Compiling Code, and Implementing Procedures

Silvina’s office hours:
• Held weekly in lab
• Send email to sign up
Unconditional Control Instructions: Jumps

- **jal**: Unconditional jump and **link**
  - Example: `jal x3, label`
  - Jump target specified as label
  - Label is encoded as an offset from current instruction
  - Link: is stored in x3

  ![20 bit immediate](image)

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 bit immediate</td>
<td>dest</td>
<td>1101111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **jalr**: Unconditional jump via register and **link**
  - Example: `jalr x3, 4(x1)`
  - Jump target specified as register value plus constant offset
  - Example: Jump target = `x1 + 4`
  - Can jump to **any 32 bit address** – supports long jumps
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 `<base address, 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)   sw src, offset(base)`

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

- Behavior:
  
  ```null
  x1 ← load(Mem[x0 + 0x4])
  x2 ← load(Mem[x0 + 0x8])
  x3 ← x1 + x2
  store(Mem[x0 + 0x10]) ← x3
  ```
Program to sum array elements

\[ \text{sum} = a[0] + a[1] + a[2] + \ldots + a[n-1] \]

(Assume 100 (address of base) already loaded into x10)

```
  lw x1, 0x0(x10)
  lw x2, 0x4(x10)
  add x3, x0, x0
  loop:
    lw x4, 0x0(x1)
    add x3, x3, x4
    addi x1, x1, 4
    addi x2, x2, -1
    bnez x2, loop
  sw x3, 0x8(x10)
```

Main Memory

<table>
<thead>
<tr>
<th>Addr (0x00)</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>a[0]</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>a[1]</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>a[n-1]</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>base</td>
</tr>
<tr>
<td>104</td>
<td>104</td>
<td>n</td>
</tr>
<tr>
<td>108</td>
<td>108</td>
<td>sum</td>
</tr>
</tbody>
</table>

Register File

- x1: Addr of a[i]
- x2: n
- x3: sum
- x10: 100
## Pseudoinstructions

- Aliases to other actual instructions to simplify assembly programming.

<table>
<thead>
<tr>
<th>Pseudoinstruction</th>
<th>Equivalent Assembly Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mv x2, x1</code></td>
<td><code>addi x2, x1, 0</code></td>
</tr>
<tr>
<td><code>ble x1, x2, label</code></td>
<td><code>bge x2, x1, label</code></td>
</tr>
<tr>
<td><code>li x2, 3</code></td>
<td><code>addi x2, x0, 3</code></td>
</tr>
<tr>
<td><code>li x3, 0x4321</code></td>
<td><code>lui x3, 0x4</code></td>
</tr>
<tr>
<td></td>
<td><code>addi x3, x3, 0x321</code></td>
</tr>
</tbody>
</table>
Registers vs Memory

add x1, x2, x3
   x1 = 0x1C

mv x4, x3
   x4 = 0x14

lw x5, 0(x3)
   x5 = 0x23

lw x6, 8(x3)
   x6 = 0x16

sw x6, 0xC(x3)

value of x6 (0x16) is written to M[0x14+0xC]
Compiling Simple Expressions

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

\textbf{Example C code}

\begin{verbatim}
int x, y, z;
...

y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
\end{verbatim}

\textbf{RISC-V Assembly}

\begin{verbatim}
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
\texttt{li} x14, 123456
add x14, x11, x14
or x11, x13, x14
slli x13, x10, 2
xor x12, x13, x11
\end{verbatim}
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

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

  We can sometimes combine `expr` and the branch

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

  ```
  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:
    ```
Loops can be compiled using *backward* branches:

C code

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

RISC-V Assembly

```asm
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

```asm
// 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:
    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

```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
```
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>
Calling procedures

- 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

```
... [0x100] j sum
... [0x678] j sum

sum:
... j ?
0x104?
0x67C?
```

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

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

  **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, jr ra to return to caller and continue execution

  ![Diagram](image)

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

  
  \[
  \begin{align*}
  \text{ra} = 0x104 & \quad \text{sum:} \\
  \text{ra} = 0x67C & \quad \text{jr ra}
  \end{align*}
  \]

  
  
  \[
  \begin{align*}
  1^{\text{st}} \text{ time: & jump to 0x104} \\
  2^{\text{nd}} \text{ time: & jump to 0x67C}
  \end{align*}
  \]
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
### 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>
<tr>
<td>zero</td>
<td>x0</td>
<td>Hardwired zero</td>
<td>---</td>
</tr>
</tbody>
</table>
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 register values according to the calling convention (e.g., s registers that procedure will overwrite)

Each procedure call has its own instance of local storage known as the procedure’s activation record.
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
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)
Thank you!

Next lecture:
Procedures, Stacks, and MMIO