Introduction to Assembly and RISC-V

Reminders:
- Lab 1 released today
- Lab hours begin today
- Sign up for piazza
“General Purpose” Processor

- It would be highly desirable if the same hardware could execute programs written in Python, Java, C, or any high-level language
“General Purpose” Processor

- It would be highly desirable if the same hardware could execute programs written in Python, Java, C, or any high-level language.
- It is also not sensible to execute every feature of a high-level language directly in hardware.
“General Purpose” Processor

- It would be highly desirable if the same hardware could execute programs written in Python, Java, C, or any high-level language.
- It is also not sensible to execute every feature of a high-level language directly in hardware.

Diagram:

```
Python  Java  C  Scheme ....
```

Machine (Assembly) Language

```
\downarrow
```

Microprocessor
“General Purpose” Processor

- It would be highly desirable if the same hardware could execute programs written in Python, Java, C, or any high-level language.
- It is also not sensible to execute every feature of a high-level language directly in hardware.

Diagram:

```
Python   Java    C    Scheme .....  
|         |         |     |                      |
|__________|__________|______|----------------------|
|          |          |      | Machine (Assembly) Language |
|          |          |      | HW-SW interface       |
|          |          |      | Microprocessor         |
```
“General Purpose” Processor

- It would be highly desirable if the same hardware could execute programs written in Python, Java, C, or any high-level language.
- It is also not sensible to execute every feature of a high-level language directly in hardware.

Python   Java   C   Scheme .....  

Machine (Assembly) Language  

Direct Hardware Execution  

HW-SW interface
“General Purpose” Processor

- It would be highly desirable if the same hardware could execute programs written in Python, Java, C, or any high-level language.
- It is also not sensible to execute every feature of a high-level language directly in hardware.

Python    Java    C    Scheme ......

Software Translation

Machine (Assembly) Language

HW-SW interface

Direct Hardware Execution

Microprocessor
Components of a MicroProcessor

Register File

\[ \begin{array}{|c|}
\hline
x0 & 100110\ldots0 \\
\hline
x1 & \text{32-bit “words”} \\
\hline
x2 & \\
\hline
\vdots & \\
\hline
x31 & \\
\hline
\end{array} \]
Components of a MicroProcessor

Register File

\[ \begin{array}{c}
\text{x0} \\
\text{x1} \\
\text{x2} \\
\vdots \\
\text{x31}
\end{array} \]

100110...0

32-bit “words”

Arithmetic Logic Unit
Components of a MicroProcessor

Register File

- x0
- x1: 100110....0
- x2
- ... 32-bit “words”
- x31

Arithmetic Logic Unit
Components of a MicroProcessor

- **Register File**
  - x0
  - x1
  - x2
  - ...
  - x31
  - 32-bit “words”

- **ALU**
Components of a MicroProcessor

- **ALU**
- **Register File**
  - x0
  - x1
  - x2
  - ...
  - x31
  - 32-bit “words”

- **Main Memory**
  - 32-bit “words”
  - Address
    - 0
    - 4
    - 8
    - 12
    - 16
    - 20
    - 31
    - 0
Components of a MicroProcessor

- **Register File**
  - x0, x1, x2, ..., x31
  - Contains 32-bit "words"

- **Arithmetic Logic Unit (ALU)**

- **Main Memory**
  - Holds program and data
  - Address: 0, 4, 8, 12, 16, 20
  - 32-bit "words"
Components of a MicroProcessor

- **Register File**
  - **x0**
  - **x1**
  - **x2**
  - **x31**
  - 32-bit “words”

- **ALU**

- **Main Memory**
  - Address
  - 0
  - 4
  - 8
  - 12
  - 16
  - 20
  - 32-bit “words”

- **Holds program and data**

September 10, 2019
Components of a MicroProcessor

- **ALU**
- **Register File**
- **Main Memory**

**Register File**
- Contains 32-bit "words"
- Registers: x0, x1, x2, ..., x31
  - x0: 100110....0

**Main Memory**
- 32-bit "words"
- Address: 0, 4, 8, 12, 16, 20, 31

**ALU**
- Holds program and data

*Machine language directly reflects this structure*
Each register is of fixed size, say 32 bits
MicroProcessor Structure / Assembly Language

- Each register is of fixed size, say 32 bits
- The number of registers are small, say 32
MicroProcessor Structure / Assembly Language

- Each register is of fixed size, say 32 bits
- The number of registers are small, say 32
- ALU directly performs operations on the register file, typically
  - \( x_i \leftarrow \text{Op}(x_j, x_k) \) where \( \text{Op} \in \{+, \text{AND}, \text{OR}, <, >, \ldots\} \)
MicroProcessor Structure / Assembly Language

