–level languages had the above constructs, the main mechanism to change the flow of execution was the GOTO command. For example, in BASIC:

```
10 LET I = 1
20 PRINT I
30 LET I = I + 1
40 IF (I < 20) GOTO 20
50 PRINT "End of the loop"
```

- The GOTO was easily translated into machine–language branch or jump instructions.
- Most programs with complex control structures ended up becoming a rat's nest of GOTO statements jumping everywhere; this is often known as *spaghetti code*.
- In 1968, Edsger W. Dijkstra sent a letter to the Communications of the Association for Computing Machinery (CACM) calling for GOTO to be abolished because of its harmful effects on programming: *Go To Statement Considered Harmful*.
- This, among other things, caused language designers to create better high–level control structures.
- Today, while many high–level languages still provide a `goto` statement (e.g. C), its use is strongly discouraged. And in some languages like Java, `goto` has been removed.
    - But "structured" GOTOs, in the form of `break;` to exit loops, is often still available.

3 Performing Function Calls

- It's time to move on to calling functions and returning functions.
- In a high–level function, this is all done by defining a function with some *parameter variables*, e.g.

```
public static int myFunction(int x, char y, double z)
{ ... }
```

  and elsewhere in the program, by calling this function with some *arguments*, e.g.

```
int result= myFunction(23, 'g', usersweight);
```

- But there is more going on here than meets the eye. For example, the function will probably want to define its own variables, and these need to be kept *local*, i.e. not visible to the other functions.
- Somehow, a return value has to come out of the function.
- And, we also want to support recursion, because it is so useful.
- So before we look at the machine instructions to call and return from functions, let's look at the requirements.

3.1 Requirements to Perform Functions

- Here are the things that we need for functions to work:

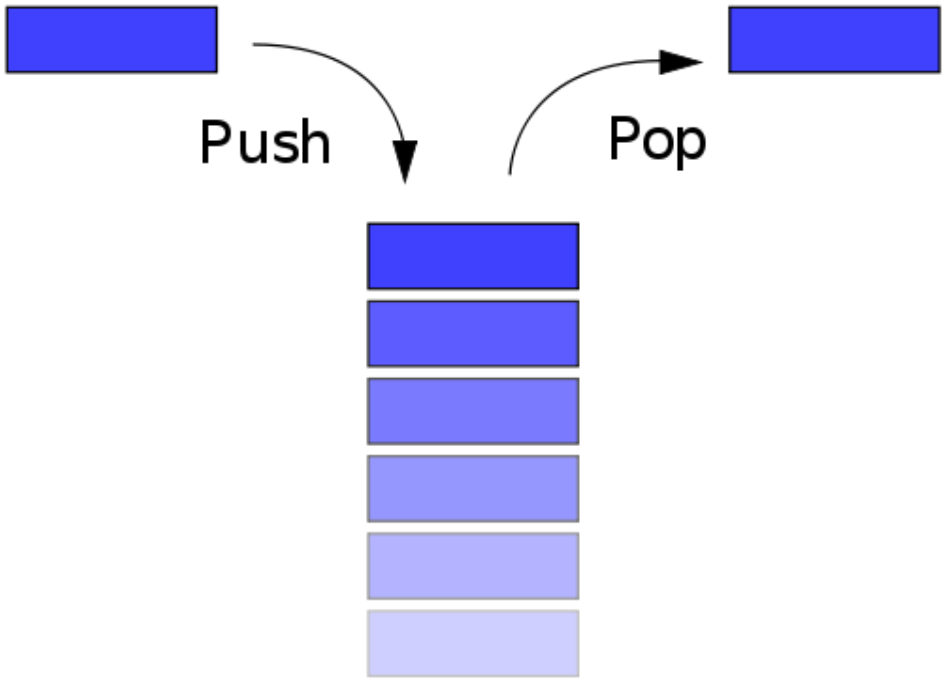
1. The function caller needs to be able to divert the program counter (PC) to the first instruction of the function.
2. At the end of the function, the function needs to be able to return the PC to the instruction *after* the function call instruction. We call this the **return address**, and it needs to be kept somewhere.
3. The function caller needs to be able to provide a set of arguments, in a specific order, to the function. The function needs to be able to see these as local variables.
4. The function needs to be able to create its own local variables which are different to those in the functions which called it.
5. Because registers are frequently used, the function caller and the function itself are going to need to agree on how to store away register values, so that each will have access to the CPU's registers without having to worry about losing values.
6. The function needs to be able to return a result to the function caller.
7. We also want recursion: the ability to write functions which call themselves an arbitrary number of times.

