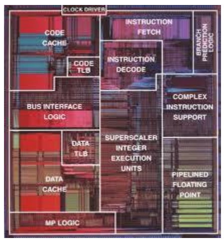
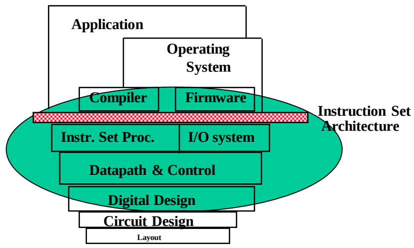
UTD

CS/SE 3340
Computer Architecture





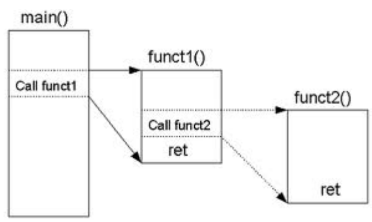
Subroutine (Procedure and Function) Call

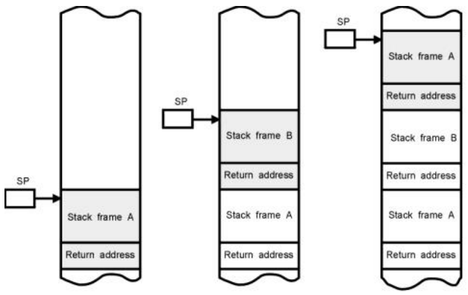
Based on slides from Prof. Aviral Shrivastava of ASU

Questions

- What are key requirements of subroutines?
- How does MIPS support subroutines?
- What are jal and jr instructions
- What is leaf/non-leaf/recursive subroutines?
- How to support non-leaf/recursive subroutines?
- What is a stack on a computer and why is it needed?
- What is registers convention?

2





a. The state of the stack during subroutine A

b. The state of the stack during subroutine B

c. The state of the stack during a second call to subroutine A

Why Subroutines?

- Subroutines (procedures, functions) allow the programmer to structure programs
  - To use the power of ***abstraction***
  - To allow for ***reuse*** – write one, use many (forever)! ☺
- Subroutines allow the programmer to concentrate on one portion of the code at a time
  - ***Divide-and-conquer*** principle
  - Parameters act as the data exchange between the subroutine and the rest of the program, allowing the subroutine to get input (input arguments) and to return results (output arguments)
- Difference between *function* and *procedure*?
  - *Function* can return a value to the caller , but *procedure* does not

Example: A Simple C Subroutine

```
main() {
   int i,j,k,m;
   ...
   i = multdiv(j,k);
   ...
   m = multdiv (i,i);
   ...
}
/* Performs a certain function */
int multdiv (int in1, int in2){
   int result;
   result = in1*2 + in2/2;

   return result;
   }
```


Requirements for Subroutines

- Can pass *arguments* (input) to the subroutine
- Can get *results* from the subroutine
- Can call the subroutine from *anywhere* in the program
- Can always *return back* to the statement just after the subroutine call
- Support *non-leaf (nested)* and *recursive* subroutines
- Support preservation of (some) registers

Keep Track of Subroutine Calls

- Registers play a major role in keeping track of information for subroutine calls
- Registers usage
  - Return address `$ra`
  - Arguments `$a0, $a1, $a2, $a3`
  - Return value `$v0, $v1`
  - Local variables `$s0, $s1, … , $s7`
- The ***stack*** is also used during subroutine calls
  - Required for *nested* and *recursive* subroutines
  - Also needed for subroutines that have more than 4 arguments

Compiling Subroutines

```
/* now need to call subroutine sum(a,b) */
    c = sum(a,b);... /* a,b:$s0,$s1 */
  }
  int sum(int x, int y) {
     return x+y;
  }
```

```
address
  1000       add  $a0,$s0,$zero  # (arg 1:$a0) x = a
  1004       add  $a1,$s1,$zero  # (arg 2:$a1) y = b
  1008       addi $ra,$zero,1016 #  ra=1016
  1012       j    sum            # jump to sum
  1016 ...
  2000 sum: add $v0,$a0,$a1      # result:$v0
  2004       jr   $ra            # new instruction
```

Requirements for Subroutines

- Pass arguments to the subroutine
  - **\$a0, \$a1, \$a2, \$a3**
- Get results from the subroutine
  - **\$v0, \$v1**
- Can call from anywhere
- Can always return back
- Support *non-leaf (nested)* and *recursive* subroutines
- Support preservation of (some) registers

Compiling Subroutines

```
/* now need to call subroutine sum(a,b) */
     c = sum(a,b);... /* a,b:$s0,$s1 */
  }
  int sum(int x, int y) {
      return x+y;
  }
```

address

```
1016        ...
2000 sum: add $v0,$a0,$a1
2004      jr   $ra    # new instruction
```

