UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)

RISC-V Procedures

Your Turn. What is in x12?

```
# assume x10 holds 0x34FF
slli      x12,x10,0x10
srli      x12,x12,0x08
and  x12,x12,x10
```

0x0
0x3400
0x4F0
0xFF00
0x34FF

Berkeley
UNIVERSITY OF CALIFORNIA

Garcia, Kao

C Functions

```
main() {
   int i,j,k,m;
   ...
   i = mult(j,k); ...
   m = mult(i,i); ...
}

/* really dumb mult function */

int mult (int mcand, int mlier){
   int product = 0;
   while (mlier > 0)  {
     product = product + mcand;
     mlier = mlier - 1; }
   return product;
}
```

What information must compiler/programmer keep track of?

What instructions can accomplish this?

Garcia, Kao

Six Fundamental Steps in Calling a Function

1. Put arguments 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 return value in a place where calling code can access it and restore any registers you used; release local storage
6. Return control to point of origin, since a function can be called from several points in a program

Garcia, Kao

RISC-V Function Call Conventions

- Registers faster than memory, so use them
- **`a0–a7 (x10-x17)`:** eight *argument* registers to pass parameters and two return values (**`a0-a1`**)
- **`ra`**: one *return address* register to return to the point of origin (**`x1`**)
- Also **`s0-s1`** (**`x8-x9`**) and **`s2-s11`** (**`x18-x27`**): saved registers (more about those later)

Instruction Support for Functions (1/4)

C

```
... sum(a,b);... /* a,b:s0,s1 */
}
        int sum(int x, int y) {
        return x+y;
}
```

RISC-V

```
address (shown in decimal)
    1000
    1004
    1008
    1012
    1016
    ...
    2000
    2004
```

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

Garcia, Kao

Instruction Support for Functions (2/4)

C

```
...          sum(a,b);... /* a,b:s0,s1 */
    }
    int sum(int x, int y) {
          return x+y;
    }
```

RISC-V

```
address (shown in decimal)
    1000 mv a0,s0              # x = a
    1004 mv a1,s1              # y = b
    1008 addi ra,zero,1016     #ra=1016
    1012 j    sum             #jump to sum
    1016 …                    # next inst.
    …
    2000 sum: add a0,a0,a1
    2004 jr  ra #new instr."jump reg"
```

Instruction Support for Functions (3/4)

C

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

RISC-V

```
...
2000 sum: add a0,a0,a1
2004 jr   ra #new instr."jump reg"
```

Instruction Support for Functions (4/4)

- Single instruction to jump and save return address: jump and link (`jal`)
- Before:

```
1008 addi ra,zero,1016  # ra=1016
1012 j sum              # goto sum
```

- After:

```
1008 jal ra,sum # ra=1012,goto sum
```

- Why have a `jal`?
  - Make the common case fast: function calls very common
  - Reduce program size
  - Don't have to know where code is in memory with `jal`!

Garcia, Kao

RISC-V Function Call Instructions

- Invoke function*: jump and link* instruction (`jal`)

  (really should be `laj` *"link and jump"*)

  - "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 the following instruction in register rd

```
jal rd, FunctionLabel
```

- Return from function: *jump register* instruction (`jr`)
  - Unconditional jump to address specified in register: `jr ra`
  - Assembler shorthand: `ret = jr ra`

Garcia, Kao

Summary of Instruction Support

Actually, only two instructions:

- **`jal rd, Label`** – jump-and-link # rd = pc+4; pc += imm
- **`jalr rd, rs, imm`** – jump-and-link register

  \# rd = pc+4; pc = R[rs1]+imm

  - As we're going to see, "ain't no free lunch", so there might not be enough bits left for the **`Label`** to go as far as we want to jump.
  - With **`jalr`**, we jump to the contents of the register **`rs`** + immediate **`imm`** (like a base pointer and offset) and set **`rd`** as in **`jal`**

`j, jr` and **`ret`** are pseudoinstructions!

- **`j: jal x0, Label`**

Garcia, Kao

Where Are Old Register Values Saved to Restore Them After Function Call?

