Procedures, Stacks, and MMIO

Reminders:

Lab 1 due Thursday, 9/19

Lab 2 will be released today

Sign up for scheduled checkoff

Recap: Procedure Storage Needs

- Basic requirements for procedure calls:
  - Input arguments
  - Return address
  - Results

  Use registers for procedures arguments, return address, and results.

- Local storage:
  - Variables that compiler can’t fit in registers
  - Space to save caller’s register values for registers that we overwrite

  Each procedure call has its own instance of local storage known as the procedure’s *activation record.*

Recap: RISC-V Calling Convention

- The calling convention specifies rules for register usage across procedures

Symbolic name	Registers	Description	Saver
a0 to a7	x10 to x17	Function arguments	Caller
a0 and a1	x10 and x11	Function return values	Caller
ra	x1	Return address	Caller
t0 to t6	x5-7, x28-31	Temporaries	Caller
s0 to s11	x8-9, x18-27	Saved registers	Callee
sp	x2	Stack pointer	Callee
gp	x3	Global pointer	---
tp	x4	Thread pointer	---
zero	x0	Hardwired zero	---

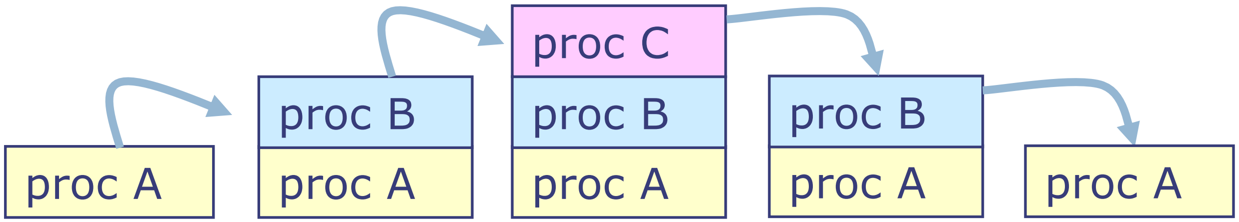
Caller-Saved vs Callee-Saved Registers

- A caller-saved register is not preserved across function calls (callee can overwrite it)
  - If caller wants to preserve its value, it must save it before transferring control to the callee
  - argument registers (aN), return address (ra), and temporary registers (tN)

- A callee-saved register is preserved across function calls
  - If callee wants to use it, it must save its value and restore it before returning control to the caller
  - Saved registers (sN), stack pointer (sp)

Recap: Activation record and procedure calls

- An *Activation record* holds all storage needs of procedure that do not fit in registers
  - A new activation record is allocated in memory when a procedure is called
  - An activation record is deallocated at the time of the procedure exit
- Activation records are allocated in a stack manner (Last-In-First-Out)



- The current procedure's activation record (a.k.a. stack frame) is always at the top of the stack

Recap: RISC-V Stack

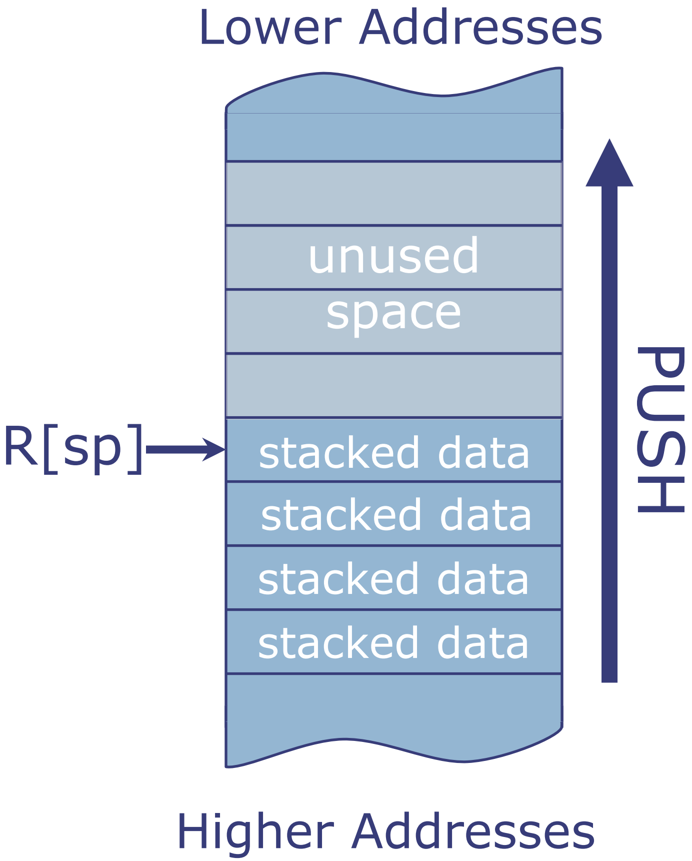
- Stack is in memory
- Stack grows down from higher to lower addresses
- sp points to top of stack (last pushed element)
- Push sequence:

```
addi sp, sp, -4
sw a1, 0(sp)
```

