Procedures and Stacks

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.
Announcements

• Schedule has shifted due to snow day
Announcements

• Schedule has shifted due to snow day

• Quiz 2 is now on Thu 4/12 (one week later)
  – Apologies if this creates a conflict
  – If you have a conflict or are overloaded that week, let us know
Announcements

• Schedule has shifted due to snow day

• Quiz 2 is now on Thu 4/12 (one week later)
  – Apologies if this creates a conflict
  – If you have a conflict or are overloaded that week, let us know

• We are conducting an informal, optional survey and would love your feedback: https://goo.gl/kxvcAE
  – Thanks to those of you who have already replied!
Today’s Plan: Compiling High-Level Languages into RISC-V Programs

- Compiling simple code fragments
  - Expressions
  - Conditionals (if, if/else)
  - Loops
Today’s Plan: Compiling High-Level Languages into RISC-V Programs

• Compiling simple code fragments
  – Expressions
  – Conditionals (if, if/else)
  – Loops

• Compiling procedures
  – Calling convention
  – Program stack
  – Nested procedures
Today’s Plan: Compiling High-Level Languages into RISC-V Programs

• Compiling simple code fragments
  – Expressions
  – Conditionals (if, if/else)
  – Loops

• Compiling procedures
  – Calling convention
  – Program stack
  – Nested procedures

• Putting it all together
  – Memory layout
Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle small constants, and the `li` pseudoinstruction to load large constants (or store the constants in memory)

**Example C code**

```c
int x, y, z;
...
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```
Compiling Simple Expressions

• Assign variables to registers
• Translate operators into computational instructions
• Use register-immediate instructions to handle small constants, and the li pseudoinstruction to load large constants (or store the constants in memory)

**Example C code**

```c
int x, y, z;
...
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

**RISC-V Assembly**

```assembly
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
```

March 15, 2018
Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle small constants, and the `li` pseudoinstruction to load large constants (or store the constants in memory)

**Example C code**

```c
int x, y, z;
...
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

**RISC-V Assembly**

```
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
```
Compiling Simple Expressions

• Assign variables to registers
• Translate operators into computational instructions
• Use register-immediate instructions to handle small constants, and the `li` pseudoinstruction to load large constants (or store the constants in memory)

**Example C code**

```c
int x, y, z;
...
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

**RISC-V Assembly**

```assembly
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
```
Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle small constants, and the \texttt{li} pseudoinstruction to load large constants (or store the constants in memory)

**Example C code**

```c
int x, y, z;
...

y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

**RISC-V Assembly**

```assembly
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
```

March 15, 2018
Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle small constants, and the `li` pseudoinstruction to load large constants (or store the constants in memory)

**Example C code**

```c
int x, y, z;
...
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

**RISC-V Assembly**

```
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
```

March 15, 2018
Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle small constants, and the `li` pseudoinstruction to load large constants (or store the constants in memory)

**Example C code**

```c
int x, y, z;
...
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

**RISC-V Assembly**

```asm
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
li x14, 123456
add x14, x11, x14
```
Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle small constants, and the `li` pseudoinstruction to load large constants (or store the constants in memory)

**Example C code**

```c
int x, y, z;
...
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

**RISC-V Assembly**

```assembly
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
li x14, 123456
add x14, x11, x14
```
Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle small constants, and the li pseudoinstruction to load large constants (or store the constants in memory)

**Example C code**

```c
int x, y, z;
...
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

**RISC-V Assembly**

```assembly
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
li x14, 123456
add x14, x11, x14
or x11, x13, x14
```
Compiling Simple Expressions

• Assign variables to registers
• Translate operators into computational instructions
• Use register-immediate instructions to handle small constants, and the `li` pseudoinstruction to load large constants (or store the constants in memory)

Example C code

```c
int x, y, z;
...
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V Assembly

```riscv
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
li x14, 123456
add x14, x11, x14
or x11, x13, x14
```

March 15, 2018
Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle small constants, and the li pseudoinstruction to load large constants (or store the constants in memory)

Example C code

```c
int x, y, z;
...
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V Assembly

```assembly
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
li x14, 123456
add x14, x11, x14
or x11, x13, x14
slli x13, x10, 2
```
Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle small constants, and the `li` pseudoinstruction to load large constants (or store the constants in memory)

**Example C code**

```c
int x, y, z;
...
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

**RISC-V Assembly**

```asm
addi x13, x10, 3
li x14, 123456
add x14, x11, x14
or x11, x13, x14
slli x13, x10, 2
```

// x: x10, y: x11, z: x12
// x13, x14 used for temporaries

March 15, 2018
Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle small constants, and the `li` pseudoinstruction to load large constants (or store the constants in memory)

