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

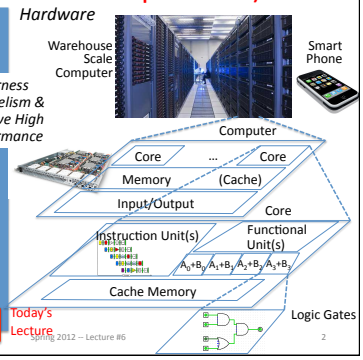
Instructor:
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/sp12

New-School Machine Structures (It's a bit more complicated!)

Software | *Hardware*

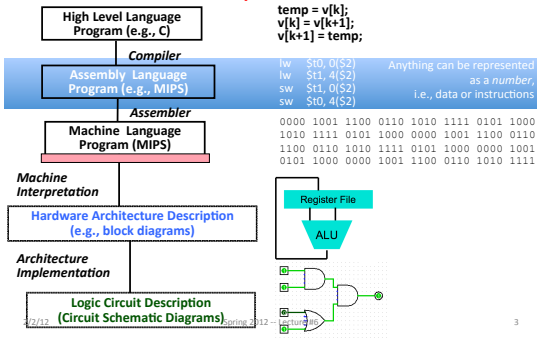
- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages

Harness Parallelism & Achieve High Performance



Today's Lecture

Big Idea #1: Levels of Representation/ Interpretation

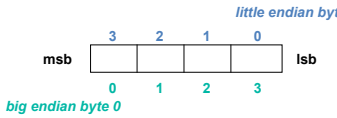


Review

- Computer words and vocabulary are called *instructions* and *instruction set* respectively
- MIPS is example RISC instruction set in this class
- Rigid format: 1 operation, 2 source operands, 1 destination
  - `add`, `sub`, `mul`, `div`, `and`, `or`, `sll`, `srl`
  - `lw`, `sw` to move data to/from registers from/to memory
- Simple mappings from arithmetic expressions, array access, if-then-else in C to MIPS instructions

Endianess

- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address



- Big Endian: address of most significant byte = word address (xx00 = "Big End" of word): IBM 360/370, MIPS, Sparc
- Little Endian: address of least significant byte = word address (xx00 = "Little End" of word): Intel 80x86
- Can only tell if access same data with load/store byte and load/store word

Agenda

- String Copy Example
- Administrivia
- Functions
- Technology Break
- MIPS register allocation convention
- Memory Heap
- Summary

Fast String Copy Code in C

- Copy x[] to y[]

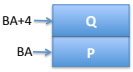
```
char *p, *q;
  p = &x[0];   /* p = x */
     /* set p to address of 1st char of x */
  q = &y[0]; /* q = y also OK */
     /* set q to address of 1st char of y */
  while((*q++ = *p++) != '\0') ;
```


Fast String Copy in MIPS Assembly

Get addresses of x and y into `$s1`, `$s2`

`p` and `q` are assigned to these registers

```
                    # $t1 = &p (BA), q @ &p + 4
                    # $s1 = p
                    # $s2 = q
Loop:               # $t2 = *p
                    # *q = $t2
                    # p = p + 1
                    # q = q + 1
                    # if *p == 0, go to Exit
      j Loop        # go to Loop
Exit: # N characters => N*6 + 3 instructions
```

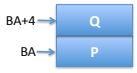


Fast String Copy in MIPS Assembly

Get addresses of x and y into `$s1`, `$s2`

`p` and `q` are assigned to these registers

```
      lw $t1, Base Address (e.g., BA)
                    # $s1 = p
                    # $s2 = q
Loop:               # $t2 = *p
                    # *q = $t2
                    # p = p + 1
                    # q = q + 1
                    # if *p == 0, go to Exit
      j Loop        # go to Loop
Exit: # N characters => N*6 + 3 instructions
```



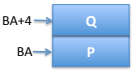
Student Roulette?

Fast String Copy in MIPS Assembly

Get addresses of x and y into `$s1`, `$s2`

`p` and `q` are assigned to these registers

```
      lw $t1, Base Address (e.g., BA)
      lw $s1,0($t1)     # $s1 = p
                        # $s2 = q
Loop:                   # $t2 = *p
                        # *q = $t2
                        # p = p + 1
                        # q = q + 1
                        # if *p == 0, go to Exit
      j Loop            # go to Loop
Exit: # N characters => N*6 + 3 instructions
```



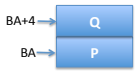
Student Roulette?