- Pop sequence:

```
lw a1, 0(sp)
addi sp, sp, 4
```

- Discipline: Can use stack *at any time*, but leave it as you found it!



Example: Using callee-saved registers

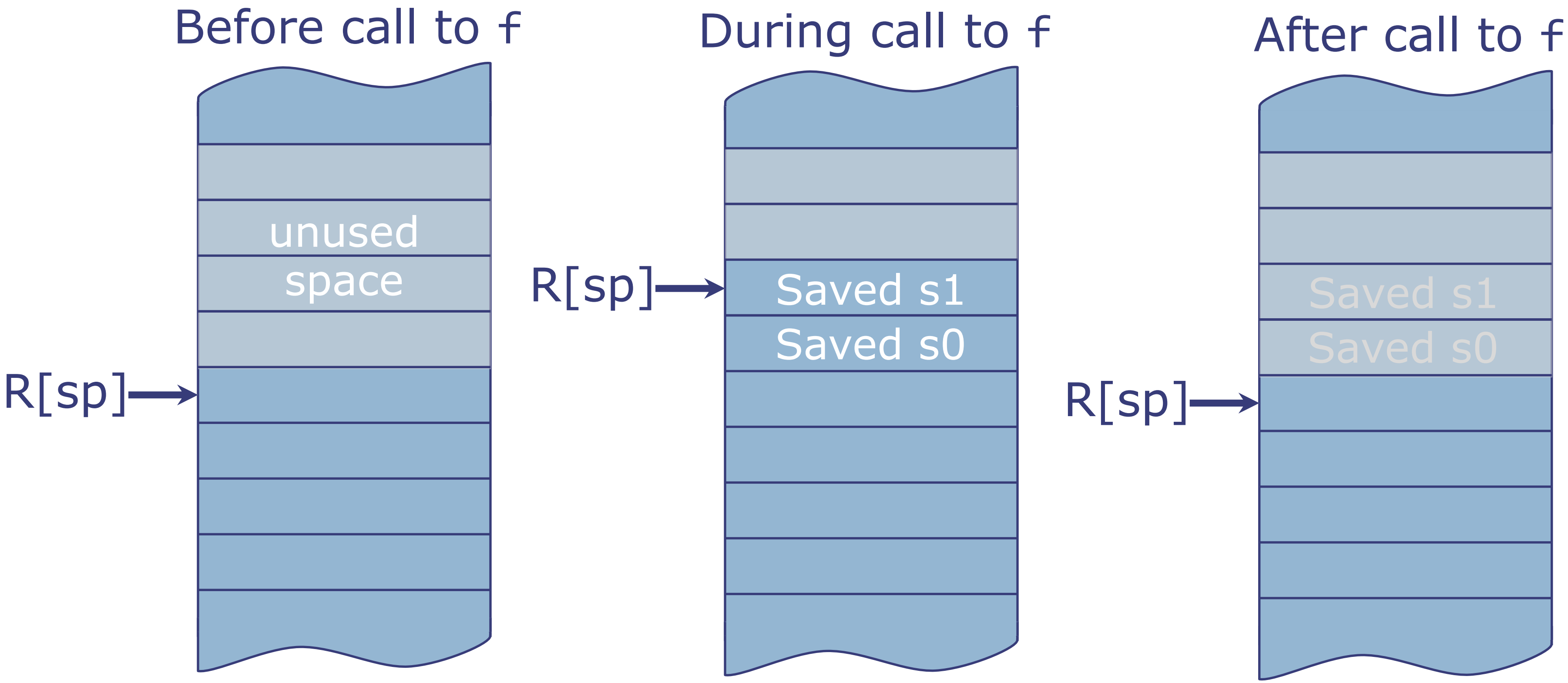
- Implement f using s0 and s1 to store temporary values

```
int f(int x, int y) {
        return (x + 3) | (y + 123456);
}
```

```
f:
  addi sp, sp, -8   // allocate 2 words (8 bytes) on stack
  sw s0, 4(sp)      // save s0
  sw s1, 0(sp)      // save s1
  addi s0, a0, 3
  li s1, 123456
  add s1, a1, s1
  or a0, s0, s1
  lw s1, 0(sp)      // restore s1
  lw s0, 4(sp)      // restore s0
  addi sp, sp, 8    // deallocate 2 words from stack
                    // (restore sp)
  ret
```