- Each register is of fixed size, say 32 bits
- The number of registers are small, say 32
- ALU directly performs operations on the register file, typically
  - \( x_i \leftarrow \text{Op}(x_j, x_k) \) where \( \text{Op} \in \{+, \text{AND}, \text{OR}, <, >, \ldots\} \)
- Memory is large, say Giga bytes, and holds program and data
MicroProcessor Structure / Assembly Language

- Each register is of fixed size, say 32 bits
- The number of registers are small, say 32
- ALU directly performs operations on the register file, typically
  - $x_i \leftarrow \text{Op}( x_j, x_k )$ where $\text{Op} \in \{+\text{, AND, OR, }<\text{, }>,\ldots\}$
- Memory is large, say Giga bytes, and holds program and data
- Data can be moved back and forth between Memory and Register File
  - Ld $x$ M[addr]
  - St M[addr] $x$
Assembly (Machine) Language Program

- An assembly language program is a sequence of instructions which execute in a sequential order unless a control transfer instruction is executed.
An assembly language program is a sequence of instructions which execute in a sequential order unless a control transfer instruction is executed.

Each instruction specifies one of the following operations:

- ALU or Reg-to-Reg operation
- Ld
- St
- Control transfer operation: e.g., if $x_i < x_j$ go to label l
Program to sum array elements

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

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

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

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

x1    load(base)
x2    load(n)
x3    0
Program to sum array elements

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

x1  load(base)
x2  load(n)
x3  0

loop:
x4  load(Mem[x1])
add  x3, x3, x4
addi  x1, x1, 4
addi  x2, x2, -1
bnez  x2, loop
Program to sum array elements

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

1. \( x1 \) load(base)
2. \( x2 \) load(n)
3. \( x3 \) 0

**loop:**

4. \( x4 \) load(Mem[x1])
5. add \( x3, x3, x4 \)
6. addi \( x1, x1, 4 \)
7. addi \( x2, x2, -1 \)
8. bnez \( x2, \text{loop} \)

store(sum) \( x3 \)
High Level vs Assembly Language

High Level Language

Assembly Language
## High Level vs Assembly Language

<table>
<thead>
<tr>
<th>High Level Language</th>
<th>Assembly Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Primitive Arithmetic and logical operations</td>
<td></td>
</tr>
</tbody>
</table>
## High Level vs Assembly Language

<table>
<thead>
<tr>
<th>High Level Language</th>
<th>Assembly Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Primitive Arithmetic and logical operations</td>
<td>1. Primitive Arithmetic and logical operations</td>
</tr>
</tbody>
</table>
# High Level vs Assembly Language

## High Level Language

1. Primitive Arithmetic and logical operations
2. Complex data types and data structures

## Assembly Language

1. Primitive Arithmetic and logical operations
# High Level vs Assembly Language

<table>
<thead>
<tr>
<th>High Level Language</th>
<th>Assembly Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Primitive Arithmetic and logical operations</td>
<td>1. Primitive Arithmetic and logical operations</td>
</tr>
<tr>
<td>2. Complex data types and data structures</td>
<td>2. Primitive data structures – bits and integers</td>
</tr>
</tbody>
</table>
# High Level vs Assembly Language

<table>
<thead>
<tr>
<th>High Level Language</th>
<th>Assembly Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Primitive Arithmetic and logical operations</td>
<td>1. Primitive Arithmetic and logical operations</td>
</tr>
<tr>
<td>2. Complex data types and data structures</td>
<td>2. Primitive data structures - bits and integers</td>
</tr>
<tr>
<td>3. Complex control structures - Conditional statements, loops and procedures</td>
<td></td>
</tr>
</tbody>
</table>

September 10, 2019
# High Level vs Assembly Language

<table>
<thead>
<tr>
<th>High Level Language</th>
<th>Assembly Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Primitive Arithmetic and logical operations</td>
<td>1. Primitive Arithmetic and logical operations</td>
</tr>
<tr>
<td>2. Complex data types and data structures</td>
<td>2. Primitive data structures – bits and integers</td>
</tr>
<tr>
<td>3. Complex control structures - Conditional statements, loops and procedures</td>
<td>3. Control transfer instructions</td>
</tr>
</tbody>
</table>
## High Level vs Assembly Language

