CS61C
Great Ideas in Computer Architecture (a.k.a. Machine Structures)

RISC-V Assembly Language

cs61c.org
RISC-V
Function Call
Example
Review: Six Basic Steps in Calling a Function

1. Put **arguments** in a place (registers) where function can access them
2. Transfer control to function (**jal**)  
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 (**ret**)
Function Call Example

```c
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 \( s1 \)
Need a place to save old values before calling function, restore them when return, and delete

Ideal is \textit{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

\textit{\textbf{sp}} is the \textit{stack pointer} in RISC-V (x2)

Convention is grow stack down from high to low addresses
  - \textit{Push} decrements \textit{sp}, \textit{Pop} increments \textit{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
Reminder: Leaf

```c
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()**

**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?
Nested Procedures

```c
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
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.
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

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

Numbers hardware understands

Human-friendly symbolic names in assembly code
Memory
Allocation
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
Stack Before, During, After Function

Before call
- \( sp \)

During call
- Saved return address (if needed)
- Saved argument registers (if any)
- Saved saved registers (if any)
- Local variables (if any)
- \( sp \)

After call
- \( sp \)
Recall - $sp$ 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?

```c
int sumSquare(int x, int y) {
    return mult(x, x) + y;
}
```
Using the Stack (2/2)

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

**sumSquare:**

```
addi sp,sp,-8          # space on stack
sw ra, 4(sp)           # save ret addr
sw a1, 0(sp)           # save y
mv a1,a0
jal mult              # mult(x,x)
lw a1, 0(sp)           # call mult
add a0,a0,a1
lw ra, 4(sp)           # restore y
lw a1, 0(sp)           # mult()+y
addi sp,sp,8          # get ret addr
jr ra                  # restore stack
```

**mult:**

...
When a C program is run, there are three important memory areas allocated:

- **Static**: Variables declared once per program, cease to exist only after execution completes - e.g., C globals
- **Heap**: Variables declared dynamically via `malloc`
- **Stack**: Space to be used by procedure during execution; this is where we can save register values
Where is the Stack in Memory?

- RV32 convention (RV64/RV128 have different memory layouts)
- Stack starts in high memory and grows down
  - Hexadecimal: \texttt{bfff\_fff0}_{\text{hex}}
  - Stack must be aligned on 16-byte boundary
    (not true in previous examples)
- RV32 programs (text segment) in low end
  - \texttt{0001\_0000}_{\text{hex}}
- static data segment (constants and other static variables) above text for static variables
  - RISC-V convention \textit{global pointer (gp)} points to static
  - RV32 \texttt{gp} = \texttt{1000\_0000}_{\text{hex}}
- Heap above static for data structures that grow and shrink; grows up to high addresses
RV32 Memory Allocation

Sp = bffff fff0_{hex}

pc = 0001 0000_{hex}

Stack

Dynamic data

Static data

Text

Reserved
“And In Conclusion...”
RV32 So Far...

- **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
Great Idea #1: Abstraction
(Levels of Representation/Interpretation)

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

Compiler

Assembly Language Program (e.g., RISC-V)

Assembler

Machine Language Program (RISC-V)

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

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

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

1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100

Out = AB + CD

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