Fast String Copy in MIPS Assembly

Get addresses of x and y into `$s1`, `$s2`

`p` and `q` are assigned to these registers

```
      lw $t1, Base Address (e.g., BA)
      lw $s1,0($t1)     # $s1 = p
      lw $s2,4($t1)     # $s2 = q
Loop:                   # $t2 = *p
                        # *q = $t2
                        # p = p + 1
                        # q = q + 1
                        # if *p == 0, go to Exit
      j Loop            # go to Loop
Exit: # N characters => N*6 + 3 instructions
```



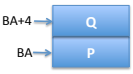
Student Roulette?

Fast String Copy in MIPS Assembly

Get addresses of x and y into `$s1`, `$s2`

`p` and `q` are assigned to these registers

```
      lw $t1, Base Address (e.g., BA)
      lw $s1,0($t1)     # $s1 = p
      lw $s2,4($t1)     # $s2 = q
Loop: lb $t2,0($s1)     # $t2 = *p
                        # *q = $t2
                        # p = p + 1
                        # q = q + 1
                        # if *p == 0, go to Exit
      j Loop            # go to Loop
Exit: # N characters => N*6 + 3 instructions
```



Student Roulette?

Fast String Copy in MIPS Assembly

Get addresses of x and y into `$s1`, `$s2`

`p` and `q` are assigned to these registers

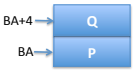
```
      lw $t1, Base Address (e.g., BA)
      lw $s1,0($t1)          # $s1 = p
      lw $s2,4($t1)          # $s2 = q
Loop: lb $t2,0($s1)          # $t2 = *p
      sb $t2,0($s2)          # *q = $t2
                             # p = p + 1
                             # q = q + 1
                             # if *p == 0, go to Exit
      j Loop                 # go to Loop
Exit: # N characters => N*6 + 3 instructions
```

Fast String Copy in MIPS Assembly

Get addresses of x and y into `$s1`, `$s2`

`p` and `q` are assigned to these registers

```
      lw $t1, Base Address (e.g., BA)
      lw $s1,0($t1)          # $s1 = p
      lw $s2,4($t1)          # $s2 = q
Loop: lb $t2,0($s1)          # $t2 = *p
      sb $t2,0($s2)          # *q = $t2
      addi $s1,$s1,1         # p = p + 1
                             # q = q + 1
                             # if *p == 0, go to Exit
      j Loop                 # go to Loop
Exit: # N characters => N*6 + 3 instructions
```



Fast String Copy in MIPS Assembly

Get addresses of x and y into `$s1`, `$s2`

`p` and `q` are assigned to these registers

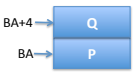
```
      lw $t1, Base Address (e.g., BA)
      lw $s1,0($t1)          # $s1 = p
      lw $s2,4($t1)          # $s2 = q
Loop: lb $t2,0($s1)          # $t2 = *p
      sb $t2,0($s2)          # *q = $t2
      addi $s1,$s1,1         # p = p + 1
      addi $s2,$s2,1         # q = q + 1
                             # if *p == 0, go to Exit
      j Loop                 # go to Loop
Exit: # N characters => N*6 + 3 instructions
```

Fast String Copy in MIPS Assembly

Get addresses of x and y into `$s1`, `$s2`

`p` and `q` are assigned to these registers

```
      lw $t1, Base Address (e.g., BA)
      lw $s1,0($t1)          # $s1 = p
      lw $s2,4($t1)          # $s2 = q
Loop: lb $t2,0($s1)          # $t2 = *p
      sb $t2,0($s2)          # *q = $t2
      addi $s1,$s1,1         # p = p + 1
      addi $s2,$s2,1         # q = q + 1
      beq $t2,$zero,Exit     # if *p == 0, go to Exit
      j Loop                 # go to Loop
Exit: # N characters => N*6 + 3 instructions
```



Which statement is TRUE?

```
char *p, *q;
  p = &x[0];  q = &y[0];
  while((*q++ = *p++) != '\0') ;
```

```
      lw $t1, Base Address
      lw $s1,0($t1)
      lw $s2,4($t1)
Loop: lb $t2,0($s1)
      sb $t2,0($s2)
      addi $s1,$s1,1
      addi $s2,$s2,1
      beq $t2,$zero,Exit
      j Loop
Exit:
```

