Compiling Code, Procedures and Stacks
RISC-V Recap

- Computational Instructions executed by ALU
  - Register-Register: `op dest, src1, src2`
  - Register-Immediate: `op dest, src1, const`

- Control flow instructions
  - Unconditional: `jal` and `jalr`
  - Conditional: `comp src1, src2, label`

- Loads and Stores
  - `lw dest, offset(base)`
  - `sw src, offset(base)`
  - Base is a register, offset is a small constant

- Pseudoinstructions
  - Shorthand for other instructions
Dealing with Constants

- **Execute** \( a = b + 3 \)
  - Small constants (12-bit) can be handled via Register-Immediate ALU operations
  
  \[
  \text{addi } x1, x2, 3
  \]

- **Execute** \( a = b + 0x123456 \)
  - Largest 12 bit 2’s complement constant is \( 2^{11} - 1 = 2047 (0x7FF) \)
  - Use \textit{li} pseudoinstruction to set register to large constant
  
  \[
  \text{li } x4, 0x123456 \\
  \text{lui } x4, 0x123 \\
  \text{addi } x4, x4, 0x456
  \]
  
  \[
  x4 = 0x123000
  \]

- Can also use \textit{li} pseudoinstruction for small constants
  
  \[
  \text{li } x4, 0x12 \]
  
  \[
  \text{addi } x4, x0, 0x12
  \]
Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle operations with small constants
- Use the `li` pseudoinstruction for large constants

**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
li x14, 123456
add x14, x11, x14
or x11, x13, x14
slli x13, x10, 2
xor x12, x13, x11
```
### 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;
...
if (x < y) {
    y = y - x;
}
```

We can sometimes combine *expr* and the branch

```risc-v
// x: x10, y: x11
slt x12, x10, x11
beqz x12, endif
sub x11, x11, x10
endif:
```

```risc-v
bge x10, x11, endif
sub x11, x11, x10
endif:
```
### Compiling Conditionals

- *if-else* statements are similar:

```c
if (expr) {
    if-body
} else {
    else-body
}
```

```asm
(compile expr into xN)
beqz xN, else
(compile if-body)
j endif
else:
    (compile else-body)
endif:
```
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?
Putting it all together

C code

while (x != y) {
    if (x > y) {
        x = x - y;
    } else {
        y = y - x;
    }
}

RISC-V Assembly

// x: x10, y: x11
j compare
loop:
    (compile while-body)
compare:
    bne x10, x11, loop
Putting it all together

C code

```c
while (x != y) {
    if (x > y) {
        x = x - y;
    } else {
        y = y - x;
    }
}
```

RISC-V Assembly

```assembly
// x: x10, y: x11
j compare
loop:
    ble x10, x11 else
    sub x10, x10, x11
    j endif
else:
    sub x11, x11, x10
endif:
compare:
    bne x10, x11, loop
```
Procedures