<table>
<thead>
<tr>
<th>High Level Language</th>
<th>Assembly Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Primitive Arithmetic and logical operations</td>
<td>1. Primitive Arithmetic and logical operations</td>
</tr>
<tr>
<td>2. Complex data types and data structures</td>
<td>2. Primitive data structures – bits and integers</td>
</tr>
<tr>
<td>3. Complex control structures - Conditional statements, loops and procedures</td>
<td>3. Control transfer instructions</td>
</tr>
<tr>
<td>4. Not suitable for direct implementation in hardware</td>
<td></td>
</tr>
</tbody>
</table>
# High Level vs Assembly Language

<table>
<thead>
<tr>
<th>High Level Language</th>
<th>Assembly Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Primitive Arithmetic and logical operations</td>
<td>1. Primitive Arithmetic and logical operations</td>
</tr>
<tr>
<td>2. Complex data types and data structures</td>
<td>2. Primitive data structures – bits and integers</td>
</tr>
<tr>
<td>3. Complex control structures - Conditional statements, loops and procedures</td>
<td>3. Control transfer instructions</td>
</tr>
<tr>
<td>4. Not suitable for direct implementation in hardware</td>
<td>4. Designed to be directly implementable in hardware</td>
</tr>
</tbody>
</table>
High Level vs Assembly Language

**High Level Language**

1. Primitive Arithmetic and logical operations
2. Complex data types and data structures
3. Complex control structures - Conditional statements, loops and procedures
4. Not suitable for direct implementation in hardware

**Assembly Language**

1. Primitive Arithmetic and logical operations
2. Primitive data structures – bits and integers
3. Control transfer instructions
4. Designed to be directly implementable in hardware

*tedious programming!*
Instruction Set Architecture (ISA)

- ISA: The contract between software and hardware
  - Functional definition of operations and storage locations
  - Precise description of how software can invoke and access them
Instruction Set Architecture (ISA)

- ISA: The contract between software and hardware
  - Functional definition of operations and storage locations
  - Precise description of how software can invoke and access them

- RISC-V ISA:
  - A new, open, free ISA from Berkeley
  - Several variants
    - RV32, RV64, RV128: Different data widths
    - ‘I’: Base Integer instructions
    - ‘M’: Multiply and Divide
    - ‘F’ and ‘D’: Single- and Double-precision floating point
    - And many other modular extensions
Instruction Set Architecture (ISA)

- ISA: The contract between software and hardware
  - Functional definition of operations and storage locations
  - Precise description of how software can invoke and access them

- RISC-V ISA:
  - A new, open, free ISA from Berkeley
  - Several variants
    - RV32, RV64, RV128: Different data widths
    - ‘I’: Base Integer instructions
    - ‘M’: Multiply and Divide
    - ‘F’ and ‘D’: Single- and Double-precision floating point
    - And many other modular extensions

- We will design an RV32I processor, which is the base integer 32-bit variant
RISC-V Processor Storage

Register File

x0
x1
x2

::

x31

32-bit "words"

Main Memory

Address

0
4
8
12
16
20

31
2
1
0

32-bit "words"

(4 bytes)
RISC-V Processor Storage

**Registers:**
- 32 General Purpose Registers
- Each register is 32 bits wide

**Register File**
- x0
- x1
- x2
- ...
- x31

**Main Memory**
- Address
- 32-bit “words” (4 bytes)

- 32 General Purpose Registers
- Each register is 32 bits wide
RISC-V Processor Storage

### Registers:
- 32 General Purpose Registers
- Each register is 32 bits wide
- \(x_0 = 0\)

### Address Registers:
- 32-bit "words"
- (4 bytes)

### Main Memory
- \(0\) to \(31\)

### Register File
- \(x_0\) hardwired to 0
RISC-V Processor Storage

**Register File**: 32 General Purpose Registers
- Each register is 32 bits wide
- x0 = 0

**Main Memory**: Each memory location is 32 bits wide (1 word)
- Instructions and data

32-bit “words” (4 bytes)
RISC-V Processor Storage

Registers:
- 32 General Purpose Registers
- Each register is 32 bits wide
- \( x_0 = 0 \)

Memory:
- Each memory location is 32 bits wide (1 word)
  - Instructions and data
- Memory is byte (8 bits) addressable
- Address of adjacent words are 4 apart.

Register File

- \( x_0 \) hardwired to 0

Main Memory

- 32-bit “words”
- 32-bit “words” (4 bytes)
RISC-V Processor Storage

**Registers:**
- 32 General Purpose Registers
- Each register is 32 bits wide
- x0 = 0

**Memory:**
- Each memory location is 32 bits wide (1 word)
  - Instructions and data
- Memory is byte (8 bits) addressable
- Address of adjacent words are 4 apart.
- Address is 32 bits
- Can address $2^{32}$ bytes or $2^{30}$ words.

