CS 61C: Great Ideas in Computer Architecture (Machine Structures)
More MIPS Machine Language

Instructors:
Randy H. Katz
David A. Patterson

http://inst.eecs.Berkeley.edu/~cs61c/sp11

Levels of Representation/Interpretation

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., MIPS)

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

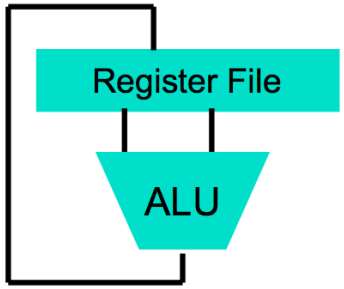
Assembler

Machine Language Program (MIPS)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

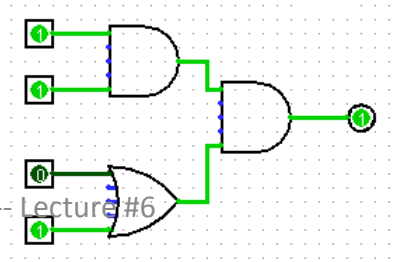
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Agenda

- Support for Strings
- **Administrivia**
- Implementing Functions In MIPS
- **Break**
- Calling Conventions
- Summary

Strings: C vs. Java

- C: each char is 8-bit ASCII. Deal with 1 byte at a time.
- Java: 16-bit UNICODE (a much larger character vocabulary). Deal with 2 bytes at a time.

Support for Characters and Strings

- Load a word, use `andi` to isolate byte

```
lw   $s0,0($s1)
andi $s0,$s0,255 # Zero everything but last 8 bits
```

- RISC Design Principle: "Make the Common Case Fast"—Many programs use text: MIPS has *load byte* instruction (*`lb`*)

```
lb $s0,0($s1)
```

- Also *store byte* instruction (*`sb`*)

Loading, Storing bytes

- What do with other 24 bits in the 32 bit register?
  - lb: sign extends to fill upper 24 bits

- Normally don't want to sign extend chars
- MIPS instruction that doesn't sign extend when loading bytes:
  - *load byte unsigned* (`lbu`)

Support for Characters and Strings

- MIPS also provides fast support for Unicode:

- *load halfword* instruction (*`lh`*)

```
lh $s0,0($s1)
```

  – There's also *load halfword unsigned* (`lhu`)

- *store halfword* instruction (*`sh`*)

```
sh $s0,0($s1)
```

MIPS Signed vs. Unsigned – Three Different Meanings!

- MIPS terms Signed/Unsigned "overloaded":
  - Do/Don't sign extend
    - `(lb, lbu, lh, lhu)`
  - Do/Don't overflow
    - `(add, addi, sub, mult, div)`
    - `(addu, addiu, subu, multu, divu)`
  - Do signed/unsigned compare
    - `(slt, slti/sltu, sltiu)`

What Do We Know Now?

- Arithmetic
  - add, addi, addu, addiu, sub, subu, and, andi, or, ori, sll, slr, slt, slti, sltiu!
- Data Transfer
  - lw, sw, lb, lbu, sb, lh, lhu, sh!
- Branches and Jumps
  - bne, beq, j

Agenda

- Support for Strings
- **Administrivia**
- Implementing Functions In MIPS
- **Break**
- Calling Conventions
- Summary

Administrivia

- HW2 Posted.
  - Remember, it's big!
- Project 1 posted by tomorrow, due 7/10.
  - No homework that week.
- Lab 4 is up.
- Please vote on times for Midterm and Final, and tell us any conflicts you have.
  - Survey linked on Piazza, emailed out.

Agenda

- Support for Strings
- **Administrivia**
- Implementing Functions In MIPS
- **Break**
- Calling Conventions
- Summary

Functions in C

```
void foo() {
    int i,j;
    i = factorial(10);
    ...
    j = factorial(25);
}

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

Functions have their own local storage

Functions can be called from multiple places.

We pass arguments to functions, and functions return values

Functions can call other functions, even themselves!

Six Fundamental Steps in Calling a Function

1. Put parameters in a place where function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put result value in a place where calling program can access it and restore any registers you used
6. Return control to point of origin, since a function can be called from several points in a program

MIPS Registers for Function Calls

- Registers way faster than memory, so use registers
- `$a0-$a3`: four *argument* registers to pass parameters
- `$v0-$v1`: two *value* registers to return values
- `$ra`: one *return address* register that saves *where a function is called from*.
- Including `$s` and `$t` registers, we now know 26 of 32 registers!

MIPS Instructions for Function Calls

- Invoke function: *jump and link* instruction (`jal`)
  - "link" means storing the location of the calling site.
  - Jumps to label and simultaneously saves the location of following instruction in register `$ra`