- ☐ `$t1` corresponds to `p`
- ☐ `$s1` corresponds to `p`
- ☐ `$s1` corresponds to `q`
- ☐ `$s1` corresponds to `*p`

Administrivia

- This week in lab and homework:
  - Lab #3 EC2
    - Note: labs graded on a scale of 0 to 2
  - HW #3 Posted
    - Note: HWs graded on a scale from 0 to 3
  - Project #1 posted
- Difficulty of Project 1 part 1, time waiting for Cloud Computing in Project 1 Part 2 decided to delay HW#3 deadline a week to Sunday Feb 11

CS61c In the News

- **Apple passes HP as top PC maker when counting iPad sales**, By Mikey Campbell Monday, January 30, 2012
- **New data released on Monday by market analyst Canalys shows that Apple's iPad was instrumental to an overall growth in the client PC market for Q4 2011, as excluding the tablet's sales numbers would see the sector posting a net loss.**
- Record-setting earnings for the December quarter put Apple ahead of HP as the world's number one PC vendor, with the company's combined sales of 15.4 million iPads and 5.2 million Macs accounting for 17 percent of the total 120 million PC shipments worldwide.
- Canalys notes that desktop, netbook, notebook and tablet sales grew 16 percent from the year ago quarter, however **if iPads are not included as PCs, that number would drop to negative 0.4 percent**.
- Apple's top performance was followed by HP, Lenovo, Dell and Acer, respectively.
- Last year was said to be "The Year of the Tablet," and Apple CEO Tim Cook quipped during the company's "most people would agree, **it was the year of the iPad for the second year in a row**."

Know Your Prof: On Wrestling Team in High School & College

Why Would Wrestling Help a CS Prof? (from toast at 50 year reunion)

- Increased self-confidence
  - Physical courage => later intellectual courage
- A lesson in ethics
  - Coaching honor system that their wrestlers "made weight" (e.g., weighed ≤ 136 pounds)
  - Wrestlers imprint coach's ethics
- Learned how to form and lead teams
  - Despite largely individual sport

Assembler Pseudo-instructions

- Register $zero always contains 0
- Can use "pseudo-instruction" in assembly language to make it programming easier
- Example

```
clear $rt
```

- Implemented as:

```
add $rt, $zero, $zero
```


More Pseudo-Instructions

Name	instruction syntax	Real instruction translation
Move	move $rt,$rs	addi $rt,$rs,0
Clear	clear $rt	add $rt,$zero,$zero
Load Address	la $rd, LabelAddr	lui $rd, LabelAddr[31:16]; ori $rd,$rd, LabelAd
Load Immediate	li $rd, IMMED[31:0]	lui $rd, IMMED[31:16]; ori $rd,$rd, IMMED[15
Branch unconditionally	b Label	beq $zero,$zero,Label
Branch and link	bal $rs,Label	bgezal $zero,Label
Branch if greater than	bgt $rs,$rt,Label	slt $at,$rt,$rs; bne $at,$zero,Label
Branch if less than	blt $rs,$rt,Label	slt $at,$rs,$rt; bne $at,$zero,Label
Branch if greater than or equal	bge $rs,$rt,Label	slt $at,$rs,$rt; beq $at,$zero,Label
Branch if less than or equal	ble $rs,$rt,Label	slt $at,$rt,$rs; beq $at,$zero,Label
Branch if greater than unsigned	bgtu $rs,$rt,Label	
Branch if greater than zero	bgtz $rs,$rt,Label	
Multiplies and returns only first 32 bits	mul $d, $s, $t	mult $s, $t; mflo $d

Assembler Pseudo-instructions

- See http://en.wikipedia.org/wiki/MIPS_architecture
- Load Address (asm temp regs): $at = Label Address

```
la $at, LabelAddr
```

- Implemented as:

```
lui $at, LabelAddr[31:16];
ori $at, $at, LabelAddr[15:0]
```

Agenda

- String Copy Example
- Administrivia
- Functions
- MIPS register allocation convention
- Technology Break
- Memory Heap
- Summary

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 Function Call Conventions

- 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 to return to the point of origin
- (7 + $zero +$at of 32, 23 left!)