Register File
- x0 hardwired to 0

Main Memory
- 32-bit "words"
  - (4 bytes)
- 32 General Purpose Registers
- Each register is 32 bits wide
- x0 = 0

Address
- 32-bit "words"
RISC-V ISA: Instructions

- Three types of operations:
  - **Computational**: Perform arithmetic and logical operations on registers
  - **Loads and stores**: Move data between registers and main memory
  - **Control Flow**: Change the execution order of instructions to support conditional statements and loops.
Computational Instructions

- Arithmetic, comparison, logical, and shift operations.
Computational Instructions

- Arithmetic, comparison, logical, and shift operations.
  - Register-Register Instructions:
    - 2 source operand registers
    - 1 destination register
Computational Instructions

- Arithmetic, comparison, logical, and shift operations.
  - Register-Register Instructions:
    - 2 source operand registers
    - 1 destination register

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Comparisons</th>
<th>Logical</th>
<th>Shifts</th>
</tr>
</thead>
<tbody>
<tr>
<td>add, sub</td>
<td>slt, sltu</td>
<td>and, or, xor</td>
<td>sll, srl, sra</td>
</tr>
</tbody>
</table>
Computational Instructions

- Arithmetic, comparison, logical, and shift operations.
  - Register-Register Instructions:
    - 2 source operand registers
    - 1 destination register

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Comparisons</th>
<th>Logical</th>
<th>Shifts</th>
</tr>
</thead>
<tbody>
<tr>
<td>add, sub</td>
<td>slt, sltu</td>
<td>and, or, xor</td>
<td>sll, srl, sra</td>
</tr>
</tbody>
</table>

- Format: oper dest, src1, src2
Computational Instructions

- Arithmetic, comparison, logical, and shift operations.
  - Register-Register Instructions:
    - 2 source operand registers
    - 1 destination register

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Comparisons</th>
<th>Logical</th>
<th>Shifts</th>
</tr>
</thead>
<tbody>
<tr>
<td>add, sub</td>
<td>slt, sltu</td>
<td>and, or, xor</td>
<td>sll, srl, sra</td>
</tr>
</tbody>
</table>

- **Format**: oper dest, src1, src2
- add x3, x1, x2  
  - x3 ← x1 + x2
Computational Instructions

- Arithmetic, comparison, logical, and shift operations.
  - Register-Register Instructions:
    - 2 source operand registers
    - 1 destination register

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Comparisons</th>
<th>Logical</th>
<th>Shifts</th>
</tr>
</thead>
<tbody>
<tr>
<td>add, sub</td>
<td>slt, sltu</td>
<td>and, or, xor</td>
<td>s1l, srl, sra</td>
</tr>
</tbody>
</table>

- Format: oper dest, src1, src2

- add x3, x1, x2
- slt x3, x1, x2
- x3 \(\leftarrow\) x1 + x2
- If x1 < x2 then x3 = 1 else x3 = 0
Computational Instructions

- Arithmetic, comparison, logical, and shift operations.
  - Register-Register Instructions:
    - 2 source operand registers
    - 1 destination register

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Comparisons</th>
<th>Logical</th>
<th>Shifts</th>
</tr>
</thead>
<tbody>
<tr>
<td>add, sub</td>
<td>slt, sltu</td>
<td>and, or, xor</td>
<td>sll, srl, sra</td>
</tr>
</tbody>
</table>

- **Format:** oper dest, src1, src2

- add x3, x1, x2
- slt x3, x1, x2
- and x3, x1, x2
- x3 ← x1 + x2
- If x1 < x2 then x3 = 1 else x3 = 0
- x3 ← x1 & x2
Computational Instructions

- Arithmetic, comparison, logical, and shift operations.
  - Register-Register Instructions:
    - 2 source operand registers
    - 1 destination register

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Comparisons</th>
<th>Logical</th>
<th>Shifts</th>
</tr>
</thead>
<tbody>
<tr>
<td>add, sub</td>
<td>slt, sltu</td>
<td>and, or, xor</td>
<td>sl1, srl, sra</td>
</tr>
</tbody>
</table>

- **Format**: oper dest, src1, src2

- add x3, x1, x2
- slt x3, x1, x2
- and x3, x1, x2
- sll x3, x1, x2
- x3 ← x1 + x2
- If x1 < x2 then x3 = 1 else x3 = 0
- x3 ← x1 & x2
- x3 ← x1 << x2
All Values are Binary