3.2 Issues in Performing Functions

- The big problem with implementing function calls on a CPU is that functions can call functions, which can call functions *ad infinitum*.
    - We can't predict the number of function calls.
- Why is this a problem?
    - We need to store arguments to each function when we call it,
    - We need to store the return address, so the function knows where to return to, and
    - We also need to store registers before a function call, so the function can use them, and we can get our own values back when the function returns.
- If there was only 1 function call allowed, we could reserve a fixed amount of global data space for arguments, return address and registers.
- But if functions call other functions, we need a possibly unlimited space for the arguments, return addresses and registers, and we need a way to remember which function saved which registers & return address, and which function received which arguments.

3.3 The Solution: A Stack

- We need some form of dynamically growing data structure, and a mechanism to remember who owns what in this data structure.
- The solution is a **stack**.
- In a stack, items can be pushed on the stack, and items can be popped off the stack.



(from Wikipedia)

- In the strictest sense, items pushed on the stack have to be popped off in reverse order: if we push A, B and C on the stack, then C must be popped off first before we can pop B and get access to it.
- In our implementation of the stack below, we are also able to inspect & modify values already on the stack, i.e. if A, B and C are on the stack, we can look at all three items.
- The stack provides a solution to our problem. Just before each function call, the caller can:
    - save any registers that it wants to keep by pushing them on the stack.
    - push the arguments to the function on the stack.
    - push the return address on the stack.
    - jump to the function's first instruction
- The function now has access to the arguments which are on the stack.
- At the end of the function, the function jumps to the return address which is stored on the stack.
- We are now back in the function caller, which can clean up the stack by popping everything off that it pushed on.
- And if functions call other functions, the same pushing/call/return/popping operation will occur.
- The stack keeps track of the order and depth of the function calls.

3.4 Implementing the Stack

- To implement the stack in a CPU, we need a few things:
    1. A **stack pointer**, which points at the top–most item on the stack.

2. Instructions to implement pushing and popping.
3. A large amount of RAM which can be used to hold the stack's contents.

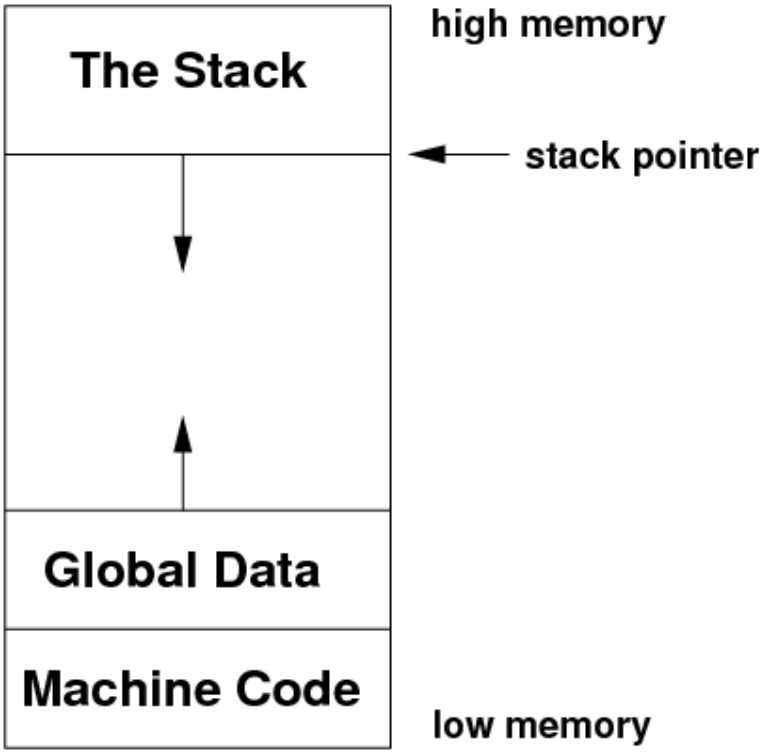
- Implementing the stack pointer is easy if our CPU provides indirect addressing: we can reserve one register to be the stack pointer, or we can reserve a specific memory address to be the stack pointer.
- On the MIPS ISA, we already know enough instructions to implement pushing and popping. For example, to push a value:

```
subu $sp, $sp, 4     # Decrement $sp by 4
sw $t1, ($sp)        # and copy $t1 onto the stack
```

