CSE-413 Computer Architecture
Lecture 6

Supporting Procedures in Computer Hardware

Introduction

In executing a procedure, the program must follow these six steps:

- Place parameters in a place where the procedure can access them.(Set the parameters)
- Transfer control to the procedure.(Reusability)
- Acquire the storage procedure needed by the procedure.
- Perform the desired task.
- Place the result value in a place where the calling program can access them.
- Return control to the point of origin, since a procedure can be called from several points in a program.

What are Procedures?

- Also called:
  - functions
  - methods
  - subroutines
- Key Idea:
  - main routine *M* **calls** a procedure *P*
  - *P* does some work, then **returns** to *M*
    - execution in *M* picks up where left off
    - i.e., the instruction in *M* right after the one that called *P*

Why Use Procedures?

- Readability
  - divide up long program into smaller procedures
- Reusability
  - call same procedure from many parts of code
  - programmers can use each others' code
- Parameterizability
  - same function can be called with different arguments/parameters at runtime
- Polymorphism (in OOP)
  - in C++/Java, behavior can be determined at runtime as opposed to compile time

Cont.

Registers used in procedure calling:

MIPS software follows the following convention for procedure calling in allocating its 32 registers:

- $a0--$a3: Four argument registers in which to pass parameters.
- $v0--$v1: Two value registers in which to return values.
- $ra: One return address register to return to the point of origin.

Cont.

- In addition to allocating these registers, MIPS assembly language includes an instruction just for the procedures: it jumps to an address and simultaneously saves the address of the following instruction in register $ra.

- The **jump-and-link instruction (jal)** is simply written

jal ProcedureAddress

- The *link portion of the name means that an address or link is formed that points to* the calling site to allow the procedure to return to the proper address.

- This "link," stored in register $ra (register 31), is called the **return address. The return address** is needed because the same procedure could be called from several parts of the program.

Cont.

To support such situations, computers like MIPS use *jump register instruction* (jr), meaning an unconditional jump to the address specified in a register:

```
jr $ra
```

Jump register instruction jumps to the address stored in register $ra

Thus, the calling program, or **caller, puts the parameter values** in $a0-$a3 and uses jal X to jump to procedure X (sometimes named the **callee).** The callee then performs the calculations, places the results in $v0 and $v1, and returns control to the caller using jr $ra.

Cont.

Execution Sequence Using MIPS instruction:

1. The calling program puts the parameter values in $a0--$a3.
2. Use **jal X** (jump and link) to jump to the procedure. [X is the name of the called procedure]
3. Procedure X performs the calculations.
4. Place the result in $v0--$v1.
5. Return control to the calling program using **jr** (jump register) **$ra**.

Need for a Stack in Procedure Calling

- Any registers needed by the caller must be restored to the values that they contained before the procedure was invoked.
- The convention is to store the registers used by the caller into a stack and restores them from the stack when the caller need them.
- The stack pointer ($sp) is used to store the address of the most recently allocated address in the stack.

Cont.

- Stack pointer is a value denoting the most recently allocated address in a stack that shows where old register values can be found. In MIPS, it is register $sp.

- The **stack pointer is adjusted by one word for each** register that is saved or restored.

- MIPS software reserves register 29 for the stack pointer, Stacks "grow" from higher addresses to lower addresses. This convention means that you push values onto the stack by subtracting from the stack pointer.

- Adding to the stack pointer shrinks the stack, thereby popping values off the stack.

Example

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

What is the compiled MIPS assembly code?

Solution

The parameter variables g, h, i, and j correspond to the argument registers $a0, $a1, $a2, and $a3, and f corresponds to $s0. The compiled program starts with the label of the procedure:

```
leaf_example:
```

The next step is to save the registers used by the procedure. We need to save three registers: $s0, $t0, and $t1.
We "push" the old values onto the stack by creating space for three words (12 bytes) on the stack and then store them.