- Suppose: $x_1 = 00101; x_2 = 00011$
  - add $x_3, x_1, x_2$
All Values are Binary

- Suppose: \( x_1 = 00101; x_2 = 00011 \)
  - add \( x_3, x_1, x_2 \)

Base 10

\[
\begin{align*}
5 & \\
+ 3 & \\
\hline \\
8
\end{align*}
\]
All Values are Binary

- Suppose: $x_1 = 00101; x_2 = 00011$
  - add $x_3, x_1, x_2$

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>00101</td>
</tr>
<tr>
<td>+ 3</td>
<td>+ 00011</td>
</tr>
<tr>
<td>8</td>
<td>00101</td>
</tr>
</tbody>
</table>
Suppose: \( x_1 = 00101; x_2 = 00011 \)

- add \( x_3, x_1, x_2 \)

\[
\begin{array}{c c c c c}
\text{Base 10} & & & & \\
5 & + & 3 & & \\
\hline
8 & & & & \\
\end{array}
\quad
\begin{array}{c c c c c}
\text{Base 2} & & & & \\
1 & & & & \\
\hline
00101 & + & 00011 & & \\
\hline
0 & & & & \\
\end{array}
\]
All Values are Binary

- Suppose: \( x_1 = 00101; x_2 = 00011 \)
  - add \( x_3, x_1, x_2 \)

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>+ 3</td>
<td>00101</td>
</tr>
<tr>
<td>8</td>
<td>00</td>
</tr>
</tbody>
</table>

September 10, 2019  MIT 6.004 Fall 2019  L02-12
All Values are Binary

- Suppose: $x_1 = 00101; x_2 = 00011$
  - add $x_3, x_1, x_2$

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>111</td>
</tr>
<tr>
<td>+ 3</td>
<td>00101</td>
</tr>
<tr>
<td>8</td>
<td>00011</td>
</tr>
<tr>
<td></td>
<td>000</td>
</tr>
</tbody>
</table>
All Values are Binary

- Suppose: $x_1 = 00101; x_2 = 00011$
  - add $x_3, x_1, x_2$

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>111</td>
</tr>
<tr>
<td>+ 3</td>
<td>00101</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>00011</td>
</tr>
</tbody>
</table>
All Values are Binary

- Suppose: $x_1 = 00101$; $x_2 = 00011$
  - add $x_3$, $x_1$, $x_2$

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>111</td>
</tr>
<tr>
<td>+ 3</td>
<td>00101</td>
</tr>
<tr>
<td>8</td>
<td>01000</td>
</tr>
</tbody>
</table>

$5 + 3 + 00011 = 01000$
All Values are Binary

- Suppose: \( x_1 = 00101; x_2 = 00011 \)
  - add \( x_3, x_1, x_2 \)
    
    | Base 10       | Base 2       |
    |---------------|--------------|
    | 5            | 111          |
    | + 3          | 00101        |
    | **8**        | **01000**    |

- \textit{sll} \( x_3, x_1, x_2 \)
  - Shift \( x_1 \) left by \( x_2 \) bits
  - \text{00101}
All Values are Binary

- Suppose: $x_1 = 00101; x_2 = 00011$
  - add $x_3, x_1, x_2$
    
    \[
    \begin{array}{c@{}c@{}c@{}c@{}c@{}c}
    & 1 & 0 & 1 & 0 & 1 \\
    + & 0 & 0 & 0 & 1 & 1 \\
    \hline
    0 & 1 & 0 & 0 & 0 & 0
    \end{array}
    \]

    - $sll x_3, x_1, x_2$
      
      \[
      \begin{array}{c@{}c@{}c@{}c@{}c@{}c}
      & 0 & 0 & 1 & 0 & 1 \\
      \uparrow & 0 & 1 & 0 & 1 & 0 \\
      \downarrow & 0 & 1 & 0 & 1 & 0 \\
    \end{array}
    \]
All Values are Binary

- Suppose: \( x_1 = 00101; x_2 = 00011 \)
  - add \( x_3, x_1, x_2 \)

  \[
  \begin{array}{c}
  \text{Base 10} \\
  5 \\
  + 3 \\
  \hline \\
  8 \\
  \end{array}
  \begin{array}{c}
  \text{Base 2} \\
  111 \\
  + 00111 \\
  \hline \\
  01000 \\
  \end{array}
  \]

- \text{sll } x_3, x_1, x_2
  - Shift \( x_1 \) left by \( x_2 \) bits

  \[
  00101 \quad 010100 \\
  101000 \\
  \]
All Values are Binary

- Suppose: \( x_1 = 00101; x_2 = 00011 \)
  - add \( x_3, x_1, x_2 \)