*Why use **jr** but not simple **j** here?*

*Because **sum** is called from any place and must be able to return to a location that is not fixed (variable)!*

Compiling Subroutines – cont'd

- Further optimization: single instruction to save return address and then jump!
  - jump and link (`jal`)
- Before

```
1008 addi $ra,$zero,1016   # $ra=1016
1012 j sum                 # jump to sum
1016  …
```

- After

```
1008 jal sum             # $ra=1012 (?),go to sum
```

- Why MIPS provides `jal` instruction?
  - *Make the common case run fast! (subroutine calls are very common)*
- Also, programmers do not need to know where the code is loaded into memory with `jal` (e.g. 1016 above)
  - *Can call from anywhere*
  - *Less error-prone!*

Compiling Subroutines – cont'd

- Syntax for **`jal`** (jump and link) is same as for **`j`** (jump):

```
jal   label
```

- **`jal`** should really be called **`laj`** for "link and jump"! ☺
  - Step 1 (link): save address of *next* instruction (which is the return address) into **`$ra`**
  - *why next instruction but not the current one?*
  - Step 2 (jump): jump to the address given by the label

Compiling Subroutines – cont'd

- Syntax for **`jr`** (jump register)

  `jr register`

- Instead of providing a label to jump to, the **`jr`** instruction provides a *register* which contains a memory address to jump to
- Useful if we know exact memory address to jump to only at run-time
- Very useful for function calls
  - **`jal`** stores return address in register **`($ra)`**
  - **`jr $ra`** jumps back to that address

Requirements for Subroutine

- Pass arguments to the subroutine
  - **`$a0, $a1, $a2, $a3`**
- Get results from the subroutine
  - **`$v0, $v1`**
- Can call from anywhere
  - **`jal`**
- Can always return back
  - **`jr $ra`**
- Support *non-leaf (nested)* and *recursive* subroutines
- Support preservation of (some) registers

Non-Leaf (Nested) Subroutines

- Subroutines that call other subroutines
  - Other subroutines may change content of registers: **`$ra, $a0, $a1, $v0,`** etc..
- Thus for nested subroutine call, caller needs to save on the *stack*:
  - Its *return address*
  - Any *arguments* and *temporaries* needed after the call
- And restore from the *stack* after the call

15

Leaf Subroutines Example

- C code:

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

  - Arguments **`g, …, j`** in **`$a0, …, $a3`**
  - Result in **`$v0`**

16

Recursive Subroutines Example

- C code:

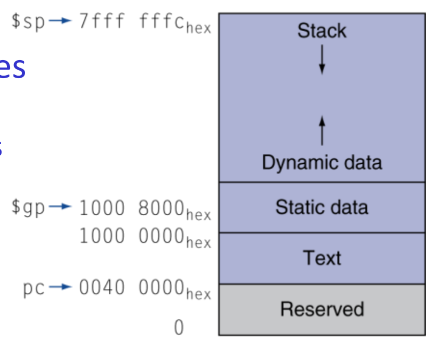
```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

  – Argument **n** in **$a0**
  – Result in **$v0**

17

Memory Layout

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



18

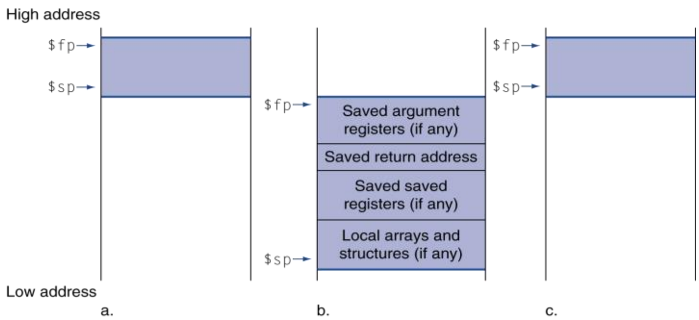
Using the Stack

- Register **`$sp`** always points to the last used space in the stack
- 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)
{
 int z;          /* local vars */
  z = mult(x,x);
  return z + y;
}
```

19

Local Data on the Stack



- Local data allocated by *the callee*
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage
  - Before (.a), during (b.) and after (c.) a subroutine call

20

Requirements for Subroutines

- Pass arguments to the subroutine
  - `$a0, $a1, $a2, $a3`
- Get results from the subroutine
  - `$v0, $v1`
- Can call from anywhere
  - `jal`
- Can always return back
  - `jr $ra`
- *Nested* and *recursive* subroutine
  - Save `$ra` and other necessary information on the stack
- Support preservation of (some) registers