```
jal ProcedureAddress
```

- Return from function: *jump register* instruction (`jr`)
  - Unconditional jump to address specified in register

```
jr $ra
```

Function Call Example (1/4)

```
... sum(a,b);... /* a,b:$s0,$s1 */
```

C

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

address (shown in decimal)

MIPS

```
1000
1004
1008
1012
1016
…
2000
2004
```

In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.

Function Call Example (2/4)

```
... sum(a,b);... /* a,b:$s0,$s1 */
```

C

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

If we didn't have jal…

address (shown in decimal)

MIPS

```
1000 add  $a0,$s0,$zero   # x = a
1004 add  $a1,$s1,$zero   # y = b
1008 addi $ra,$zero,1016  # $ra=1016
1012 j    sum             # jump to sum
1016
…
2000 sum: add $v0,$a0,$a1
2004 jr   $ra          # new instruction
```

Function Call Example (3/4)

```
... sum(a,b);... /* a,b:$s0,$s1 */
```

C

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

MIPS

address (shown in decimal)

```
1000 add  $a0,$s0,$zero  # x = a
1004 add  $a1,$s1,$zero  # y = b
1008 jal  sum   # $ra = 1012 , goto sum
1012
…
2000 sum: add $v0,$a0,$a1
2004 jr   $ra        # new instruction
```

Function Call Example (4/4)

```
... sum(a,b);... /* a,b:$s0,$s1 */
```

C

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

- Question: Why use `jr` here? Why not use `j`?
- Answer: `sum` might be called by many places, so we can't return to a fixed place. The calling proc to `sum` must be able to say "return here" somehow.

MIPS

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

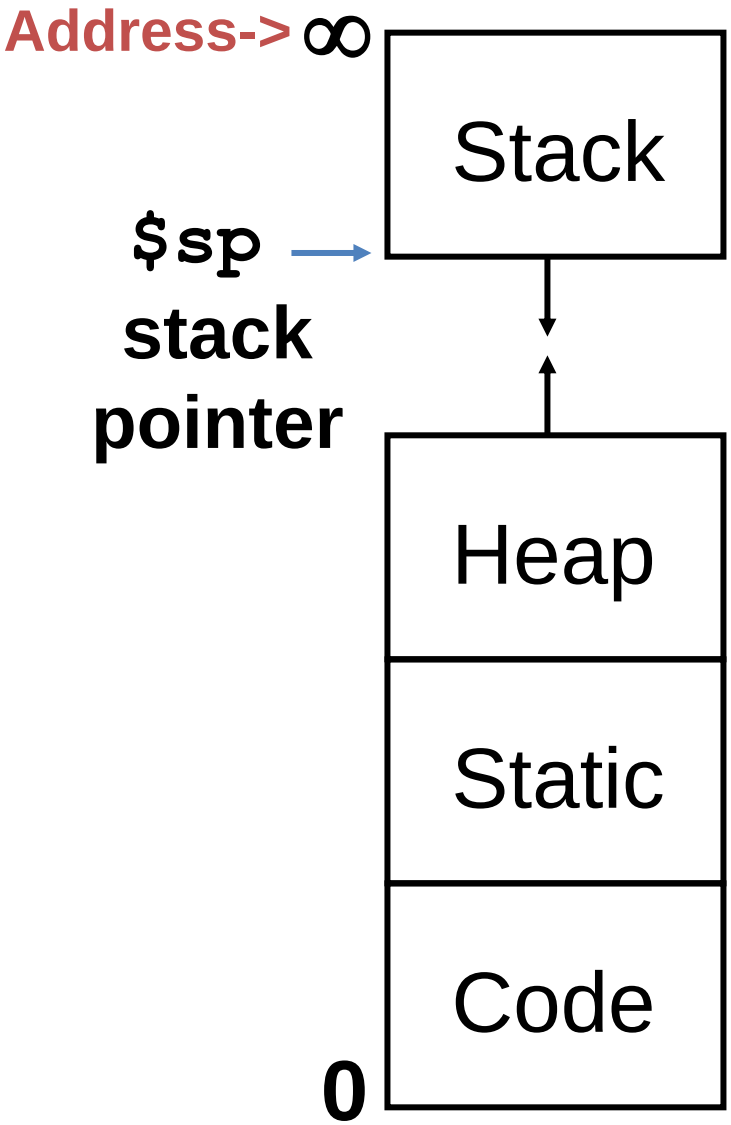
Notes on Functions

- Calling program (*caller*) puts parameters into registers `$a0-$a3` and uses `jal X` to invoke X (*callee*)
- Must have register in computer with address of currently executing instruction
  - Instead of Instruction Address Register (better name), historically called *Program Counter* (*PC*)
  - It's a program's counter, it doesn't count programs!
  - Doesn't count as one of the 32 we've mentioned.
- `jal` puts (PC+4) into `$ra` (then jumps to label)
- `jr $ra` puts address inside `$ra` into PC

What If a Function Calls another Function?

- Would overwrite `$ra`
  - Might also need to reuse other registesrs
- What is the solution?
- Save `$ra` (and anything else that might be needed later) on the stack!
- `$sp` register contains pointer to current bottom of stack.

Recall: Memory Layout



Space for saved procedure information

Explicitly created space, i.e., `malloc()`

Variables declared once per program; e.g., globals

Program

Example: sumSquare

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

- Call to mult will overwrite `$ra`, must save it.
- Need to reuse `$a1` to pass second argument to `mult`, but need current value (`y`) later. Must save `$a1`.
- To save something to stack, move `$sp` down required amount and fill the space it's created.

Example: sumSquare

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