- Popping a data item is the reverse:

```
lw $t1, ($sp)
addu $sp, $sp, 4
```

- On some CPUs (usually the CISC ones), there are individual PUSH and POP instructions.
- Where to place the stack is another question? It needs to go in main memory somewhere, but given that it grows, where to put it?
- One common solution is to place it at the top of memory, growing downwards (i.e. an upside–down stack):

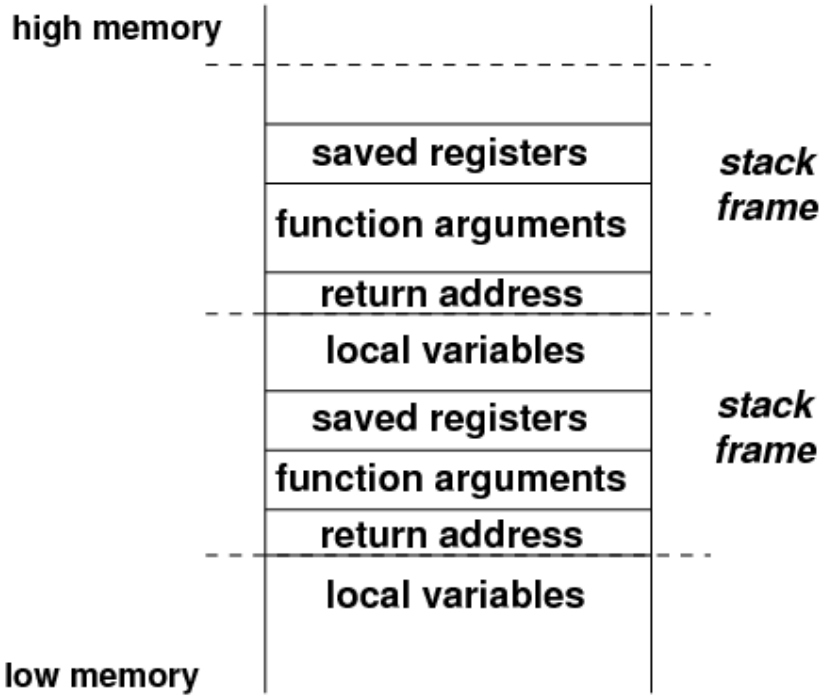


- The program's machine code is fixed in size, and can be placed at the bottom. Above that is the program's global data, and the arrangement allows the program's global data to grow as well.

3.5 The Stack Frame

- We have PUSH and POP operations on the stack, but these are for individual items.

- When dealing with functions, there are going to be a lot of items pushed onto the stack.
- We call the group of items related to a function call a **stack frame**.
- A typical set of stack frames looks like:



- I have drawn each frame with dotted lines to indicate whose responsibility it is to remove the frame from the stack, i.e. the function caller.
  - In this format, the function caller owns the arguments that it sends to the function.
  - In some texts, the frame is drawn with the arguments belonging to the function, not the function caller.
- Each frame holds:
  - any registers saved
  - the arguments to the function being called
  - the return address
- But also note that the frame has the function's own local variables.
- The stack is the natural place to store local variables. Because these are not at fixed or named locations in memory, other functions won't know where they are, which keeps them somewhat private.
- And the stack can grow to any size, so each function can create as many local variables as it needs, as long as it remembers to remove them from the stack before it returns.