C code

```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;
}
```

RISC-V Assembly

```assembly
// x: x10, y: x11
j compare
loop:
    ble x10, x11 else
    sub x10, x10, x11
    j endif
else:
    sub x11, x11, x10
endif:
compare:
    bne x10, x11, loop
```
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 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
```
Managing a procedure’s register space

- A caller uses the same register set as the called procedure
  - We could have a convention regarding how the registers are divided between the caller and callee
    - Not satisfactory – If proc A calls proc B calls proc C, it would be difficult to manage such a division of register space, which was scarce to begin with
  - Better solution, a caller should not rely on how the called procedure manages its register space

- Either the caller or the callee saves the caller’s registers in memory and restores them when the procedure call has completed execution
Implementing procedures

- A caller needs to pass parameters to the called procedure, as well as get results back from the called procedure
  - both are done through registers
- A procedure can be called from many different places
  - The caller can get to the called procedure code simply by executing a unconditional jump instruction
  - However, to return to the correct place in the calling procedure, the called procedure has to know which of the possible return addresses it should use

Return address must be saved and passed to the called procedure!
Procedure Linking

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

\[
\text{proc\_call: jal ra, label}
\]
1. Stores address of proc\_call + 4 in register ra (return address register)
2. Jumps to instruction at address label where label is the name of the procedure
3. After executing procedure, j ra to return to caller and continue execution

\[
\begin{align*}
\text{...} & \quad \text{jal ra, sum} \\
[0x100] & \quad \text{...} \\
\text{...} & \quad \text{jal ra, sum} \\
[0x678] & \quad \text{...}
\end{align*}
\]

\[
\text{sum:}
\]

\[
\begin{align*}
\text{ra = 0x104} & \quad \text{jr ra} \\
\text{ra = 0x67C} & \quad 1^\text{st time: j 0x104} \\
\text{...} & \quad 2^\text{nd time: j 0x67C}
\end{align*}
\]
**Procedure calls: Complications**

- Suppose proc A calls proc B calls C
  - a single return address register won’t work; the return address for proc B would wipe out the return address for proc A!
  - a similar complication arises in the memory space where the registers of proc A are saved – this space has to be different from the place where the registers of proc B are saved
Procedure Storage Needs

- Basic requirements for procedure calls:
  - Input arguments
  - Return address
  - 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 all this data known as the procedure’s activation record.
Insight (ca. 1960): We Need a Stack!

- Need data structure to hold activation records
- Activation records are allocated and deallocated in last-in-first-out (LIFO) order
- Stack: push, pop, access to top element
- We only need to access to the activation record of the currently executing procedure
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!
Using the stack

Sample entry sequence
  addi sp, sp, -N
  sw ra, 0(sp)
  sw a0, 4(sp)

Corresponding Exit sequence
  lw ra, 0(sp)
  lw a0, 4(sp)
  lw a0, 4(sp)
  addi sp, sp, N
Calling Convention

- The calling convention specifies rules for register usage across procedures.
- 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>
<tr>
<td>zero</td>
<td>x0</td>
<td>Hardwired zero</td>
<td>---</td>
</tr>
</tbody>
</table>
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 on the stack 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 on stack and restore it before returning control to the caller
  - Saved registers (sN), stack pointer (sp)
Example: Using callee-saved registers

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

\[
\text{int } f(\text{int } x, \text{int } y) \{
\text{    return } (x + 3) | (y + 123456);
\}
\]

\( f \):

\[
\begin{align*}
\text{addi } sp, sp, -8 & \quad \text{// allocate 2 words (8 bytes) on stack} \\
\text{sw } s0, 4(sp) & \quad \text{// save } s0 \\
\text{sw } s1, 0(sp) & \quad \text{// save } s1 \\
\text{addi } s0, a0, 3 & \\
\text{li } s1, 123456 & \\
\text{add } s1, a1, s1 & \\
\text{or } a0, s0, s1 & \\
\text{lw } s1, 0(sp) & \quad \text{// restore } s1 \\
\text{lw } s0, 4(sp) & \quad \text{// restore } s0 \\
\text{addi } sp, sp, 8 & \quad \text{// deallocate 2 words from stack} \\
\text{ret} & \quad \text{// (restore } sp) \\
\end{align*}
\]
Example: Using callee-saved registers

- Stack contents:

  Before call to \( f \):
  - \( R[sp] \) point
  - Unused space

  During call to \( f \):
  - \( R[sp] \) point
  - Saved \( s1 \)
  - Saved \( s0 \)

  After call to \( f \):
  - \( R[sp] \) point
  - Saved \( s1 \)
  - Saved \( s0 \)
Example: Using caller-saved registers

**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;
}
```

```asm
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)
lw a1, 4(sp)  // restore y
jal ra, sum
// a0 = sum(z, y)
lw ra, 0(sp)
addi sp, sp, 8
```

*Why did we save and restore a1?*

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