21

Register Convention

- CalleR: the calling subroutine
- CalleE: the subroutine being called
- When *callee* returns, the *caller* needs to know which registers may have changed and which are guaranteed to be unchanged
- Register convention: a set of generally accepted rules as to which registers will be unchanged after a procedure call (`jal`) and which may be changed

22

Register Preservation Choices

- No register is guaranteed
  - Caller will be saving lots of registers that callee doesn't use!
  - *Inefficient!*
- All registers are guaranteed
  - Callee will be saving lots of registers that caller doesn't use!
  - *Inefficient!*
- Register convention: a balance between these two extremes

23

Register Convention – Saved Registers

- **`$0:`** No change, always contains 0!
- **`$s0-$s7:`** need to be restored if changed! (that's why they're called saved registers

  If the callee changes these in any way, it must restore the original values before returning

  Callee *saves!*
- **`$sp:`** *must be* restored if changed!

  The stack pointer must point to the same place before and after the **`jal`** call, otherwise the caller won't be able to restore values from the stack

24

Register Convention – Volatile Registers

- `$ra:` can change
  The `jal` instruction calls itself will change this register
  Caller needs to save this register on stack if nested or recursive call
- `$v0-$v1:` **can change**
  These contain the new returned values
- `$a0-$a3:` **can change**
  These are volatile argument registers
  Caller needs to save if they'll need them after the call
- `$t0-$t9:` **can change**
  That's why they're called temporary: any procedure may change them at any time
  Caller needs to save if they'll need them afterwards

25

MIPS Registers Convention Summary

Name	Register Number	Usage	Should preserve on call?
$zero	0	the constant 0	N/A
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	no
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

26

Requirements for Subroutines

- Pass arguments to the subroutine
  - `$a0, $a1, $a2, $a3`
- Get results from the subroutine
  - `$v0, $v1`
- Can call from anywhere
  - `jal`
- Can always return back
  - `jr $ra`
- Nested and recursive subroutine
  - Save `$ra` and other necessary information on the stack
- Saving and restoring registers
  - Registers convention

Recursive Subroutine Call

- C code:

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

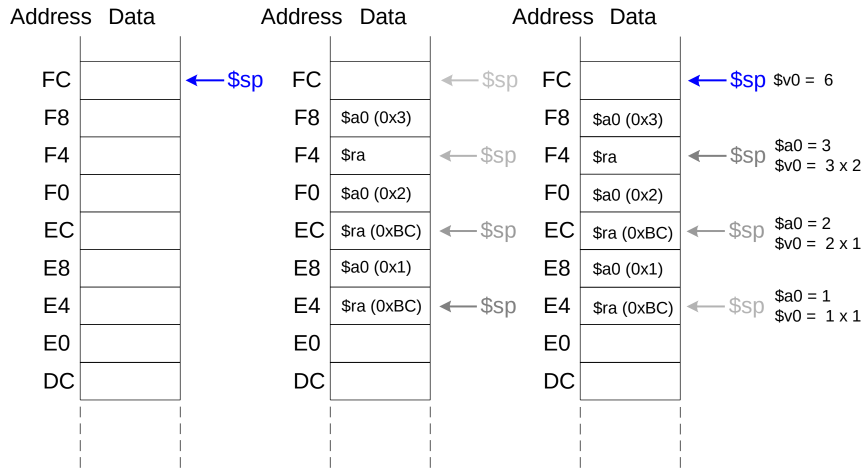
  - Argument **n** in **$a0**
  - Result in **$v0**

Recursive Subroutine Call – cont'd

- MIPS assembly code

```
0x90 factorial: addi $sp, $sp, -8   # make room
0x94            sw   $a0, 4($sp)    # store $a0
0x98            sw   $ra, 0($sp)    # store $ra
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0 # a <= 1 ?
0xA4            beq  $t0, $0, else # no: go to else
0xA8            addi $v0, $0, 1    # yes: return 1
0xAC            addi $sp, $sp, 8   # restore $sp
0xB0            jr   $ra           # return
0xB4       else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial     # recursive call
0xBC            lw   $ra, 0($sp)   # restore $ra
0xC0            lw   $a0, 4($sp)   # restore $a0
0xC4            addi $sp, $sp, 8   # restore $sp
0xC8            mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC            jr   $ra           # return
```


Stack During Recursive Calls



Subroutine Call Summary

Caller

- Put arguments in **`$a0-$a3`**
- Save any needed registers (**`$ra`**, maybe **`$t0-$t9,$a0-$a3`**)
- **`jal callee`**
- Restore registers
- Look for result in **`$v0-$v1`**

Callee

- Save registers that might be disturbed **`($s0-$s7)`**
- Perform function
- Put result in **`$v0-$v1`**
- Restore registers
- **`jr $ra`**

31