Procedures, Stacks, and MMIO
Components of a MicroProcessor

- **ALU**: Holds program and data
- **Register File**: Contains registers
  - `x0`, `x1`, `x2`, ..., `x31`
  - 32-bit “words”
- **Main Memory**: Stores program and data
  - 32-bit “words”
  - Address: `0x0`, `0x4`, `0x8`, `0xC`, `0x10`, `0x14`

Machine language directly reflects this structure.
Program Counter

- The PC register keeps track of the address of your current instruction.
- For non control-flow instructions, PC ← PC + 4
- Instructions and data are stored as 32-bit binary values in memory.

```
. = 0x0
blt x1, x2, label  if x1 < x2: PC ← 0x8
addi x1, x1, 1  else: PC ← PC + 4

label: xor x2, x1, x1
lw x3, 0x100(x0)  x3 = 0x12345678

. = 0x100
.word 0x12345678
```

```
Address
31 0
0x0 blt instr
0x4 addi instr
0x8 xor instr
0xC lw instr
0x10...
0x14...
0x100...
```

Address
Recap: RISC-V Calling Convention

- The calling convention specifies rules for register usage across procedures

<table>
<thead>
<tr>
<th>Symbolic name</th>
<th>Registers</th>
<th>Description</th>
<th>Saver</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0 to a7</td>
<td>x10 to x17</td>
<td>Function arguments</td>
<td>Caller</td>
</tr>
<tr>
<td>a0 and a1</td>
<td>x10 and x11</td>
<td>Function return values</td>
<td>Caller</td>
</tr>
<tr>
<td>ra</td>
<td>x1</td>
<td>Return address</td>
<td>Caller</td>
</tr>
<tr>
<td>t0 to t6</td>
<td>x5-7, x28-31</td>
<td>Temporaries</td>
<td>Caller</td>
</tr>
<tr>
<td>s0 to s11</td>
<td>x8-9, x18-27</td>
<td>Saved registers</td>
<td>Callee</td>
</tr>
<tr>
<td>sp</td>
<td>x2</td>
<td>Stack pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>gp</td>
<td>x3</td>
<td>Global pointer</td>
<td>---</td>
</tr>
<tr>
<td>tp</td>
<td>x4</td>
<td>Thread pointer</td>
<td>---</td>
</tr>
<tr>
<td>zero</td>
<td>x0</td>
<td>Hardwired zero</td>
<td>---</td>
</tr>
</tbody>
</table>
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!
Using the stack for procedures

Sample entry sequence

addi sp, sp, -8
sw ra, 0(sp)
sw a1, 4(sp)

Corresponding exit sequence

lw ra, 0(sp)
lw a1, 4(sp)
addi sp, sp, 8
jr ra
Example: Using callee-saved registers

- Implement $f$ using $s0$ and $s1$ to store temporary values

```plaintext
f:

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

- Stack contents:
  - Before call to \( f \): Unused space
  - During call to \( f \): Saved s0, Saved s1
  - After call to \( f \): Saved s0, Saved s1
Example: Using caller-saved registers

**Caller**

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

```assembly
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**

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

```assembly
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 any aN, tN, or ra registers, whose values need to be maintained past procedure call, on the stack prior to proc call and restores it upon return.

```assembly
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
```

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

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

February 25, 2021
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
  ```
Computing with large data structures

- Suppose we want to write a procedure that finds the maximum value in an array \(a[]\).
  - Assume the array is too large to be stored in registers.

- We will compare each element of \(a[]\) to the max value found so far, and update the max if \(a[i]\) is larger than current max.

- How do we pass the array \(a[]\) as an argument?
  - Pass the base address and the size of the array 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

```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);
}
```
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);
}
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
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.
- `pc` (program counter) points to the current instruction.
Handling Inputs and Outputs

Used in Lab 2
Accessing Inputs and Outputs

- Programs may want to print output to a display
- Programs may want to receive input from a keyboard
- Assign special addresses to represent I/O devices
  - Can execute `lw` or `sw` instructions to these addresses to access the devices.
Memory Mapped I/O (MMIO)

- Assign dedicated addresses to I/O devices

- MMIO addresses can only be used for I/O and not for regular storage.

- I/O Devices monitor the processor memory requests and respond to memory requests that use the address associated with the I/O device.
MMIO Addresses

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

- **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.
Memory Mapped IO Example 1

Program that reads two inputs from keyboard, adds them, and displays result on monitor.

// 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 in decimal
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: The Digital Abstraction