Example: Using callee-saved registers

- Stack contents:



Example: Using caller-saved registers

Caller

```
int x = 1;
int y = 2;
int z = sum(x, y);
int w = sum(z, y);
```

```
li a0, 1
li a1, 2
addi sp, sp, -8
sw ra, 0(sp)
sw a1, 4(sp) // save y
jal ra, sum
// a0 = sum(x, y) = z
lw a1, 4(sp) // restore y
jal ra, sum
// a0 = sum(z, y) = w
lw ra, 0(sp)
addi sp, sp, 8
```

Callee

```
int sum(int a, int b) {
     return a + b;
}
```

```
sum:
  add a0, a0, a1
  ret
```

Why did we save a1?

Callee may have modified a1 (caller doesn’t see implementation of sum!)

Calling Conventions Summary

Caller: Saves ra register on the stack prior to procedure call and restores it upon return. Also saves any aN or tN registers whose values need to be maintained past procedure call.

```
      addi sp, sp, -8
      sw ra, 0(sp)
      sw a1, 4(sp)
      call func
      lw ra, 0(sp)
      lw a1 4(sp)
      addi sp, sp, 8

  func:
```

Callee: Saves original value of sN registers before using them in a procedure. Must restore sN registers and stack before exiting procedure.

```
func:
   addi sp, sp, -4
   sw s0, 0(sp)
   …
   lw s0, 0(sp)
   addi sp, sp, 4
   ret
```

Nested Procedures

- If a procedure calls another procedure, it needs to save its own return address
  - Remember that `ra` is caller-saved
- Example:

```
bool coprimes(int a, int b) {
  return gcd(a, b) == 1;
}
```

```
coprimes:
  addi sp, sp, -4
  sw ra, 0(sp)
  call gcd                // overwrites ra
  addi a0, a0, -1
  sltiu a0, a0, 1
  lw ra, 0(sp)
  addi sp, sp, 4
  ret                     // needs original ra
```

Recursive Procedures

- Recursive procedures are just one particular case of nested procedures
- Example:

```
// Computes nth Fibonacci number
// Assume n >= 0
int fib(int n) {
   if (n < 2) return n;
   else return fib(n-1) + fib(n-2);
}
```

Order of these instructions
is critical for correct behavior

```
fib:
  li t0, 2
  blt a0, t0, fib_done
  addi sp, sp, -8
  sw ra, 4(sp)
  sw s0, 0(sp)
  mv s0, a0   // save n
  addi a0, a0, -1
  call fib
  mv t0, s0   // t0 = n
  mv s0, a0   // save fib(n-1)
  addi a0, t0, -2
  call fib
  add a0, s0, a0
  lw s0, 0(sp)
  lw ra, 4(sp)
  addi sp, sp, 8
fib_done:   // result in a0
  ret
```

Computing with large data structures

- Suppose we want to write a procedure vadd(a, b, c) to add two arrays a and b and store the result in array c
  - Assume the arrays are too large to be stored in registers

- We will bring the elements of a and b, one by one, into the registers and after adding them store the result back in memory

- How do we pass the arrays a and b as arguments?
  - Pass the base address and the size of each array as arguments

Passing Complex Data Structures as Arguments

```
// Finds maximum element in an
// array with size elements
int maximum(int a[], int size)
{
   int max = 0;
   for (int i = 0; i < size;
    i++) {
      if (a[i] > max) {
         max = a[i];
      }
   }
   return max;
}

int main() {
   int ages[5] =
      {23, 4, 6, 81, 16};
   int max = maximum(ages, 5);
}
```