- Need a place to save old values before calling function, restore them when return, and delete
- Ideal is *stack*: last-in-first-out (LIFO) 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 RISC-V (**`x2`**)
- Convention is grow stack down from high to low addresses
  - *Push* decrements **`sp`**, *Pop* increments **`sp`**

Stack

- Stack frame includes:
  - Return “instruction” address
  - Parameters (arguments)
  - Space for other local variables
- Stack frames contiguous blocks of memory; stack pointer tells where bottom of stack frame is
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames

RISC-V Function Call Example

Function Call Example

```
int Leaf (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 **s1**

RISC-V Code for Leaf()

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

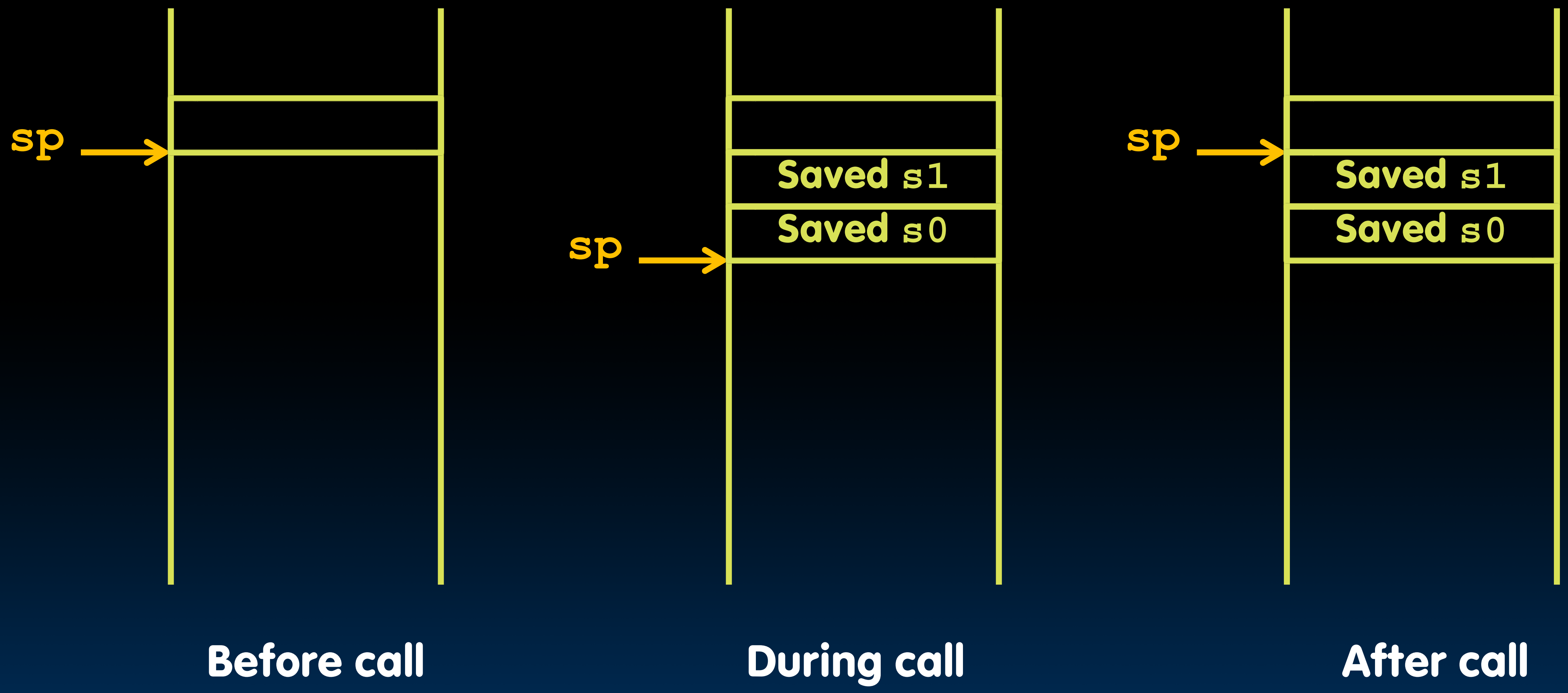
```
Leaf:  addi sp,sp,-8 # adjust stack for 2 items
       sw s1, 4(sp)  # save s1 for use afterwards
       sw s0, 0(sp)  # save s0 for use afterwards