**Example C code**

```c
int x, y, z;
...
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

**RISC-V Assembly**

```assembly
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
li x14, 123456
add x14, x11, x14
or x11, x13, x14
slli x13, x10, 2
xor x12, x13, x11
```

March 15, 2018
• *if* statements can be compiled using branches:

<table>
<thead>
<tr>
<th>C code</th>
<th>RISC-V Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (expr) {</td>
<td>(compile expr into xN)</td>
</tr>
<tr>
<td>if-body</td>
<td>beqz xN, endif</td>
</tr>
<tr>
<td>}</td>
<td>(compile <em>if-body</em>)</td>
</tr>
<tr>
<td></td>
<td>endif:</td>
</tr>
</tbody>
</table>

March 15, 2018 L10-5
Compiling Conditionals

• *if* statements can be compiled using branches:

<table>
<thead>
<tr>
<th>C code</th>
<th>RISC-V Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if (expr) {</code></td>
<td>(compile <code>expr</code> into xN)</td>
</tr>
<tr>
<td><em>if-body</em></td>
<td><code>beqz xN, endif</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td>(compile <em>if-body</em>)</td>
</tr>
<tr>
<td></td>
<td><code>endif:</code></td>
</tr>
</tbody>
</table>

• *Example: Compile the following C code*

```c
int x, y;
...
if (x < y) {
    y = y - x;
}
```
### Compiling Conditionals

- *if* statements can be compiled using branches:

<table>
<thead>
<tr>
<th>C code</th>
<th>RISC-V Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if (expr) {</code></td>
<td>(compile <code>expr</code> into xN)</td>
</tr>
<tr>
<td><code>    if-body</code></td>
<td><code>beqz xN, endif</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td>(compile <code>if-body</code>)</td>
</tr>
<tr>
<td></td>
<td><code>endif:</code></td>
</tr>
</tbody>
</table>

- **Example:** Compile the following C code

```c
int x, y; // x: x10, y: x11
...
if (x < y) {
    y = y - x;
} endif:
```
Compiling Conditionals

- *if* statements can be compiled using branches:

<table>
<thead>
<tr>
<th>C code</th>
<th>RISC-V Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (expr) {</td>
<td>(compile expr into xN)</td>
</tr>
<tr>
<td>if-body</td>
<td>beqz xN, endif</td>
</tr>
<tr>
<td>}</td>
<td>(compile if-body)</td>
</tr>
<tr>
<td>endif:</td>
<td></td>
</tr>
</tbody>
</table>

- **Example:** Compile the following C code

```c
int x, y; // x: x10, y: x11
...
if (x < y) {
  y = y - x;
}
endif:
```
Compiling Conditionals

- *if* statements can be compiled using branches:

<table>
<thead>
<tr>
<th>C code</th>
<th>RISC-V Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (expr) {</td>
<td>(compile expr into xN)</td>
</tr>
<tr>
<td><em>if-body</em></td>
<td>beqz xN, endif</td>
</tr>
<tr>
<td>}</td>
<td>(compile <em>if-body</em>)</td>
</tr>
<tr>
<td></td>
<td>endif:</td>
</tr>
</tbody>
</table>

- **Example: Compile the following C code**

```c
int x, y; // x: x10, y: x11
...
if (x < y) {
    y = y - x;
}
```

```asm
slt x12, x10, x11
beqz x12, endif
```
Compiling Conditionals

- *if* statements can be compiled using branches:

<table>
<thead>
<tr>
<th>C code</th>
<th>RISC-V Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (expr) {</td>
<td>(compile expr into xN)</td>
</tr>
<tr>
<td>if-body</td>
<td>beqz xN, endif</td>
</tr>
<tr>
<td>}</td>
<td>(compile if-body)</td>
</tr>
<tr>
<td></td>
<td>endif:</td>
</tr>
</tbody>
</table>

- **Example: Compile the following C code**

```c
int x, y; // x: x10, y: x11
...
if (x < y) {
  y = y - x;
}
```
Compiling Conditionals

• *if* statements can be compiled using branches:

<table>
<thead>
<tr>
<th>C code</th>
<th>RISC-V Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if (expr) {</code></td>
<td>(compile <code>expr</code> into <code>xN</code>)</td>
</tr>
<tr>
<td><code>  if-body</code></td>
<td><code>beqz xN, endif</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td>(compile <code>if-body</code>)</td>
</tr>
<tr>
<td><code>endif:</code></td>
<td></td>
</tr>
</tbody>
</table>

• **Example: Compile the following C code**

