“Application Binary Interface” (ABI) defines conventions for calling functions

- ABI names (e.g., ra, sp, fp, a0, etc.) specify conventional use of registers
- Conventions:
  - Use of stack
  - Passing arguments/return values
  - Saving values of registers
    - (Callee promises not to overwrite some registers)
### ABI register names and conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>ABI Name</th>
<th>Description</th>
<th>Saved By Callee?</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>zero</td>
<td>Always Zero</td>
<td>N/A</td>
</tr>
<tr>
<td>x1</td>
<td>ra</td>
<td>Return Address</td>
<td>No</td>
</tr>
<tr>
<td>x2</td>
<td>sp</td>
<td>Stack Pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>x3</td>
<td>gp</td>
<td>Global Pointer</td>
<td>N/A</td>
</tr>
<tr>
<td>x4</td>
<td>tp</td>
<td>Thread Pointer</td>
<td>N/A</td>
</tr>
<tr>
<td>x5–7</td>
<td>t0–2</td>
<td>Temporary</td>
<td>No</td>
</tr>
<tr>
<td>x8</td>
<td>s0/fp</td>
<td>Saved Register/Frame Pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>x9</td>
<td>s1</td>
<td>Saved Register</td>
<td>Yes</td>
</tr>
<tr>
<td>x10–x17</td>
<td>a0–7</td>
<td>Function Arguments/Return Values</td>
<td>No</td>
</tr>
<tr>
<td>x18–27</td>
<td>s2–11</td>
<td>Saved Registers</td>
<td>Yes</td>
</tr>
<tr>
<td>x28–31</td>
<td>t3–6</td>
<td>Temporaries</td>
<td>No</td>
</tr>
</tbody>
</table>
Registers are “caller-saved” or “calleesaved”

• Caller = function performing the call
• Callee = function that is called

• Caller-saved registers
  • Callee can do whatever it wants to them!
  • Caller needs to save values if it needs them later

• Callee-saved registers
  • Callee promises to restore original value
  • Must store and restore old value before returning (if it uses them)
Conventions give us some preferences for register allocation!

• If you don’t call any functions:
  • Use only caller-saved registers if you can!
  • (In general, do this for any variables not live across function calls)

• For variables live across a bunch of function calls:
  • Use callee-saved registers if you can!

More on this later
## Basic Steps in Calling a Function

### Caller
- Save caller-saved registers (if needed)
- Put parameters in a place where function can access them
- Transfer control to function

### Callee
- Save callee-saved registers (if needed)
- Acquire (local) storage resources needed for function
- Perform desired task of the function
- Put result value in a place where calling code can access it and maybe restore any registers you used
- Return control to point of origin.
Conventions for Registers

- ra/x1: Return Address
- a0-a7/x10-x17: Function arguments
  - If more than 8 arguments, put the rest on the stack
- a0(-a1): Return values
- sp/x2: Stack pointer (bottom of stack)
- fp/x8: Frame pointer (top of stack frame)
## More Detailed Steps in Calling a Function

<table>
<thead>
<tr>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Save caller-saved registers (if needed)</td>
<td>• Save callee-saved registers (if needed)</td>
</tr>
<tr>
<td>• Put first 8 arguments in a0-a7</td>
<td>• Acquire (local) storage resources needed for function</td>
</tr>
<tr>
<td>• Put remaining arguments on stack</td>
<td>• Perform desired task of the function</td>
</tr>
<tr>
<td>• Transfer control to function, linking to ra</td>
<td>• Put result value in a0</td>
</tr>
<tr>
<td></td>
<td>• Pop callee stack frame, restoring saved registers</td>
</tr>
<tr>
<td></td>
<td>• Return control to ra</td>
</tr>
<tr>
<td></td>
<td>• Pop saved arguments, registers (restore registers)</td>
</tr>
</tbody>
</table>
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`.
- Assume we compute `f` by using `s0` and `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  # s0 = 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 call

During call

After call

sp

fp

Saved s1

Saved s0

sp

Saved s1

Saved s0
Nested function calls will clobber a0-a7, ra

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

Need to save ra, caller-saved regs before calling
Compiling nested (/recursive) functions

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

`sumSquare:`

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

```
mult: ...  # recursive function
```

```
"push"  
```

```
"pop"  
```

"pop"
More detailed stack

Before/after call

Before jump

During call
Does that mean we can’t use a0-a7?

```asm
sumSquare:
  "push"
  addi sp, sp, -8 # reserve space on stack
  sw ra, 4(sp) # save ret addr
  sw a1, 0(sp) # save y
  mv a1, a0 # Store x in a1 also
  jal ra, mult # call mult
  lw a1, 0(sp) # restore y
  add a0, a0, a1 # mult()+y
  lw ra, 4(sp) # get ret addr
  addi sp, sp, 8 # restore stack
  jr ra

mult: ...
```

"pop"
If we do the convention stuff in LLVM, reg alloc can do a lot of work for us

```llvm
define i32 @f() {
  f__entry:
    %a = call i32 @g(i32 42)
    ret i32 %a
}
```

```llvm
define i32 @f() {
  f__entry:
    a0 = bitcast i32 42 to i32
    %a = call i32 @g(i32 a0)
    a0 = bitcast i32 %a to i32
    ret i32 a0
}
```
 Wait, what does it mean to do register allocation on registers?

• Nodes corresponding to registers are “pre-colored”
  • Assign them to themselves before we start register allocation
Pre-colored nodes get handled specially during register allocation

• Don’t simplify them out—can’t give them a color anyway
• Definitely don’t try to spill them
How far can we take this?

```assembly
define i32 @f() {
  f__entry:
    %saved_s0 = bitcast i32 s0 to i32
    %saved_s1 = bitcast i32 s1 to i32
    ...
    a0 = bitcast i32 42 to i32
    %a = call i32 @g(i32 42)
    a0 = bitcast i32 %a to i32
    ret i32 a0
  s0 = bitcast i32 %saved_s0 to i32
  ...
}
```

Save callee-saved registers

- %saved_s0 = bitcast i32 s0 to i32
- %saved_s1 = bitcast i32 s1 to i32
- ...
- a0 = bitcast i32 42 to i32
- %a = call i32 @g(i32 42)
- a0 = bitcast i32 %a to i32
- ret i32 a0

Restore callee-saved registers

- s0 = bitcast i32 %saved_s0 to i32
- ...
```
%saved_s0, etc. interfere with every local temp.
• RA will try to put %saved_si in si.
• RA will avoid callee-saved temps whenever possible

```assembly
define i32 @f() {
  f__entry:
  %saved_s0 = bitcast i32 s0 to i32
  %saved_s1 = bitcast i32 s1 to i32
  ...
  a0 = bitcast i32 42 to i32
  %a = call i32 @g(i32 42)
  a0 = bitcast i32 %a to i32
  ret i32 a0
  s0 = bitcast i32 %saved_s0 to i32
  ...
}
```
Make calls interfere with caller-saved regs

define i32 @f() {
    f__entry:
    %y = ...
    %a = call i32 @g(i32 42) ; Pretend this defines all caller-saved regs
    %z = add i32 %a, %y
    ret i32 %z
}

• RA will try to avoid putting %y in a caller-saved reg
• ... and if it can’t, it’ll spill(/save) %y without us doing anything!