MIPS Registers Assembly Language Conventions

- $t0-$t9: 10 x temporaries (intermediates)
- $s0-$s7: 8 x "saved" temporaries (program variables)
- 18 registers
- 32 – (18 + 9) = 5 left

MIPS Function Call Instructions

- Invoke function: *jump and link* instruction (`jal`)
  - "link" means form an *address* or *link* that points to calling site to allow function to return to proper address
  - Jumps to address and simultaneously saves the address of following instruction in register `$ra`

```
jal ProcedureAddress
```

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

```
jr $ra
```


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!
- `jr $ra` puts address inside `$ra` into PC
- What value does `jal X` place into `$ra`? **????**

Student Roulette?

Where Save Old Registers Values to Restore Them After Function Call

- Need a place to place old values before call function, restore them when return, and delete
- Ideal is *stack*: last-in-first-out queue (e.g., stack of plates)
  - Push: placing data onto stack
  - Pop: removing data from stack
- Stack in memory, so need register to point to it
- `$sp` is the *stack pointer* in MIPS
- Convention is grow from high to low addresses
  - Push decrements `$sp`, Pop increments `$sp`
- (28 out of 32, 4 left!)

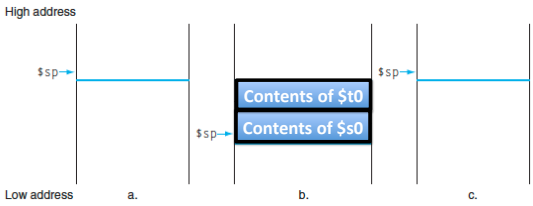
Example

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

- Parameter variables `g`, `h`, `i`, and `j` in argument registers `$a0`, `$a1`, `$a2`, and `$a3`, and `f` in `$s0`
- Assume need one temporary register `$t0`

Stack Before, During, After Function

- Need to save old values of `$s0` and `$t0`



MIPS Code for leaf_example

```
leaf_example:
                    # adjust stack for 2 int items
                    # save $t0 for use afterwards
                    # save $s0 for use afterwards
                    # f = g + h
                    # t0 = i + j
                    # return value (g + h) – (i + j)
                    # restore $s0 for caller
                    # restore $t0 for caller
                    # delete 2 items from stack
                    # jump back to calling routine
```


Student Roulette?

MIPS Code for leaf_example

```
leaf_example:
  addi $sp,$sp,-8   # adjust stack for 2 int items
                    # save $t0 for use afterwards
                    # save $s0 for use afterwards
                    # f = g + h
                    # $t0 = i + j
                    # return value (g + h) – (i + j)
                    # restore register $s0 for caller
                    # restore register $t0 for caller
                    # adjust stack to delete 2 items
                    # jump back to calling routine
```


Student Roulette?

MIPS Code for leaf_example

```
leaf_example:
  addi $sp,$sp,-8   # adjust stack for 2 int items
  sw $t0, 4($sp)    # save $t0 for use afterwards
                    # save $s0 for use afterwards
                    # f = g + h
                    # $t0 = i + j
                    # return value (g + h) – (i + j)
                    # restore register $s0 for caller
                    # restore register $t0 for caller
                    # adjust stack to delete 2 items
                    # jump back to calling routine
```


Student Roulette?

MIPS Code for leaf_example

leaf_example:

```
  addi $sp,$sp,-8   # adjust stack for 2 int items
  sw $t0, 4($sp)    # save $t0 for use afterwards
  sw $s0, 0($sp)    # save $s0 for use afterwards
                    # f = g + h
                    # $t0 = i + j
                    # return value (g + h) – (i + j)
                    # restore register $s0 for caller
                    # restore register $t0 for caller
                    # adjust stack to delete 2 items
                    # jump back to calling routine
```


MIPS Code for leaf_example

leaf_example:

```
  addi $sp,$sp,-8   # adjust stack for 2 int items
  sw $t0, 4($sp)    # save $t0 for use afterwards
  sw $s0, 0($sp)    # save $s0 for use afterwards
  add $s0,$a0,$a1   # f = g + h
                    # $t0 = i + j
                    # return value (g + h) – (i + j)
                    # restore register $s0 for caller
                    # restore register $t0 for caller
                    # adjust stack to delete 2 items
                    # jump back to calling routine
```