```c
int x, y;  // x: x10, y: x11
...
if (x < y) {
   y = y - x;
}
```

```assembly
     slt x12, x10, x11
beqz x12, endif
     sub x11, x11, x10
endif:       
```

We can sometimes combine `expr` and the branch

```assembly
     bge x10, x11, endif
sub x11, x11, x10
endif:       
```
### Compiling Conditionals

- *if-else* statements are similar:

<table>
<thead>
<tr>
<th>C code</th>
<th>RISC-V Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ((expr)) {</td>
<td>(compile (expr) into (xN))</td>
</tr>
<tr>
<td>\hspace{1em} if-body</td>
<td>beqz (xN), else</td>
</tr>
<tr>
<td>} else {</td>
<td>(compile if-body)</td>
</tr>
<tr>
<td>\hspace{1em} else-body</td>
<td>j endif</td>
</tr>
<tr>
<td>}</td>
<td>else:</td>
</tr>
<tr>
<td></td>
<td>(compile else-body)</td>
</tr>
<tr>
<td></td>
<td>endif:</td>
</tr>
</tbody>
</table>
Compiling Loops

- Loops can be compiled using *backward* branches:

  C code                        RISC-V Assembly
  
  while (expr) {
    while-body
  }

  (compile expr into xN)
  beqz xN, endwhile
  (compile while-body)
  j while
  endwhile:
Compiling Loops

- Loops can be compiled using *backward* branches:

  C code
  ```c
  while (expr) {
    while-body
  }
  ```

  RISC-V Assembly
  ```
  while:
  (compile expr into xN)
  beqz xN, endwhile
  (compile while-body)
  j while
  endwhile:
  ```
Compiling Loops

- Loops can be compiled using *backward* branches:

  C code
  ```c
  while (expr) {
    while-body
  }
  ```

  RISC-V Assembly
  ```assembly
  while:             (compile expr into xN)
    beqz xN, endwhile
    (compile while-body)
    j while
  endwhile:
  ```

- *Can you write a version that executes fewer instructions?*
Compiling Loops

- Loops can be compiled using *backward* branches:

  C code
  ```c
  while (expr) {
    while-body
  }
  ```

  RISC-V Assembly
  ```
  while:  
    (compile expr into xN)
    beqz xN, endwhile
    (compile while-body)
  j while
  endwhile:  // Version with one branch
  // or jump per iteration
  j compare
  loop:  
    (compile while-body)
  compare:  
    (compile expr into xN)
    bnez xN, loop
  ```

- *Can you write a version that executes fewer instructions?*
Procedures

• Procedure (a.k.a. function or subroutine): Reusable code fragment that performs a specific task

```c
int gcd(int a, int b) {
    int x = a;
    int y = b;
    while (x != y) {
        if (x > y) {
            x = x - y;
        } else {
            y = y - x;
        }
    }
    return x;
}
```
• Procedure (a.k.a. function or subroutine): Reusable code fragment that performs a specific task
  – Single named entry point

```c
int gcd(int a, int b) {
    int x = a;
    int y = b;
    while (x != y) {
        if (x > y) {
            x = x - y;
        } else {
            y = y - x;
        }
    }
    return x;
}
```
Procedures

- Procedure (a.k.a. function or subroutine): Reusable code fragment that performs a specific task
  - Single named entry point
  - Zero or more formal parameters

```c
int gcd(int a, int b) {
    int x = a;
    int y = b;
    while (x != y) {
        if (x > y) {
            x = x - y;
        } else {
            y = y - x;
        }
    }
    return x;
}
```
Procedures

- Procedure (a.k.a. function or subroutine): Reusable code fragment that performs a specific task
  - Single named entry point
  - Zero or more formal parameters
  - Local storage

```c
int gcd(int a, int b) {
    int x = a;
    int y = b;
    while (x != y) {
        if (x > y) {
            x = x - y;
        } else {
            y = y - x;
        }
    }
    return x;
}
```
Procedures

• Procedure (a.k.a. function or subroutine): Reusable code fragment that performs a specific task
  – Single named entry point
  – Zero or more formal parameters
  – Local storage
  – Returns control to the caller when finished

```c
int gcd(int a, int b) {
    int x = a;
    int y = b;
    while (x != y) {
        if (x > y) {
            x = x - y;
        } else {
            y = y - x;
        }
    }
    return x;
}
```
Procedures

• Procedure (a.k.a. function or subroutine): Reusable code fragment that performs a specific task
  – Single named entry point
  – Zero or more formal parameters
  – Local storage
  – Returns control to the caller when finished

• Using procedures enables abstraction and reuse
  – Compose large programs from collections of simple procedures