       add s0,a0,a1 # f = g + h
       add s1,a2,a3 # s1 = i + j
       sub a0,s0,s1 # return value (g + h) - (i + j)

       lw s0, 0(sp) # restore register s0 for caller
       lw s1, 4(sp) # restore register s1 for caller
       addi sp,sp,8 # adjust stack to delete 2 items
       jr ra        # jump back to calling routine
```

Stack Before, During, After Function

- Need to save old values of **`s0`** and **`s1`**



Nested Calls and Register Conventions

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

- Would clobber values in **`a0-a7`** and **`ra`**
- What is the solution?

Garcia, Kao

Nested Procedures

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

- Something called `sumSquare`, now `sumSquare` is calling `mult`
- So there's a value in `ra` that `sumSquare` wants to jump back to, but this will be overwritten by the call to `mult`

Need to save sumSquare return address before call to `mult` – again, use stack

Berkeley UNIVERSITY OF CALIFORNIA

Garcia, Kao

Register Conventions (1/2)

- 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 (2/2)

To reduce expensive loads and stores from spilling and restoring registers, RISC-V function-calling convention divides registers into two categories:

1. Preserved across function call
   - Caller can rely on values being unchanged
   - **sp**, **gp**, **tp**,
     "saved registers" **s0- s11** (**s0** is also **fp**)
2. Not preserved across function call
   - Caller *cannot* rely on values being unchanged
   - Argument/return registers **a0-a7,ra**,
     "temporary registers" **t0-t6**

RISC-V Symbolic Register Names

Numbers hardware understands

Register	ABI Name	Description	Saver
`x0`	`zero`	Hard-wired zero	-
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	-
`x4`	`tp`	Thread pointer	-
`x5`	`t0`	Temporary/Alternate link register	Caller
`x6-7`	`t1-2`	Temporaries	Caller
`x8`	`s0/fp`	Saved register/Frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10-11`	`a0-1`	Function arguments/Return values	Caller
`x12-17`	`a2-7`	Function arguments	Caller
`x18-27`	`s2-11`	Saved registers	Callee
`x28-31`	`t3-6`	Temporaries	Caller

Human-friendly symbolic names in assembly code

Garcia, Kao

L10 Function Calls… Which one is False?

RISC-V 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 (the t registers) without saving and restoring them

The caller can rely on save registers (the s registers) without fear of callee changing them

And in Conclusion, the RV32 So Far...

(Watch 12m bonus video on Memory Allocation!)

https://drive.google.com/file/d/1MR32HeTNj1phgeR5cVRmhx-GUrHBtcEb/view?usp=sharing

- Arithmetic/logic

```
add rd, rs1, rs2
sub rd, rs1, rs2
and rd, rs1, rs2
or  rd, rs1, rs2
xor rd, rs1, rs2
sll rd, rs1, rs2
srl rd, rs1, rs2
sra rd, rs1, rs2
```

- Immediate

```
addi rd, rs1, imm
subi rd, rs1, imm
andi rd, rs1, imm
ori  rd, rs1, imm
xori rd, rs1, imm
slli rd, rs1, imm
srli rd, rs1, imm
srai rd, rs1, imm
```

- Load/store

```
lw  rd,  rs1, imm
lb  rd,  rs1, imm
lbu rd,  rs1, imm
sw  rs1, rs2, imm
sb  rs1, rs2, imm
```

- Branching/jumps

```
beq  rs1, rs2, Label
bne  rs1, rs2, Label
bge  rs1, rs2, Label
blt  rs1, rs2, Label
bgeu rs1, rs2, Label
bltu rs1, rs2, Label
jal  rd, Label
jalr rd, rs, imm
```