3.6 Returning Values from a Function Call

- Mathematically speaking, functions return a single value.
- We don't need to be limited by this. On computers, we can return zero, one, or any number of results.

- If there are only a few return values, the simplest way to return them is to use registers.
- For large items, it makes sense to store them in memory somewhere, and return pointers to the data items (i.e. in registers).
- Beware though! Don't make the mistake of returning a pointer to a local variable. Why not?
- Local variables are kept on the stack. As a function, if you return a pointer to one of your own local variables, then there is a good chance that it will be overwritten on the next function call.
- If the function caller expects to be able to use your data, then it will be corrupted after the next function call made by the caller.

3.7 Function Calls on MIPS

- Let's now look at the instructions available on different ISAs to call functions, return from functions, deal with the stack and the stack frames.
- As you would expect, on the RISC MIPS CPU, support is minimal.
- One register is reserved as the stack pointer, $sp, and the return address for each function is stored in the $ra register.
- This immediately implies that each function has to manually store the $ra on the stack before they make a function call, otherwise their own return address (in $ra) will be destroyed by the function call.
- Two instructions for function calls exist: JAL (jump and link) and JR (jump register):

    JAL address

        $ra <– address of the instruction after the JAL, so the function can return here

        PC <– address in the JAL instruction, i.e. jump to the start of function

    JR $nn

        PC <– address stored in the $nn register

- That's it, there is no ISA support for the stack frame, so it all has to be managed manually.
- There is a convention for the format of the MIPS stack frame, and who manages which part; we will look at this in the lab.

3.8 Function Calls on the PDP–11

- The PDP–11 minicomputer ISA reserves one of the registers as the stack pointer.
- On this architecture, the stack pointer points at the next free position on the stack.

- Two instructions for function calls exist: JSR (jump to subroutine) and RTS (return from subroutine).

  JSR Rn, address

  (SP– –) <– Rn, i.e. push the return address of our function before we call the next one

  Rn <– address of the instruction after the JSR, so the function can return here

  PC <– address in the JSR instruction, i.e. jump to the start of function

  RTS Rn

  PC <– Rn, i.e. prepare PC to return to the next instruction after the JSR

  Rn <– (++SP), i.e. pop the return address of the caller back into the Rn register

- Note that very little is done for the programmer, except to save and restore the return address of the caller of each function on the stack.

3.9 Function Calls on the IA–32 Family

- The Intel IA–32 ISA provides similar instructions as the PDP–11 to jump to and return from a function; they are named CALL and RET (call procedure and return from procedure).
- There are some differences. One is that no register needs to be specified, as a stack frame is created which contains the value of the return address.
- Another difference is that RET takes an optional immediate operand.
  - If provided, this operand is added to the stack pointer. It is used to help clean up the stack frame as the function returns, if the programmer manually lowered the stack pointer themselves.
- There are two other instructions which help to manage stack frames, ENTER and LEAVE:

  ENTER size, num

  - Extend a stack frame by size bytes on the stack, point a frame pointer at the frame, and also put pointers to *num* previous stack frames in the new frame. This allows a function to have access to the local variables from the previous function(s).