```c
int gcd(int a, int b) {
    int x = a;
    int y = b;
    while (x != y) {
        if (x > y) {
            x = x - y;
        } else {
            y = y - x;
        }
    }
    return x;
}

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

coprimes(5, 10); // false
coprimes(9, 10); // true
```
Implementing Procedures

• Option 1: Inlining
  – Compiler substitutes procedure call with body
Implementing Procedures

• Option 1: Inlining
  – Compiler substitutes procedure call with body
  – Problems?
Implementing Procedures

• Option 1: Inlining
  – Compiler substitutes procedure call with body
  – *Problems?*
    
  Code size
Implementing Procedures

• Option 1: Inlining
  – Compiler substitutes procedure call with body
  – *Problems?*
    
    Code size
    Recursion

    int factorial(int n) {
      if (n > 0) {
        return n * factorial(n - 1);
      } else {
        return 1;
      }
    }
Implementing Procedures

• Option 1: Inlining
  – Compiler substitutes procedure call with body
  – *Problems?*
    Code size
    Recursion

```c
int factorial(int n) {
  if (n > 0) {
    return n * factorial(n - 1);
  } else {
    return 1;
  }
}
```

• Option 2: Linking
  – Produce separate code for each procedure
  – Caller evaluates input arguments, stores them and transfers control to the callee’s entry point
  – Callee runs, stores result, transfers control to caller
Procedure Linking: Key Questions

• How to communicate arguments and return values?
Procedure Linking: Key Questions

• How to communicate arguments and return values?

• How to transfer control to callee and back to caller?
Procedure Linking: Key Questions

• How to communicate arguments and return values?

• How to transfer control to callee and back to caller?

• How should caller and callee use registers? What if they need to use the same register?
Procedure Linking: Key Questions

• How to communicate arguments and return values?

• How to transfer control to callee and back to caller?

• How should caller and callee use registers? What if they need to use the same register?

• How to let procedures use more storage than can fit in registers?
Calling Convention

• The calling convention specifies rules for register usage across procedures

• Every register is either callee-saved or caller-saved
Calling Convention

• The calling convention specifies rules for register usage across procedures

• Every register is either \textit{callee-saved} or \textit{caller-saved}

• A \textit{callee-saved} register is \textit{preserved} across function calls
  – If callee wants to use it, it must save its value elsewhere and restore it before returning control to the caller
Calling Convention

• The calling convention specifies rules for register usage across procedures

• Every register is either callee-saved or caller-saved

• A callee-saved register is preserved across function calls
  – If callee wants to use it, it must save its value elsewhere and restore it before returning control to the caller

• 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 elsewhere before transferring control to the callee
RISC-V Calling Convention

• RISC-V calling convention gives symbolic names to registers x0-x31 to denote their role:

<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>
</tbody>
</table>
RISC-V Calling Convention

- RISC-V calling convention gives symbolic names to registers x0-x31 to denote their role:

<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>
</tbody>
</table>
RISC-V Calling Convention

- RISC-V calling convention gives symbolic names to registers x0-x31 to denote their role:

<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>
</tbody>
</table>
RISC-V Calling Convention

• RISC-V calling convention gives symbolic names to registers x0-x31 to denote their role:

<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>
</tbody>
</table>

- Example: What does `add t0, s3, a0` translate to?
RISC-V Calling Convention

- RISC-V calling convention gives symbolic names to registers x0-x31 to denote their role:

<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>
</tbody>
</table>

- Example: What does add t0, s3, a0 translate to?
  
  add x5, x19, x10
Calling a Procedure

Caller places arguments in registers a0-a7

- Caller transfers control to callee using jump-and-link to capture the return address in register ra