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 5 )</td>
<td>( 111 )</td>
</tr>
<tr>
<td>( + 3 )</td>
<td>( 00101 )</td>
</tr>
<tr>
<td>( 8 )</td>
<td>( 00011 )</td>
</tr>
</tbody>
</table>

- \( \text{sll } x_3, x_1, x_2 \)
  - Shift \( x_1 \) left by \( x_2 \) bits
  - \( 00101 \)
  - \( 01010 \)
  - \( 10100 \)
  - \( 01000 \)
All Values are Binary

Suppose: \( x_1 = 00101; x_2 = 00011 \)

- add \( x_3, x_1, x_2 \)

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>111</td>
</tr>
<tr>
<td>+ 3</td>
<td>00101</td>
</tr>
<tr>
<td>8</td>
<td>01000</td>
</tr>
</tbody>
</table>

- \( \text{sll } x_3, x_1, x_2 \)
  - Shift \( x_1 \) left by \( x_2 \) bits
  - Notice fixed width

\[
\begin{array}{c}
00101 \\
01010 \\
01000 \\
01000
\end{array}
\]
Register-Immediate Instructions

- One operand comes from a register and the other is a small constant that is encoded into the instruction.
Register–Immediate Instructions

- One operand comes from a register and the other is a small constant that is encoded into the instruction.
  - **Format:** oper dest, src1, const
Register-Immediate Instructions

- One operand comes from a register and the other is a small constant that is encoded into the instruction.
  
  **Format:** oper dest, src1, const
  
  - addi x3, x1, 3
  - andi x3, x1, 3
  - slli x3, x1, 3
    
    - x3 ← x1 + 3
    - x3 ← x1 & 3
    - x3 ← x1 << 3
Register-Immediate Instructions

- One operand comes from a register and the other is a small constant that is encoded into the instruction.
  - **Format**: oper dest, src1, const
    - `addi x3, x1, 3`  
    - `andi x3, x1, 3`  
    - `slli x3, x1, 3`  
      - `x3 ← x1 + 3`  
      - `x3 ← x1 & 3`  
      - `x3 ← x1 << 3`
Register-Immediate Instructions

- One operand comes from a register and the other is a small constant that is encoded into the instruction.
  - **Format**: `oper dest, src1, const`
    - `addi x3, x1, 3`
    - `andi x3, x1, 3`
    - `slli x3, x1, 3`
    - `x3 ← x1 + 3`
    - `x3 ← x1 & 3`
    - `x3 ← x1 << 3`

<table>
<thead>
<tr>
<th>Format</th>
<th>Arithmetic</th>
<th>Comparisons</th>
<th>Logical</th>
<th>Shifts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-Register</td>
<td>add, sub</td>
<td>slt, sltu</td>
<td>and, or, xor</td>
<td>sll, srl, sra</td>
</tr>
<tr>
<td>Register-Immediate</td>
<td>addi</td>
<td>slti, sltiu</td>
<td>andi, ori, xor</td>
<td>slli, srli, srai</td>
</tr>
</tbody>
</table>

- No `subi`, instead use negative constant.
  - `addi x3, x1, -3`
  - `x3 ← x1 - 3`
Compound Computation

- Execute \( a = ((b+3) \gg c) - 1; \)
Compound Computation

- Execute \( a = ((b+3) \gg c) - 1; \)

1. Break up complex expression into basic computations.
Compound Computation

- Execute \( a = ((b+3) \gg c) - 1; \)

1. Break up complex expression into **basic computations**.
   - Our instructions can only specify two source operands and one destination operand (also known as three address instruction)
Compound Computation

- Execute \( a = ((b+3) \gg c) - 1; \)

1. Break up complex expression into basic computations.
   - Our instructions can only specify two source operands and one destination operand (also known as three address instruction)

```plaintext
t0 = b + 3;
t1 = t0 \gg c;
a = t1 - 1;
```
Compound Computation

- Execute \( a = ((b+3) \gg c) - 1; \)

1. Break up complex expression into basic computations.
   - Our instructions can only specify two source operands and one destination operand (also known as three address instruction)

2. Assume \( a, b, c \) are in registers \( x1, x2, \) and \( x3 \) respectively. Use \( x4 \) for \( t0 \), and \( x5 \) for \( t1 \).

\[
\begin{align*}
    t0 &= b + 3; \\
    t1 &= t0 \gg c; \\
    a &= t1 - 1;
\end{align*}
\]
Compound Computation

- Execute $a = ((b+3) \gg c) - 1$;