Passing Complex Data Structures as Arguments

```
int main() {
   int ages[5] =
      {23, 4, 6, 81, 16};
   int max = maximum(ages, 5);
}
```

```
main:   li a0, ages
        li a1, 5
        call maximum
        // max returned in a0
```

```
ages:   23
        4
        6
        81
        16
```

Passing Complex Data Structures as Arguments

```c
// Finds maximum element in an
// array with size elements
int maximum(int a[], int size)
{
   int max = 0;
   for (int i = 0; i < size;
    i++) {
      if (a[i] > max) {
        max = a[i];
      }
   }
   return max;
}

int main() {
   int ages[5] =
      {23, 4, 6, 81, 16};
   int max = maximum(ages, 5);
}
```

```
maximum:
    mv t0, zero    // t0: i
    mv t1, zero    // t1: max
    j compare
loop:
    slli t2, t0, 2 // t2: i*4
    // t3: addr of a[i]
    add t3, a0, t2
    lw t4, 0(t3)   // t4: a[i]
    ble t4, t1, endif
    mv t1, t4      // max = a[i]
endif:
    addi t0, t0, 1 // i++
compare:
    blt t0, a1, loop

    mv a0, t1  // a0 = max
    ret
```

Why not always use pointers as arguments?

```
// Find perimeter of a triangle
int perimA(int a, int b,
   int c) {
   int res = a + b + c;
   return res;
}

int perimB(int sides[], int
size) {
   int res = 0;
   for (int i = 0; i < size;
    i++) {
      res = res + sides[i];
   }
   return res;
}
```

```
perimA:
     add t0, a0, zero   // t0: res
     add t0, t0, a1
     add t0, t0, a2
     mv a0, t0
     ret

perimB:
     mv t0, zero        // t0: i
     mv t1, zero        // t1: res
     j compare
loop:
     slli t2, t0, 2     // t2: i*4
     // t3: addr of sides[i]
     add t3, a0, t2
     lw t4, 0(t3)       // t4: sides[i]
     add t1, t1, t4
     addi t0, t0, 1     // i++
compare:
     blt t0, a1, loop
     mv a0, t1          // a0 = res
     ret
```

Indirection can be expensive

- Extra memory references
- Slower execution

Passing Complex Data Structures as Arguments or Return Values

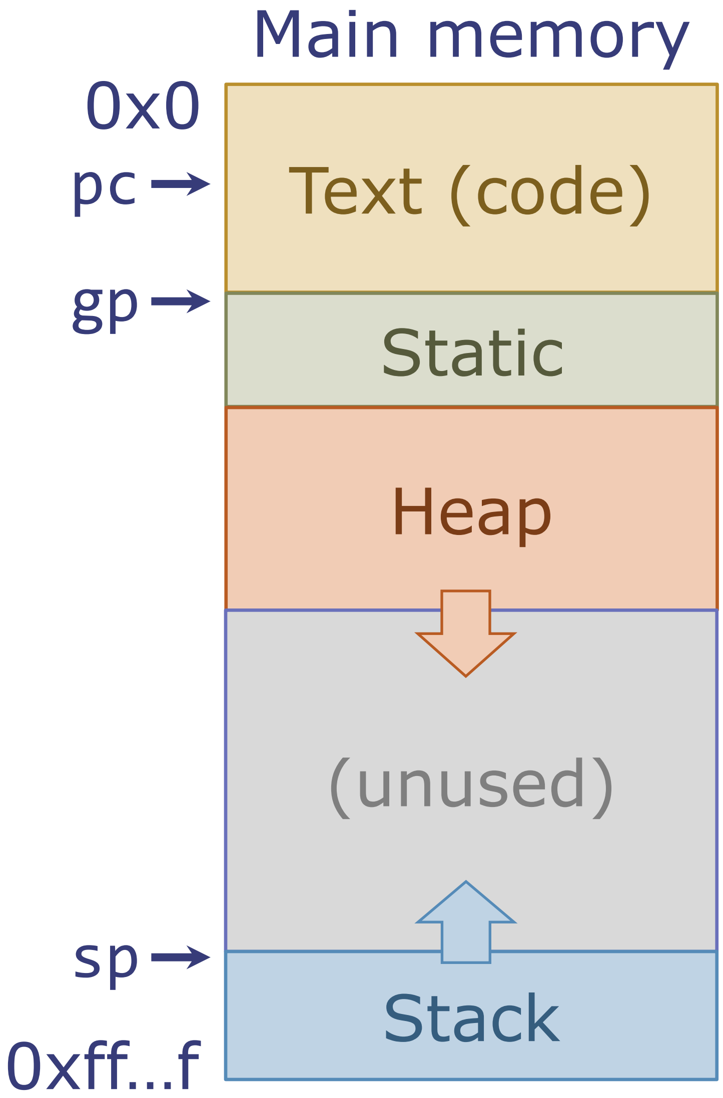
- Other complex data structures like dictionaries, structures, linked lists, etc. would follow the same methodology of passing a pointer to the data structure as the argument to the procedure along with any additional required information like number of elements, etc.