  - jal ra, label

- Callee runs, places results in registers a0 and a1

- Callee transfers control to caller using jump-register
  - jalr rd, imm(rs1) : R[rd] ← pc + 4;
    pc ← {(R[rs1] + imm)[31:1], 1'b0}
  - Pseudoinstruction jr rs1 ← jalr x0, 0(rs1)
  - Pseudoinstruction ret ← jr ra ← jalr x0, 0(ra)
Calling a Procedure

Caller places arguments in registers a0-a7

• Caller transfers control to callee using jump-and-link to capture the return address in register ra
  - jal rd, imm : R[rd] ← pc + 4; pc ← pc + imm
  - jal ra, label

• Callee runs, places results in registers a0 and a1
• Callee transfers control to caller using jump-register
  - jalr rd, imm(rs1) : R[rd] ← pc + 4;
    pc ← {(R[rs1] + imm)[31:1], 1'b0}
  - Pseudoinstruction jr rs1 ↔ jalr x0, 0(rs1)
  - Pseudoinstruction ret ↔ jr ra ↔ jalr x0, 0(ra)
Calling a Procedure

jal ra, label

Caller places arguments in registers a0-a7

- Caller transfers control to callee using jump-and-link to capture the return address in register ra
  - jal rd, imm : R[rd] ← pc + 4; pc ← pc + imm
  - Pseudoinstruction jal label ← jal ra, label

- Callee runs, places results in registers a0 and a1
- Callee transfers control to caller using jump-register
  - jalr rd, imm(rs1) : R[rd] ← pc + 4;
    pc ← {(R[rs1] + imm)[31:1], 1′b0}
  - Pseudoinstruction jr rs1 ← jalr x0, 0(rs1)
  - Pseudoinstruction ret ← jr ra ← jalr x0, 0(ra)
Calling a Procedure

jal ra, label

**Caller** places arguments in registers a0-a7

- Caller transfers control to callee using jump-and-link to capture the return address in register ra
  - jal rd, imm : R[rd] \(\leftarrow\) pc + 4; pc \(\leftarrow\) pc + imm
  - Pseudoinstruction jal label \(\leftrightarrow\) jal ra, label

- Callee runs, places results in registers a0 and a1
- Callee transfers control to caller using jump-register
- Callee transfers control to caller using jump-register
  - jalr rd, imm(rs1) : R[rd] \(\leftarrow\) pc + 4;
    pc \(\leftarrow\) \{(R[rs1] + imm)[31:1], 1'b0\}
  - Pseudoinstruction jr rs1 \(\leftrightarrow\) jalr x0, 0(rs1)
  - Pseudoinstruction ret \(\leftrightarrow\) jr ra \(\leftrightarrow\) jalr x0, 0(ra)
Calling a Procedure

jal ra, label

**Caller** places arguments in registers **a0-a7**

- Caller transfers control to callee using jump-and-link to capture the return address in register **ra**
  - jal rd, imm : R[rd] ← pc + 4; pc ← pc + imm
  - Pseudoinstruction jal label ← jal ra, label

- Callee runs, places results in registers **a0** and **a1**

- Callee transfers control to caller using jump-register
  - jalr rd, imm(rs1) : R[rd] ← pc + 4;
    pc ← {(R[rs1] + imm)[31:1], 1'b0}
    pc ← {(R[rs1] + imm)[31:1], 1'b0}
  - Pseudoinstruction jr rs1 ← jalr x0, 0(rs1)
  - Pseudoinstruction ret ← jr ra ← jalr x0, 0(ra)
Calling a Procedure

\[ \text{jalr } x0, 0(\text{rs1}) \]

\[ \text{jal } ra, \text{ label} \]

**Caller** places arguments in registers \texttt{a0-a7}

- Caller transfers control to callee using jump-and-link to capture the return address in register \texttt{ra}
  - \texttt{jal rd, imm : R[rd] \leftarrow pc + 4; pc \leftarrow pc + imm}
  - Pseudoinstruction \texttt{jal label \leftrightarrow jal ra, label}

- Callee runs, places results in registers \texttt{a0} and \texttt{a1}

- Callee transfers control to caller using jump-register
  - \texttt{jalr rd, imm(rs1) : R[rd] \leftarrow pc + 4; pc \leftarrow (R[rs1] + imm)[31:1], 1'b0}
  - Pseudoinstruction \texttt{jr rs1 \leftrightarrow jalr x0, 0(rs1)}
  - Pseudoinstruction \texttt{jr rs1 \leftrightarrow jalr x0, 0(rs1)}
  - Pseudoinstruction \texttt{ret \leftrightarrow jr ra \leftrightarrow jalr x0, 0(ra)}
Calling a Procedure

jr ra   jalr x0, 0(ra)
jalr x0, 0(rs1)
jal ra, label

**Caller places arguments in registers a0-a7**

- Caller transfers control to callee using jump-and-link to capture the return address in register ra
  - jal rd, imm : R[rd] ← pc + 4; pc ← pc + imm
  - Pseudoinstruction jal label ← jal ra, label

- Callee runs, places results in registers a0 and a1

- Callee transfers control to caller using jump-register
  - jalr rd, imm(rs1) : R[rd] ← pc + 4;
    pc ← {(R[rs1] + imm)[31:1], 1'b0}
  - Pseudoinstruction ret ← jr ra ← jalr x0, 0(ra)
  - Pseudoinstruction jr rs1 ← jalr x0, 0(rs1)
    Pseudoinstruction ret ← jr ra ← jalr x0, 0(ra)
Calling a Procedure: Example

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