1. Break up complex expression into basic computations.
   - Our instructions can only specify two source operands and one destination operand (also known as three address instruction)

2. Assume $a$, $b$, $c$ are in registers $x1$, $x2$, and $x3$ respectively. Use $x4$ for $t0$, and $x5$ for $t1$.

   
   $t0 = b + 3$

   $t1 = t0 \gg c$

   $a = t1 - 1$

   
   | addi $x4$, $x2$, 3 |
   | srl $x5$, $x4$, $x3$ |
   | addi $x1$, $x5$, -1 |
Control Flow Instructions

- Execute `if (a < b):  c = a + 1
  else:  c = b + 2`
Control Flow Instructions

- Execute \( \text{if } (a < b): \quad c = a + 1 \)
  \( \text{else: } \quad c = b + 2 \)

- Need Conditional branch instructions:
  - Format: \text{comp src1, src2, label}
Control Flow Instructions

- Execute \texttt{if (a < b): c = a + 1}
  \texttt{else: c = b + 2}

- Need Conditional branch instructions:
  - Format: \texttt{comp src1, src2, label}
  - First performs comparison to determine if branch is taken or not: \texttt{src1 comp src2}
Control Flow Instructions

- Execute \( \text{if } (a < b): \quad c = a + 1 \)
  \( \text{else: } \quad c = b + 2 \)

- Need Conditional branch instructions:
  - Format: \text{comp src1, src2, label}
  - First performs comparison to determine if branch is taken or not: \text{src1 comp src2}
  - If comparison returns True, then branch is taken, else continue executing program in order.
Control Flow Instructions

- Execute if \(a < b\): \(c = a + 1\) 
  else: \(c = b + 2\)

- Need Conditional branch instructions:
  - Format: \(comp\ src1, src2, label\)
  - First performs comparison to determine if branch is taken or not: \(src1 comp src2\)
  - If comparison returns True, then branch is taken, else continue executing program in order.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>beq</th>
<th>bne</th>
<th>blt</th>
<th>bge</th>
<th>bltu</th>
<th>bgeu</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>comp</em></td>
<td>==</td>
<td>!=</td>
<td>&lt;</td>
<td>≥</td>
<td>&lt;</td>
<td>≥</td>
</tr>
</tbody>
</table>
Control Flow Instructions

- **Execute**
  
  ```
  if (a < b):
      c = a + 1
  else:
      c = b + 2
  ```

- **Need Conditional branch instructions:**
  - **Format:** `comp src1, src2, label`
  - First performs comparison to determine if branch is taken or not: `src1 comp src2`
  - If comparison returns True, then branch is taken, else continue executing program in order.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>beq</th>
<th>bne</th>
<th>blt</th>
<th>bge</th>
<th>bltu</th>
<th>bgeu</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>comp</strong></td>
<td>==</td>
<td>!=</td>
<td>&lt;</td>
<td>≥</td>
<td>&lt;</td>
<td>≥</td>
</tr>
</tbody>
</table>

- Assume
  
  ```
  assume
  x1=a; x2=b; x3=c;
  ```
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 (To be discussed next lecture): is stored in x3
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 (To be discussed next lecture): is stored in x3

- **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
Constants and Instruction Encoding

Limitations

- Instructions are encoded as 32 bits.
  - Need to specify operation (10 bits)
  - Need to specify 2 source registers (10 bits) or 1 source register (5 bits) plus a small constant.
  - Need to specify 1 destination register (5 bits).
Constants and Instruction Encoding

Limitations

- Instructions are encoded as 32 bits.
  - Need to specify operation (10 bits)
  - Need to specify 2 source registers (10 bits) or 1 source register (5 bits) plus a **small** constant.
  - Need to specify 1 destination register (5 bits).

- The constant in register-immediate instructions has to be smaller than 12 bits; bigger constants have to be stored in the memory or a register and then used explicitly.
Constants and Instruction Encoding Limitations

- Instructions are encoded as 32 bits.
  - Need to specify operation (10 bits)
  - Need to specify 2 source registers (10 bits) or 1 source register (5 bits) plus a small constant.
  - Need to specify 1 destination register (5 bits).

- The constant in register-immediate instructions has to be smaller than 12 bits; bigger constants have to be stored in the memory or a register and then used explicitly.

