Lecture 27, Nov 17, 2022

Using the Stack

- A region of memory used for temporary storage of data
  - LIFO structure
- Starts at a large address offset, grows downward (i.e. to lower addresses)
- The *stack pointer* (in the `sp` register) points to the element at the top of the stack
  - Adding a word to the stack decrements the stack pointer by 4
- The stack is important for subroutine calls since we can use it to save and restore registers
  - By saving registers onto the stack, we can make sure a subroutine does not trample on the caller
- In RISC-V there are preserved registers and nonpreserved registers
  - Preserved registers `s0` to `s11` and `sp` must take on the same values before and after a subroutine call (i.e. subroutines must save these)
  - Non-preserved registers `t0` to `t6` can be changed by subroutines (i.e. subroutines are free to modify these)
    * Registers `a0` to `a7` are also non-preserved
  - In the example from the previous lecture, in order to respect the calling convention we need to push the `s` registers onto the stack, or use the `t` registers
  - Note this is only a convention and not enforced in hardware
- Example: pushing 3 registers onto the stack:

```
addi sp, sp, -12
sw s1, 8(sp)
sw s2, 4(sp)
sw s3, 0(sp)
```

- To restore the registers back:

```
lw s3, 0(sp)
sw s2, 4(sp)
lw s1, 8(sp)
addi sp, sp, 12
```

Nested Subroutines

- To call a subroutine from another subroutine, we need to save the `ra` register onto the stack
- Example:

```
int main() {
    add6(11, 22, 33, 44, 55, 66);
}

int add6(int a, int b, int c, int d, int e, int f) {
    return add3(a, b, c) + add3(d, e, f);
}

int add3(int x, int y, int z) {
    return x + y + z;
}
```

```
_start:
    # Load all the arguments into registers
    addi a0, zero, 11
    addi a1, zero, 22
    addi a2, zero, 33
    addi a3, zero, 44
    addi a4, zero, 55
    addi a5, zero, 66
    # Call subroutine
    jal add6
END:
    ebreak

add6:
    # Push the return address register onto the stack
    addi sp, sp, -4
    sw ra, 0(sp)
    # Call add3, which makes a0 = a0 + a1 + a2
    # This will overwrite ra
    jal add3
    # Save a0 temporarily
    addi t0, zero, a0
    # Load the arguments and call add3 again
    addi a0, zero, a3
    addi a1, zero, a4
    addi a2, zero, a5
    jal add3
    # Add the 2 results
    add a0, a0, t0
    # Return, but first pop ra off the stack
    lw ra, 0(sp)
    addi sp, sp, 4
    jr ra

add3:
    add a0, a0, a1
    add a0, a0, a2
    jr ra
```

- Using the stack we can push additional arguments onto it if we need more than 8 arguments
  - Freeing these arguments is the responsibility of the caller – the callee does not restore the stack pointer
- Caller save: `t0` to `t7`, `a0` to `a7`, `sp` if necessary
- Callee save: `s0` to `s11`, saved and restored before the callee returns