```c
int sum(int a, int b) {
    return a + b;
}
```
Calling a Procedure: Example

**Caller**

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

li a0, 1
li a1, 2
jal sum
```

**Callee**

```
int sum(int a, int b) {
    return a + b;
}
```
Calling a Procedure: Example

### Caller

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

li a0, 1
li a1, 2
jal sum
// a0 = sum(x, y)
```

### Callee

```c
int sum(int a, int b) {
    return a + b;
}
```
# Calling a Procedure: Example

<table>
<thead>
<tr>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int x = 1;</code></td>
<td><code>int sum(int a, int b) {</code></td>
</tr>
<tr>
<td><code>int y = 2;</code></td>
<td><code>    return a + b;</code></td>
</tr>
<tr>
<td><code>int z = sum(x, y);</code></td>
<td>}</td>
</tr>
<tr>
<td><code>int w = sum(z, y);</code></td>
<td></td>
</tr>
</tbody>
</table>

```
li a0, 1
li a1, 2
jal sum
// a0 = sum(x, y)
li a1, 2
jal sum
// a0 = sum(z, y)
```

March 15, 2018
Calling a Procedure: Example

### Caller

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

### Callee

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

```
li a0, 1
li a1, 2
jal sum
// a0 = sum(x, y)
li a1, 2
jal sum
// a0 = sum(z, y)
```

March 15, 2018
Calling a Procedure: Example

**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
jal sum
// a0 = sum(x, y)
li a1, 2
jal sum
// a0 = sum(z, y)
```

**Callee**

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

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

Each invocation of `sum` returns control to the right address

```assembly
jal sum ↔ jal ra, sum
ret ↔ jr ra
```
Calling a Procedure: Example

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

```assembly
li a0, 1
li a1, 2
jal sum
// a0 = sum(x, y)
```

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

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

Each invocation of sum returns control to the right address:
```
jal sum ↔ jal ra, sum
ret ↔ jr ra
```
Calling a Procedure: Example

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

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

** Caller **

```assembly
li a0, 1
li a1, 2
jal sum  // a0 = sum(x, y)
```

** Callee **

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

Each invocation of sum returns control to the right address:

```assembly
jal sum ↔ jal ra, sum
ret ↔ jr ra
```
## Calling a Procedure: Example

<table>
<thead>
<tr>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x = 1;</td>
<td>int sum(int a, int b) { return a + b; }</td>
</tr>
<tr>
<td>int y = 2;</td>
<td>}</td>
</tr>
<tr>
<td>int z = sum(x, y);</td>
<td></td>
</tr>
<tr>
<td>int w = sum(z, y);</td>
<td></td>
</tr>
<tr>
<td>li a0, 1</td>
<td>sum:</td>
</tr>
<tr>
<td>li a1, 2</td>
<td>add a0, a0, a1</td>
</tr>
<tr>
<td>jal sum</td>
<td>ret</td>
</tr>
<tr>
<td>// a0 = sum(x, y)</td>
<td>Each invocation of sum returns</td>
</tr>
<tr>
<td>li a1, 2</td>
<td>control to the right address</td>
</tr>
<tr>
<td>jal sum</td>
<td>jal sum ↔ jal ra, sum</td>
</tr>
<tr>
<td>// a0 = sum(z, y)</td>
<td>ret ↔ jr ra</td>
</tr>
</tbody>
</table>
Calling a Procedure: Example

**Caller**

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

**Callee**

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

Each invocation of `sum` returns control to the right address:

```
jal sum ↔ jal ra, sum
ret ↔ jr ra
```
Calling a Procedure: Example

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

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

**Caller**

```
li a0, 1
li a1, 2
jal sum
// a0 = sum(x, y)
```

**Callee**

```
add a0, a0, a1
ret
```

```
jal sum ← jal ra, sum
ret ← jr ra
```

Each invocation of sum returns control to the right address.
Calling a Procedure: Example