MIPS Code for leaf_example

leaf_example:

```
  addi $sp,$sp,-8   # adjust stack for 2 int items
  sw $t0, 4($sp)    # save $t0 for use afterwards
  sw $s0, 0($sp)    # save $s0 for use afterwards
  add $s0,$a0,$a1   # f = g + h
  add $t0,$a2,$a3   # $t0 = i + j
                    # return value (g + h) – (i + j)
                    # restore register $s0 for caller
                    # restore register $t0 for caller
                    # adjust stack to delete 2 items
                    # jump back to calling routine
```


MIPS Code for leaf_example

leaf_example:

```
  addi $sp,$sp,-8   # adjust stack for 2 int items
  sw $t0, 4($sp)    # save $t0 for use afterwards
  sw $s0, 0($sp)    # save $s0 for use afterwards
  add $s0,$a0,$a1   # f = g + h
  add $t0,$a2,$a3   # $t0 = i + j
  sub $v0,$s0,$t0   # return value (g + h) – (i + j)
                    # restore register $s0 for caller
                    # restore register $t0 for caller
                    # adjust stack to delete 2 items
                    # jump back to calling routine
```


MIPS Code for leaf_example

leaf_example:

```
  addi $sp,$sp,-8   # adjust stack for 2 int items
  sw $t0, 4($sp)    # save $t0 for use afterwards
  sw $s0, 0($sp)    # save $s0 for use afterwards
  add $s0,$a0,$a1   # f = g + h
  add $t0,$a2,$a3   # t0 = i + j
  sub $v0,$s0,$t0   # return value (g + h) – (i + j)
  lw $s0, 0($sp)    # restore register $s0 for caller
                    # restore register $t0 for caller
                    # adjust stack to delete 2 items
                    # jump back to calling routine
```


MIPS Code for leaf_example

leaf_example:

```
  addi $sp,$sp,-8   # adjust stack for 2 int items
  sw $t0, 4($sp)    # save $t0 for use afterwards
  sw $s0, 0($sp)    # save $s0 for use afterwards
  add $s0,$a0,$a1   # f = g + h
  add $t0,$a2,$a3   # $t0 = i + j
  sub $v0,$s0,$t0   # return value (g + h) – (i + j)
  lw $s0, 0($sp)    # restore register $s0 for caller
  lw $t0, 4($sp)    # restore register $t0 for caller
                    # adjust stack to delete 2 items
                    # jump back to calling routine
```

MIPS Code for leaf_example

leaf_example:

```
  addi $sp,$sp,-8   # adjust stack for 2 int items
  sw $t0, 4($sp)    # save $t0 for use afterwards
  sw $s0, 0($sp)    # save $s0 for use afterwards
  add $s0,$a0,$a1   # f = g + h
  add $t0,$a2,$a3   # $t0 = i + j
  sub $v0,$s0,$t0   # return value (g + h) – (i + j)
  lw $s0, 0($sp)    # restore register $s0 for caller
  lw $t0, 4($sp)    # restore register $t0 for caller
  addi $sp,$sp,8    # adjust stack to delete 2 items
                    # jump back to calling routine
```


Student Roulette?

MIPS Code for leaf_example

leaf_example:

```
  addi $sp,$sp,-8   # adjust stack for 2 int items
  sw $t0, 4($sp)    # save $t0 for use afterwards
  sw $s0, 0($sp)    # save $s0 for use afterwards
  add $s0,$a0,$a1   # f = g + h
  add $t0,$a2,$a3   # $t0 = i + j
  sub $v0,$s0,$t0   # return value (g + h) – (i + j)
  lw $s0, 0($sp)    # restore register $s0 for caller
  lw $t0, 4($sp)    # restore register $t0 for caller
  addi $sp,$sp,8    # adjust stack to delete 2 items
  jr $ra            # jump back to calling routine
```


What will the printf output?

- ☐ Print -4
- ☐ Print 4
- ☐ a.out will crash
- ☐ None of the above

```
static int *p;
int leaf (int g, int h,
  int i, int j)
{
  int f; p = &f;
  f = (g + h) - (i + j);
  return f;
}
int main(void) { int x;
  x = leaf(1,2,3,4);
  x = leaf(3,4,1,2);
  …
  printf("%d\n",*p);
}
```


What If a Function Calls a Function? Recursive Function Calls?