```
sumSquare:
        addi $sp,$sp,-8       # make space on stack
"push"  sw $ra, 4($sp)        # save ret addr
        sw $a1, 0($sp)        # save y
        add $a1,$a0,$zero     # set 2nd mult arg
        jal mult              # call mult
        lw $a1, 0($sp)        # restore y
        add $v0,$v0,$a1       # ret val = mult(x,x)+y
"pop"   lw $ra, 4($sp)        # get ret addr
        addi $sp,$sp,8        # restore stack
        jr $ra
mult: ...
```

Basic Structure of a Function

Prologue

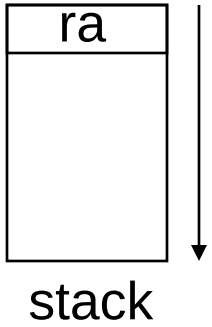
```
func_label:
addi $sp,$sp, -framesize
sw $ra, framesize-4($sp)
save other regs if need be
```

Body **(call other functions…)**

```
...
```

Epilogue

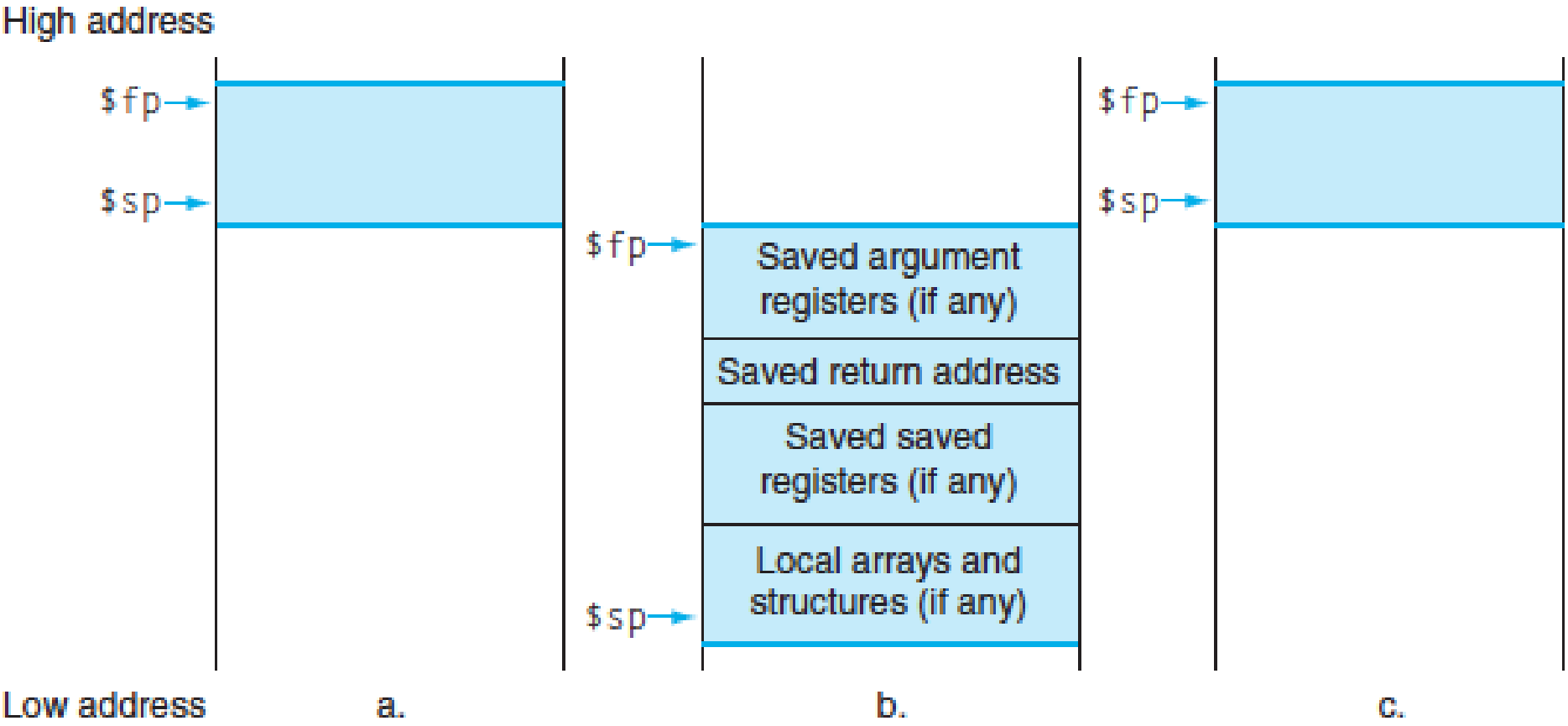
```
restore other regs if need be
lw $ra, framesize-4($sp)
addi $sp,$sp, framesize
jr $ra
```



Local Variables and Arrays

- Any local variables the compiler cannot assign to registers will be allocated as part of the stack frame.
- Locally declared arrays and structs are also allocated as part of the stack frame.
- Stack manipulation is same as before: move stack pointer down an extra amount, use the space it created as storage.

Stack Before, During, After Call



Agenda

- Support for Strings
- **Administrivia**
- Implementing Functions In MIPS
- **Break**
- Calling Conventions
- Summary

Agenda

- Support for Strings
- **Administrivia**