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

li a0, 1
li a1, 2
jal sum
// a0 = sum(x, y)
li a1, 2
jal sum
// a0 = sum(z, y)
```

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

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

Why is second
li a1, 2 needed?
Calling a Procedure: Example

<table>
<thead>
<tr>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x = 1;</td>
<td>int sum(int a, int b) {</td>
</tr>
<tr>
<td>int y = 2;</td>
<td>return a + b;</td>
</tr>
<tr>
<td>int z = sum(x, y);</td>
<td>}</td>
</tr>
<tr>
<td>int w = sum(z, y);</td>
<td></td>
</tr>
<tr>
<td>li a0, 1</td>
<td></td>
</tr>
<tr>
<td>li a1, 2</td>
<td></td>
</tr>
<tr>
<td>jal sum</td>
<td></td>
</tr>
<tr>
<td>// a0 = sum(x, y)</td>
<td></td>
</tr>
<tr>
<td>li a1, 2</td>
<td></td>
</tr>
<tr>
<td>jal sum</td>
<td></td>
</tr>
<tr>
<td>// a0 = sum(z, y)</td>
<td></td>
</tr>
</tbody>
</table>

Why is second li a1, 2 needed? Callee may have modified a1 (caller doesn’t see implementation of sum!)
Procedure Storage Needs

• Procedures often need storage beyond registers:
  - To save callee-saved registers that they want to use and caller-saved registers that they want to preserve
Procedure Storage Needs

- Procedures often need storage beyond registers:
  - To save callee-saved registers that they want to use and caller-saved registers that they want to preserve
  - To pass arguments/results that do not fit in arg/res registers
Procedure Storage Needs

- Procedures often need storage beyond registers:
  - To save callee-saved registers that they want to use and caller-saved registers that they want to preserve
  - To pass arguments/results that do not fit in arg/res registers
  - To store local variables that cannot fit in registers
Procedure Storage Needs

• Procedures often need storage beyond registers:
  – To save callee-saved registers that they want to use and caller-saved registers that they want to preserve
  – To pass arguments/results that do not fit in arg/res registers
  – To store local variables that cannot fit in registers

• We only need to access the local storage of the currently-executing procedure
Procedure Storage Needs

- Procedures often need storage beyond registers:
  - To save callee-saved registers that they want to use and caller-saved registers that they want to preserve
  - To pass arguments/results that do not fit in arg/res registers
  - To store local variables that cannot fit in registers

- We only need to access the local storage of the currently-executing procedure

- A stack is the right data structure for this purpose
  - Stack = Last-In First-Out (LIFO) queue
  - Can push/pop data into/from stack, and access the top element
RISC-V Stack

- Stack is in memory → need a register to point to it
  - In RISC-V, stack pointer `sp` is `x2`
RISC-V Stack

- Stack is in memory → need a register to point to it
  - In RISC-V, stack pointer `sp` is `x2`

- Stack grows down from higher to lower addresses
  - Push decreases `sp`
  - Pop increases `sp`
RISC-V Stack

- Stack is in memory → need a register to point to it
  - In RISC-V, stack pointer \( sp \) is \( x2 \)

- Stack grows down from higher to lower addresses
  - Push decreases \( sp \)
  - Pop increases \( sp \)

- \( sp \) points to top of stack (last pushed element)
RISC-V Stack

- Stack is in memory → need a register to point to it
  - In RISC-V, stack pointer \( sp \) is \( x2 \)

- Stack grows down from higher to lower addresses
  - Push decreases \( sp \)
  - Pop increases \( sp \)

- \( sp \) points to top of stack (last pushed element)

- 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:

```c
int f(int x, int y) {
    return (x + 3) | (y + 123456);
}
```
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);
  }
  ```
Example: Using callee-saved registers

- Implement \( f \) using \( s0 \) and \( s1 \) to store temporary values.

\[
\begin{align*}
f &: \quad \text{int } f(\text{int } x, \text{int } y) \{ \\
    & \quad \text{return } (x + 3) \mid (y + 123456); \\
\}
\end{align*}
\]

```assembly
addi s0, a0, 3
li s1, 123456
add s1, a1, s1
or a0, s0, s1
ret
```
Example: Using callee-saved registers

- Implement \( f \) using \( s0 \) and \( s1 \) to store temporary values.

\[
\begin{align*}
\text{f:} \\
\text{addi } s0, a0, 3 \\
\text{li } s1, 123456 \\
\text{add } s1, a1, s1 \\
\text{or } a0, s0, s1 \\
\text{ret}
\end{align*}
\]

\[
\text{int } f(int x, int y) \{ \\
\quad \text{return } (x + 3) | (y + 123456); \\
\}
\]
Example: Using callee-saved registers

- Implement \( f \) using \( s0 \) and \( s1 \) to store temporary values

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

\( f \):

```bash
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
ret
```
Example: Using callee-saved registers

- Implement f using s0 and s1 to store temporary values

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

f:
```assembly
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
ret
```
Example: Using callee-saved registers

• Stack contents:

Before call to f

R[sp] →

unused space
Example: Using callee-saved registers

- Stack contents:

  Before call to f
  
  | unused space |
  |             |
  |             |
  |             |
  |             |
  |             |
  |             |
  |             |

  During call to f
  
  | Saved s1    |
  | Saved s0    |
  |             |
  |             |
  |             |
  |             |
  |             |
  |             |
Example: Using callee-saved registers

- Stack contents:

  Before call to f
  
  ![Before call stack diagram]

  During call to f
  
  ![During call stack diagram]

  After call to f
  
  ![After call stack diagram]
Nested Procedures

- If a procedure calls another procedure, it needs to save its own return address
  - Remember that ra is caller-saved
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;
}
```
Nested Procedures

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

• Example:

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

```assembly
jal gcd // overwrites ra
addi a0, a0, -1
sltiu a0, a0, 1

ret // needs original ra
```
Nested Procedures

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

• Example:

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

coprimes:
```
addi sp, sp, -4
sw ra, 0(sp)
jal gcd    // overwrites ra
addi a0, a0, -1
sltiu a0, a0, 1
```

ret    // needs original ra
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)
  jal 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
Recursive Procedures

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

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

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

- Recursive procedures are just one particular case of nested procedures

- Example:

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

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

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

```
fib:
    li t0, 2
    blt a0, t0, fib_done

fib_done:
    ret
```
Recursive Procedures

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

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

```
fib:
    li t0, 2
    blt a0, t0, fib_done
    addi a0, a0, -1
    jal fib
    mv t0, s0
    addi a0, a0, -2
    jal fib
    add a0, s0, a0
    fib_done:
    ret
```
Recursive Procedures

- Recursive procedures are just one particular case of nested procedures

Example:

```c
int fib(int n) {
    if (n < 2) return n;
    else return fib(n-1) + fib(n-2);
}
```

```
fib:
    li t0, 2
    blt a0, t0, fib_done
    addi sp, sp, -8
    sw ra, 4(sp)
    sw s0, 0(sp)
    addi a0, a0, -1
    jal fib
    mv t0, s0
    mv s0, a0 // save fib(n-1)
    addi a0, t0, -2
    jal fib
    add a0, s0, a0

fib_done:
    ret
```
Recursive Procedures

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

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

```assembly
fib:
li t0, 2
blt a0, t0, fib_done
addi sp, sp, -8
sw ra, 4(sp)
sw s0, 0(sp)
addi a0, a0, -1
jal fib
mv t0, s0
mv s0, a0 // save fib(n-1)
addi a0, t0, -2
jal fib
add a0, s0, a0
lw s0, 0(sp)
lw ra, 4(sp)
addi sp, sp, 8
fib_done:
ret
```
Stack Frames

- A procedure call’s stack frame is the region of the stack holding its saved registers and local variables.
Stack Frames

• A procedure call’s stack frame is the region of the stack holding its saved registers and local variables
• Compilers use a consistent stack frame convention:
Stack Frames

- A procedure call’s stack frame is the region of the stack holding its saved registers and local variables.
- Compilers use a consistent stack frame convention:
Stack Frames

- A procedure call’s stack frame is the region of the stack holding its saved registers and local variables.
- Compilers use a consistent stack frame convention:

Before procedure call:
- Extra arguments (if any)
- Unused space

During procedure call:
- Extra arguments (if any)
- Local variables (if any)
- Saved registers (if any)
- Saved ra
- Saved argument registers (if any)
- Extra arguments (if any)
Stack Frames

• A procedure call’s stack frame is the region of the stack holding its saved registers and local variables
• Compilers use a consistent stack frame convention:
Memory Layout

• Most programming languages (including C) have three distinct memory regions for data:
  – **Stack**: Holds data used by procedure calls.
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.
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.
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()`.
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).
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 segment holds program code.
RISC-V Memory Layout

Main memory

0x0

Text (code)

Static

Heap

(unused)

Stack

0xff...f
RISC-V Memory Layout

- **Text, static, and heap** regions are placed consecutively, starting from low addresses.
RISC-V Memory Layout

- Text, static, and heap regions are placed consecutively, starting from low addresses.
- Heap grows towards higher addresses.

Diagram:

```
Main memory

0x0

Text (code)

Static

Heap

(unused)

Stack

0xff...f
```
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.

Diagram:

- 0x0: Main memory
  - Text (code)
  - Static
  - Heap (unused)
  - Stack

March 15, 2018

L10-23
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.
• **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.
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.
Thank you!

Next lecture: Implementing a single-cycle RISC-V processor