- The constant in a jal instruction is 20 bits wide (7 bits for operation, and 5 bits for register).
Performing Computations on Values in Memory

\[ a = b + c \]
Performing Computations on Values in Memory

\[ a = b + c \]

\[ x_1 \leftarrow \text{load}(\text{Mem}[b]) \]
\[ x_2 \leftarrow \text{load}(\text{Mem}[c]) \]
\[ x_3 \leftarrow x_1 + x_2 \]
\[ \text{store}(\text{Mem}[a]) \leftarrow x_3 \]
Performing Computations on Values in Memory

\[ a = b + c \]

\[ x_1 \leftarrow \text{load(Mem}[b]\text{}] \]
\[ x_2 \leftarrow \text{load(Mem}[c]\text{}] \]
\[ x_3 \leftarrow x_1 + x_2 \]
\[ \text{store(Mem}[a]\text{])} \leftarrow x_3 \]

\[ x_1 \leftarrow \text{load}(0x4) \]
\[ x_2 \leftarrow \text{load}(0x8) \]
\[ x_3 \leftarrow x_1 + x_2 \]
\[ \text{store}(0x10) \leftarrow x_3 \]
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 specified as a small constant
  - Format: lw dest, offset(base)     sw src, offset(base)
# 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 specified as a small constant
  - Format: `lw dest, offset(base)`  `sw src, offset(base)`

## Assembly:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw</code></td>
<td><code>lw x1, 0x4(x0)</code></td>
</tr>
<tr>
<td><code>lw</code></td>
<td><code>lw x2, 0x8(x0)</code></td>
</tr>
<tr>
<td><code>add</code></td>
<td><code>add x3, x1, x2</code></td>
</tr>
<tr>
<td><code>sw</code></td>
<td><code>sw x3, 0x10(x0)</code></td>
</tr>
</tbody>
</table>

## Behavior:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x1</code></td>
<td><code>x1 ← load(Mem[x0 + 0x4])</code></td>
</tr>
<tr>
<td><code>x2</code></td>
<td><code>x2 ← load(Mem[x0 + 0x8])</code></td>
</tr>
<tr>
<td><code>x3</code></td>
<td><code>x3 ← x1 + x2</code></td>
</tr>
<tr>
<td><code>store</code></td>
<td><code>store(Mem[x0 + 0x10]) ← x3</code></td>
</tr>
</tbody>
</table>
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)
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)
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)

\[ \text{lw} \ x1, 0x0(x10) \]
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)

\[
\text{lw } x1, \ 0x0(x10) \\
\text{lw } x2, \ 0x4(x10)
\]
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)

\begin{align*}
lw \ x1, \ 0x0(x10) \\
lw \ x2, \ 0x4(x10) \\
add \ x3, \ x0, \ x0
\end{align*}
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)

\begin{verbatim}
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

\end{verbatim}
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)

\[
\begin{align*}
lw & \quad x1, 0x0(x10) \\
lw & \quad x2, 0x4(x10) \\
add & \quad x3, x0, x0 \\
\text{loop:} & \\
lw & \quad x4, 0x0(x1) \\
add & \quad x3, x3, x4 \\
addi & \quad x1, x1, 4 \\
addi & \quad x2, x2, -1 \\
bnez & \quad x2, \text{loop} \\
sw & \quad x3, 0x8(x10)
\end{align*}
\]
Pseudoinstructions

- Aliases to other actual instructions to simplify assembly programming.
Pseudoinstructions

- Aliases to other actual instructions to simplify assembly programming.

Pseudoinstruction: \texttt{mv x2, x1}  
Equivalent Assembly Instruction: \texttt{addi x2, x1, 0}
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>mv x2, x1</td>
<td>addi x2, x1, 0</td>
</tr>
<tr>
<td>li x2, 3</td>
<td>addi x2, x0, 3</td>
</tr>
</tbody>
</table>
Pseudoinstructions

- Aliases to other actual instructions to simplify assembly programming.

Pseudoinstruction:  
mv x2, x1  
li x2, 3  
ble x1, x2, label

Equivalent Assembly Instruction:  
addi x2, x1, 0  
addi x2, x0, 3  
bge x2, x1, label
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>mv x2, x1</td>
<td>addi x2, x1, 0</td>
</tr>
<tr>
<td>li x2, 3</td>
<td>addi x2, x0, 3</td>
</tr>
<tr>
<td>ble x1, x2, label</td>
<td>bge x2, x1, label</td>
</tr>
<tr>
<td>beqz x1, label</td>
<td>beq x1, x0, label</td>
</tr>
<tr>
<td>bnez x1, label</td>
<td>bne x1, x0, label</td>
</tr>
<tr>
<td>j label</td>
<td>jal x0, label</td>
</tr>
</tbody>
</table>

September 10, 2019
Thank you!

Next lecture:
Implementing Procedures in Assembly