- Similarly, when the return value is a complex data structure, then the data structure is stored in memory and a pointer to the data structure is returned by the procedure.

Memory Layout

- Most programming languages (including C) have three distinct memory regions for data:
  - Stack: Holds data used by procedure calls
  - Static: Holds global variables that exist for the entire lifetime of the program
  - Heap: Holds dynamically-allocated data
    - In C, programmers manage the heap manually, allocating new data using `malloc()` and releasing it with `free()`
    - In Python, Java, and most modern languages, the heap is managed automatically: programmers create new objects (e.g., `d = dict()` in Python), but the system frees them only when it is safe (no pointers in the program point to them)
- In addition, the text region holds program code

RISC-V Memory Layout



- Text, static, and heap regions are placed consecutively, starting from low addresses
- Heap grows towards higher addresses
- Stack starts on highest address, grows towards lower addresses
- sp (stack pointer) points to top of stack
- gp (global pointer) points to start of static region

Handling Inputs and Outputs

Used in Lab 2

How do we handle Inputs and Outputs in Assembly?

- Memory Mapped I/O
  - Uses the same address space to map both memory and I/O Devices.
  - I/O Devices monitor the CPU memory requests and respond to memory requests that use the address associated with the I/O device.
  - MMIO addresses can only be used for I/O and not for regular storage.

MMIO Addresses

- Outputs:
  - 0x 4000 0000 - performing a **sw** to this address prints an ASCII character to the console corresponding to the **ASCII** equivalent of the value stored at this address
  - 0x 4000 0004 - a **sw** to this address prints a **decimal** number
  - 0x 4000 0008 - a **sw** to this address prints a **hexadecimal** number

- Inputs
  - 0x 4000 4000 - performing a **lw** from this address will read one signed word from the console.
  - Repeating a **lw** to this address will read the next input word and so on.

Memory Mapped IO Example 1

```
// load the read port into t0
li t0, 0x40004000

// read the first input
lw a0, 0(t0)
// read the second input
lw a1, 0(t0)

// add them together
add a0, a0, a1

// load the write port into t0
li t0, 0x40000004
// write the output
sw a0, 0(t0)
```

MMIO for Performance Measures

- Performance Measures
  - 0x 4000 5000 – **lw** to get **instruction count** from start of program execution
  - 0x 4000 6000 – **lw** get **performance counter** – number of instructions between turning the performance counter on and then off.
  - 0x 4000 6004
    - **sw 0** to turn **performance counting off**
    - **sw 1** to turn it **on**

Memory Mapped IO Example 2

```
// prepare to read input from console
li t0, 0x40004000
// get user input
lw a0, 0(t0)
lw a1, 0(t0)
// load the performance counter address into t1
li t1, 0x40006000
li t2, 1
// start the performance counter by storing 1 to the magic address
sw t2, 4(t1)
add a0, a0, a1
// stop the performance counter by storing 0 the the address
sw zero, 4(t1)
// prepare to print decimal to console
li t0, 0x40000004
// first print sum
sw a0, 0(t0)
// get the count from the performance counter
lw t2, 0(t1)
// print the count
sw t2, 0(t0)
```

Thank you!

Next lecture: Boolean Algebra