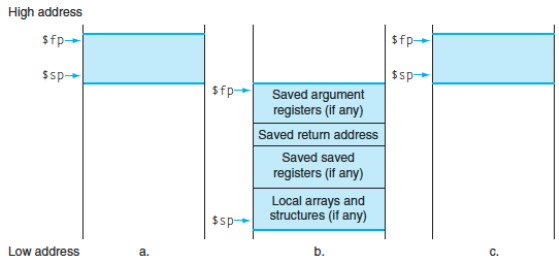
- Would clobber values in `$a0` to `$a1` and `$ra`
- What is the solution?

Student Roulette?

Allocating Space on Stack

- C has two storage classes: automatic and static
  - *Automatic* variables are local to function and discarded when function exits.
  - *Static* variables exist across exits from and entries to procedures
- Use stack for automatic (local) variables that don't fit in registers
- *Procedure frame* or *activation record*: segment of stack with saved registers and local variables
- Some MIPS compilers use a frame pointer (`$fp`) to point to first word of frame
- (29 of 32, 3 left!)

Stack Before, During, After Call



Recursive Function Factorial

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


Recursive Function Factorial

```
Fact:
  # adjust stack for 2 items
  addi $sp,$sp,-8
  # save return address
  sw $ra, 4($sp)
  # save argument n
  sw $a0, 0($sp)
  # test for n < 1
  slti $t0,$a0,1
  # if n >= 1, go to L1
  beq $t0,$zero,L1
  # Then part (n==1) return 1
  addi $v0,$zero,1
  # pop 2 items off stack
  addi $sp,$sp,8
  # return to caller
  jr $ra
```

```
L1:
  # Else part (n >= 1)
  # arg. gets (n - 1)
  addi $a0,$a0,-1
  # call fact with (n - 1)
  jal fact
  # return from jal: restore n
  lw $a0, 0($sp)
  # restore return address
  lw $ra, 4($sp)
  # adjust sp to pop 2 items
  addi $sp, $sp,8
  # return n * fact (n - 1)
  mul $v0,$a0,$v0
  # return to the caller
  jr $ra
```

mul is a pseudo instruction

Optimized Function Convention

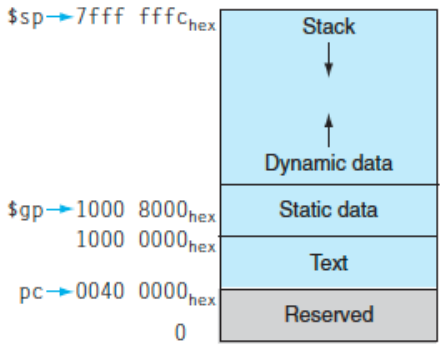
To reduce expensive loads and stores from spilling and restoring registers, MIPS divides registers into two categories:

1. Preserved across function call
   - Caller can rely on values being unchanged
   - `$ra`, `$sp`, `$gp`, `$fp`, "saved registers" `$s0- $s7`
2. Not preserved across function call
   - Caller *cannot* rely on values being unchanged
   - Return value registers `$v0`,`$v1`, Argument registers `$a0-$a3`, "temporary registers" `$t0-$t9`

Where is the Stack in Memory?

- MIPS convention
- Stack starts in high memory and grows down
  - Hexadecimal (base 16) : 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
  - (30 of 32, 2 left! – will see when talk about OS)
- *Heap* above static for data structures that grow and shrink ; grows up to high addresses

MIPS Memory Allocation



Register Allocation and Numbering

Name	Register number	Usage	Preserved on call?
`$zero`	0	The constant value 0	n.a.
`$v0–$v1`	2–3	Values for results and expression evaluation	no
`$a0–$a3`	4–7	Arguments	no
`$t0–$t7`	8–15	Temporaries	no
`$s0–$s7`	16–23	Saved	yes
`$t8–$t9`	24–25	More temporaries	no
`$gp`	28	Global pointer	yes
`$sp`	29	Stack pointer	yes
`$fp`	30	Frame pointer	yes
`$ra`	31	Return address	yes

Which statement is FALSE?

- MIPS uses `jal` to invoke a function and `jr` to return from a function
- `jal` saves PC+1 in `%ra`
- The callee can use temporary registers (`%t`*i*) without saving and restoring them
- **The caller can rely on save registers (`%s`*i*) without fear of callee changing them**

56

And in Conclusion, …

- C is function oriented; code reuse via functions
  - Jump and link (`jal`) invokes, jump register (`jr $ra`) returns
  - Registers `$a0`–`$a3` for arguments, `$v0`–`$v1` for return values
- Stack for spilling registers, nested function calls, C local (automatic) variables