- Implementing Functions In MIPS
- **Break**
- Calling Conventions
- Summary

MIPS Registers

The constant 0	$0	$zero
Reserved for Assembler	$1	$at
Return Values	$2-$3	$v0-$v1
Arguments	$4-$7	$a0-$a3
Temporary	$8-$15	$t0-$t7
Saved	$16-$23	$s0-$s7
More Temporary	$24-$25	$t8-$t9
Used by Kernel	$26-27	$k0-$k1
Global Pointer	$28	$gp
Stack Pointer	$29	$sp
Frame Pointer	$30	$fp
Return Address	$31	$ra

Already discussed Not yet

(From MIPS green sheet)
Use <u>names</u> for registers -- code is clearer!

Register Conventions (1/4)

- CalleR: the calling function
- CalleE: the function being called
- When callee returns from executing, 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 (2/4) – saved

- **`$0`**: No Change. Always `0`.
- **`$s0-$s7`**: Restore if you change. Very important, that's why they're called saved registers. If the callee changes these in any way, it must restore the original values before returning.
- **`$sp`**: Restore if you 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.
- **HINT -- All saved registers start with S!**

Register Conventions (3/4) – volatile

- **`$ra`**: Can Change. The **`jal`** call itself will change this register. Caller needs to save on stack if nested call.

- **`$v0-$v1`**: Can Change. These will contain the new returned values.

- **`$a0-$a3`**: Can change. These are volatile argument registers. Caller needs to save if they are needed 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.