  LEAVE

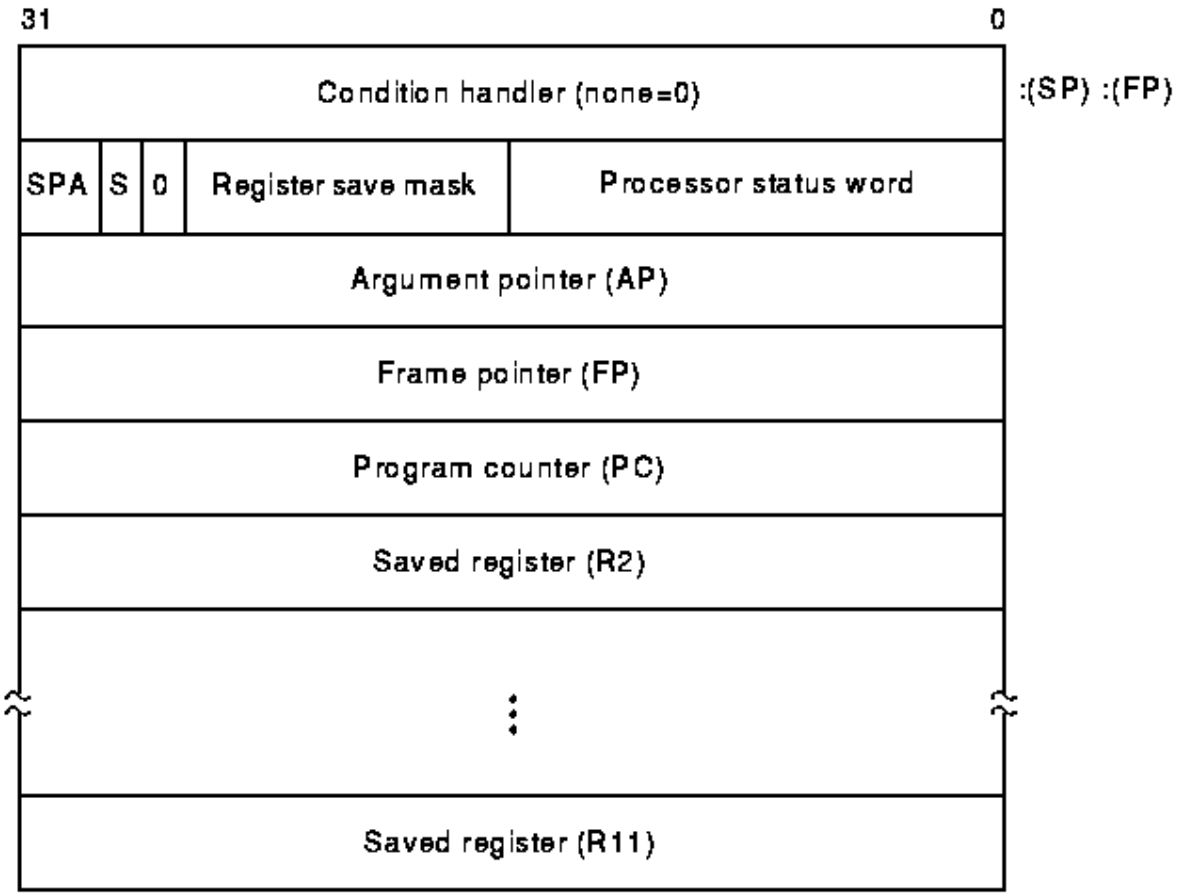
  - Undo the last stack frame extension performed by an ENTER instruction.

3.10 Function Calls on the VAX Architecture

- The VAX, being the canonical CISC architecture, provides a lot of programmer support for function calls.
- Two instructions exist which call a function:
  - CALLG: call procedure with general argument list, and
  - CALLS: call procedure with stack argument list.
- We will look at CALLS.
- Before the function call, the programmer will have placed a certain number of arguments to the function on the stack.
- Then, CALLS is involved. The format is:

  ```
  CALLS numargs, address
  ```

  where *numargs* is the number of arguments already on the stack, and *address* is the address of the function.
- The CALLS instruction:
  - pushes *numargs* on the stack
  - creates the stack frame shown below on the stack
  - saves the old values of some of the registers into the stack frame
  - saves the old status flag values (N, Z, V, C) into the stack frame
  - updates registers AP, FP, SP to point to the new frame
  - sets PC to the first instruction in the function, given by the *address*



- Thus, the work to save the caller's registers is done automatically by one instruction!
- Obviously, this instruction is a very slow one. The RISC designers would say that in most cases, only a few registers need to be saved, so if the programmer does it by hand it would be faster.
- The RET (return from procedure) instruction effectively undoes the work done by CALLS and CALLG:
    - loads the saved values of the registers from the stack frame, including AP, FP and SP.
    - loads the saved values of the CPU frames from the stack frame.
    - it even removes the arguments on the stack from before the CALLS, based on the *numargs* value stored before the stack frame!

File translated from TEX by TTH, version 3.85.
On 8 Dec 2010, 08:43.