Compiling Code, and Implementing Procedures

Reminders:
• Lab 1 due Thursday 9/23
• checkoff begins on Friday 9/17
  (signup via Doodle)
Components of a MicroProcessor

- Register File
  - x0
  - x1: 100110...0
  - x2
  -...
  - x31

- ALU

- Main Memory
  - Address:
    - 0
    - 4
    - 8
    - 12
    - 16
    - 20
  - Holds program and data

- Arithmetic Logic Unit
RISC-V Instruction Types

- **Computational Instructions executed by ALU**
  - Register-Register: \texttt{op dest, src1, src2}
  - Register-Immediate: \texttt{op dest, src1, const}

- **Control flow instructions**
  - Conditional: \texttt{br\_comp src1, src2, label}
  - Unconditional: \texttt{jal label} and \texttt{jalr register}

- **Loads and Stores**
  - \texttt{lw dest, offset(base)}
  - \texttt{sw src, offset(base)}
  - Base is a register, offset is a small constant

- **Pseudoinstructions**
  - Shorthand for other instructions
Performing Computations on Values in Memory

\[ a = b + c \]

b: \( x_1 \leftarrow \text{load}(\text{Mem}[0x4]) \)

c: \( x_2 \leftarrow \text{load}(\text{Mem}[0x8]) \)

\( x_3 \leftarrow x_1 + x_2 \)

a: \( \text{store}(\text{Mem}[0x10]) \leftarrow x_3 \)
RISC-V Load and Store Instructions

- Address is specified as a \(<\text{base address}, \text{offset}>\) pair;
  - base address is always stored in a register
  - the offset is encoded as a 12 bit constant in the instruction
  - Format: \(lw\) dest, offset(base) \quad sw\) src, offset(base)

- Assembly:
  - \(lw\) x1, 0x4(x0)
  - \(lw\) x2, 0x8(x0)
  - add x3, x1, x2
  - \(sw\) x3, 0x10(x0)

- Behavior:
  - \(x1 \leftarrow \text{load(Mem[x0 + 0x4])}\)
  - \(x2 \leftarrow \text{load(Mem[x0 + 0x8])}\)
  - \(x3 \leftarrow x1 + x2\)
  - \(\text{store(Mem[x0 + 0x10])} \leftarrow x3\)
Registers vs Memory

add x1, x2, x3
  x1 = 0x18

mv x4, x3
  x4 = 0x10

lw x5, 0(x3)
  x5 = 0x22

lw x6, 4(x3)
  x6 = 0x23

sw x6, 8(x3)

value of x6 (0x23) is written to M[0x10+0x8]
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 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

```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
slli x13, x10, 2
xor x12, x13, x11
```
Compiling Conditionals

- *if* statements can be compiled using branches:

  C code          | RISC-V Assembly
  ---             | ---
  `if (expr) {`   | (compile expr into xN)
  `  if-body`     | beqz xN, endif
  `  }`           | (compile if-body)
  `endif:`

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

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

  We can sometimes combine *expr* and the branch

  ```assembly
  // x: x10, y: x11
  slt x12, x10, x11
  beqz x12, endif
  sub x11, x11, x10
  endif:
  ```
### Compiling Conditionals

- **if-else statements are similar:**

  **C code**
  ```c
  if (expr) {
    if-body
  } else {
    else-body
  }
  ```

  **RISC-V Assembly**
  ```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
  while (expr) {
    while-body
  }
  ```

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

- **Can you write a version that executes fewer instructions?**
## Putting it all together

<table>
<thead>
<tr>
<th>C code</th>
<th>RISC-V Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>while (x != y) {</code></td>
<td>// x: x10, y: x11</td>
</tr>
<tr>
<td><code>if (x &gt; y) {</code></td>
<td>j compare</td>
</tr>
<tr>
<td><code>x = x - y;</code></td>
<td>loop:</td>
</tr>
<tr>
<td>} else {</td>
<td>(compile <code>while-body</code>)</td>
</tr>
<tr>
<td><code>y = y - x;</code></td>
<td>compare:</td>
</tr>
<tr>
<td>}</td>
<td>bne x10, x11, loop</td>
</tr>
</tbody>
</table>
while (x != y) {
    if (x > y) {
        x = x - y;
    } else {
        y = y - x;
    }
}

// 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 arguments
  - Local storage
  - Returns to the caller when finished

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

```cpp
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 Procedure: 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?
Arguments and return values

- A caller needs to pass arguments to the called procedure, as well as get results back from the called procedure
  - Both are done through registers

- A 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>
</tr>
</thead>
<tbody>
<tr>
<td>a0 to a7</td>
<td>x10 to x17</td>
<td>Function arguments</td>
</tr>
<tr>
<td>a0 and a1</td>
<td>x10 and x11</td>
<td>Function return values</td>
</tr>
</tbody>
</table>
A procedure can be called from many different places

- The caller can get to the called procedure code simply by executing an 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!

<table>
<thead>
<tr>
<th>Symbolic name</th>
<th>Registers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ra</td>
<td>x1</td>
<td>Return address</td>
</tr>
</tbody>
</table>

```
[0x100] j sum
[0x678] j sum
sum:
    j ?
```

September 16, 2021
MIT 6.004 Fall 2021
Procedure Linking

- How to transfer control to callee and back to caller?
  
  \texttt{proc\_call: jal ra, label}
  
  1. Stores address of \texttt{proc\_call + 4} in register \texttt{ra} (return address register)
  
  2. Jumps to instruction at address label where label is the name of the procedure
  
  3. After executing procedure, \texttt{jr ra} to return to caller and continue execution

\[ \begin{align*}
\text{...} & \quad \text{[0x100] jal ra, sum} \\
\text{...} & \quad \text{[0x678] jal ra, sum} \\
\text{...} & \end{align*} \]
Managing a procedure’s register space

- A caller uses the same register set as the called procedure
  - A caller should not rely on how the called procedure manages its register space
  - Ideally, procedure implementation should be able to use all registers
- Either the **caller** or the **callee** saves the caller’s registers in memory and restores them when the procedure call has completed execution
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 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>
Activation record and procedure calls

- An *Activation record* holds all storage needs of a 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
Thank you!

Next lecture:
Procedures, Stacks, and MMIO