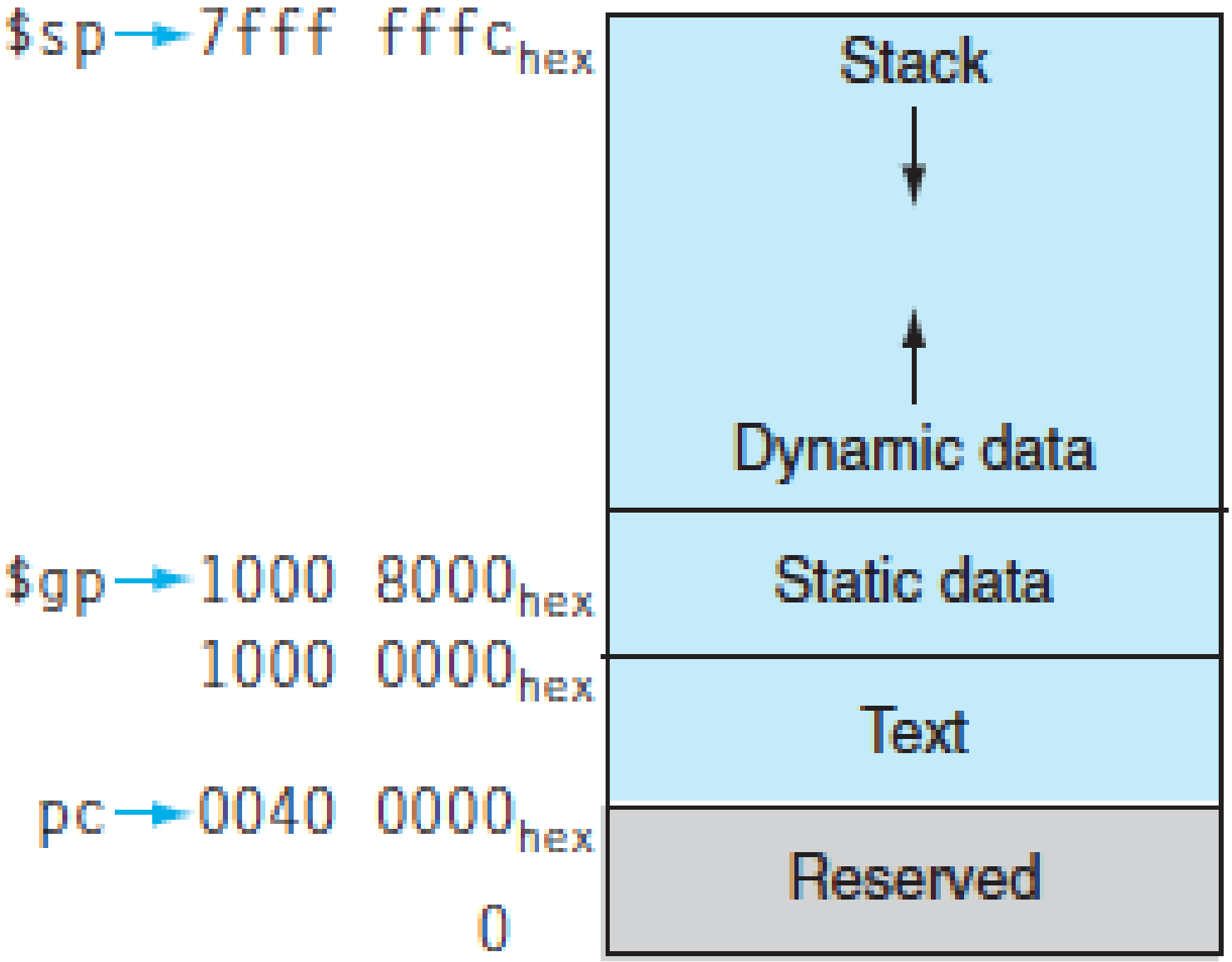
Register Conventions (4/4)

- What do these conventions mean?
  - If function R calls function E, then function R must save any temporary registers that it may be using onto the stack before making a **`jal`** call.
  - Function E must save any S (saved) registers it intends to use before garbling up their values
- Remember: caller/callee need to save only temporary/saved registers they are using, not all registers.

Where is the Stack in Memory?

- MIPS convention
- Stack starts in high memory and grows down
  - Hexadecimal : 7fff $fffc_{hex}$
- MIPS programs (*text segment*) in low end
  - 0040 0000_{hex}
- *static data segment (*constants and other static variables) above text for static variables
  - MIPS convention *global pointer* (`$gp`) points to static
- Heap above static for data structures that grow and shrink ; grows up to high addresses

MIPS Memory Allocation



And in Conclusion, …

- MIPS instructions and conventions for implementing functions.
  - Jump and link (`jal`) invokes, jump register (`jr $ra`) returns
  - Registers `$a0-$a3` for arguments, `$v0-$v1` for return values
  - Register conventions determine responsibilities of Caller and Callee for preserving values of registers.
- Stack for spilling registers, saving return address, local variables