Cont.

```
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
```

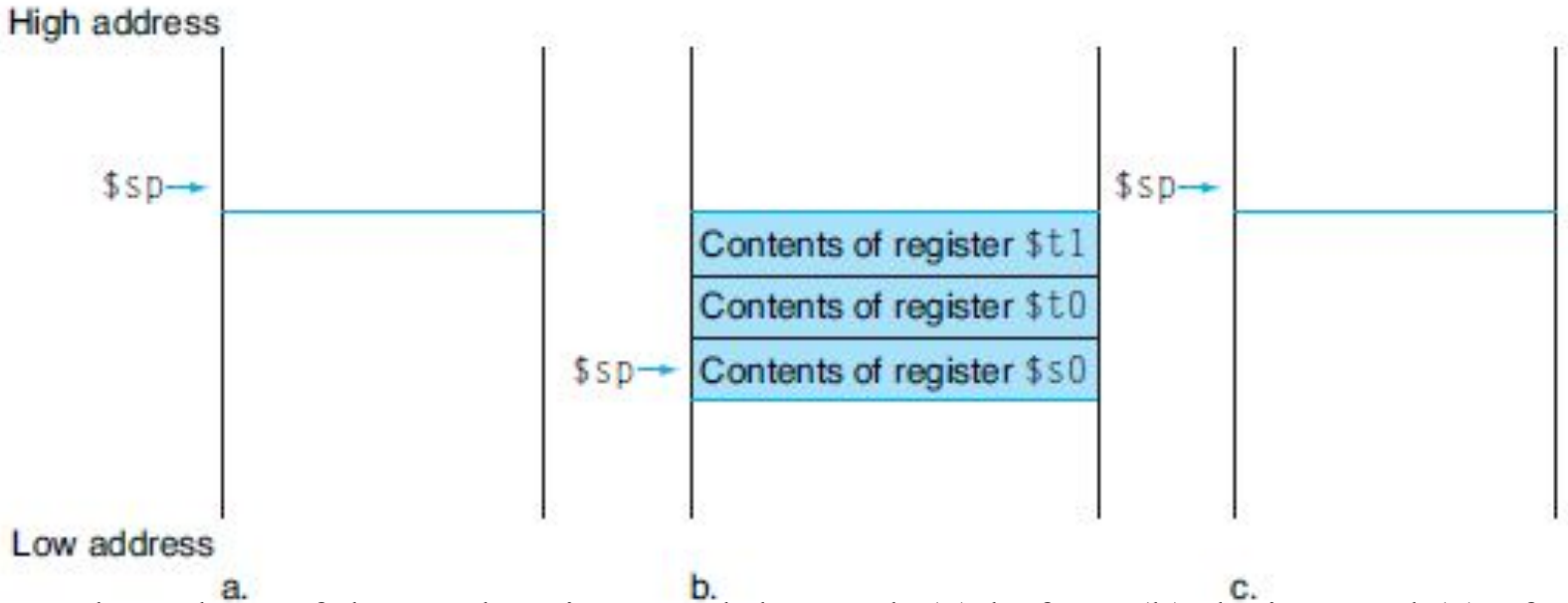

FIGURE : The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Cont.

The next three statements correspond to the body of the procedure,

```
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)
```

To return the value of f, we copy it into a return value register:

```
add $v0,$s0,$zero          # returns f ($v0 = $s0 + 0)
```

Cont.

Before returning, we restore the three old values of the registers we saved by "popping" them from the stack:

```
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
```

The procedure ends with a jump register using the return address:

```
jr $ra          # jump back to calling routine
```

Cont.

In the previous example, we used temporary registers and assumed their old values must be saved and restored. To avoid saving and restoring a register whose value is never used, which might happen with a temporary register, MIPS software separates 18 of the registers into two groups:

- $t0-$t9:ten temporary registers that are *not preserved by the callee (called* procedure) on a procedure call

- $s0-$s7:eight saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

- In the example above, since the caller does not expect registers $t0 and $t1 to be preserved across a procedure call, we can drop two stores and two loads from the code. We still must save and restore $s0, since the callee must assume that the caller needs its value.