Introduction to Assembly: RISC-V Instruction Set Architecture
Great Idea #1: Abstraction
Levels of Representation/Interpretation

High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g., RISC-V)

Assembler

Machine Language Program (RISC-V)

Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)
Assembly Language

• Basic job of a CPU (Central Processing Unit, aka Core)
  • Execute instructions one after another in sequence
  • Each instruction does a small amount of work (a tiny part of a larger program)
• Different CPUs implement different sets of instructions
• The set of instructions that a particular CPU implements is called its Instruction Set Architecture (ISA)
  • Examples: ARM, Intel x86, MIPS, RISC-V, IBM/Motorola PowerPC (old Mac)
RISC (Reduced Instruction Set Computer)

- A single instruction can only perform one operation
- Keep the instruction set small and simple, makes it easier to build fast hardware
- Philosophy developed by Cocke IBM, Patterson, Hennessy, 1980s
RISC-V

- Open source, license-free ISA
- 32-bit, 64-bit, and 128-bit variants
- RISC-I/II projects were in the 1980s
- RISC-V started Summer 2010 to support open research and teaching at Berkeley
- Many commercial and open source research projects based on RISC-V
- Read more
  - [https://riscv.org/about/history/](https://riscv.org/about/history/)
  - [https://riscv.org/technical/specifications/risc-v-genealogy/](https://riscv.org/technical/specifications/risc-v-genealogy/)
Why do we teach RISC-V?

• Simple
  • Don’t need to get bogged down by the details

• If you learn RISC-V, you’ll have the basic background to learn any other assembly language
Registers

• Unlike high-level languages like C or Java, assembly languages do not use variables
• Instead, they use registers
• Small storage units that are located in the processor
• Operations are performed on registers
• Registers are extremely fast due their location and size
  • Unlike the memory which is located outside of the processor
Registers are Inside the Processor
Registers

• 32 registers in RISC-V
  • Smaller is faster, but too small is bad (Goldilocks principle)

• In this class, the registers are 32-bits wide because we teach the 32-bit variant

• Word = 32 bits

• Register File = the general purpose registers inside of the processor
Registers

- Registers are numbered from 0 to 31
  - Referred to as x0 - x31
  - We cannot choose the names of the registers
- x0 is a special register because it always holds the value 0
- Unlike variables in C, registers do not have a type
  - The same register can be used to represent ints, chars, etc
Speed of Registers vs Memory

• Given that
  • Registers: 32 words (128 Bytes)
  • Memory (DRAM): Billions of bytes (2 GB to 16 GB on laptop)

• and physics dictates…
  • Smaller is faster

• Registers are about 50-500 times faster than memory!
  • in terms of latency of one access
Comments in Assembly

TAs will not be able to help you if you don’t have comments :(  

```
add x1, x2, x3  # x1 = x2 + x3
```
Addition

Operation \( \rightarrow \text{add } x_1, x_2, x_3 \)  
\# \( x_1 = x_2 + x_3 \)

Source register 1

Destination register

Source register 2
Subtraction

\[
\text{Operation} \rightarrow \textbf{sub} \ x_1, \ x_2, \ x_3 \quad \# \ x_1 = x_2 - x_3
\]
Addition and Subtraction Example

• How can I execute the following C statement in assembly?

\[ a = b + c + d - e; \]

```
add x10, x11, x12  # temp = b + c
add x10, x10, x13  # temp = temp + d
sub x10, x10, x14  # a = temp - e
```
Addition and Subtraction Example

• How can I execute the following C statement in assembly?

\[
f = (g + h) - (i + j);
\]

\[
\begin{align*}
x_{10} & \quad x_{11} & \quad x_{12} & \quad x_{13} & \quad x_{14} \\
\end{align*}
\]

\[
\begin{align*}
& \text{add } x_{5}, x_{11}, x_{12} \ # \ a_{\text{temp}} = g + h \\
& \text{add } x_{6}, x_{13}, x_{14} \ # \ b_{\text{temp}} = i + j \\
& \text{sub } x_{10}, x_{5}, x_{6} \ # \ f = (g+h)-(i+j)
\end{align*}
\]
Register x0

- Very useful: always holds zero and can never be changed (does not require initialization)
- Ex: Moving a value from one register to another:

\[ \begin{align*} f &= g \quad \text{(in C)} \\ \text{add } x3, x4, x0 &\quad \text{(in RISC-V)} \end{align*} \]
Immediates

- Immediates are used to provide numerical constants
- Constants appear often in code, so there are special instructions for them:
- Ex: Add Immediate:

  \[
  f = g - 10 \quad \text{ (in C)}
  \]

  \[
  \text{addi } x3, x4, -10 \quad \text{ (in RISC-V)}
  \]
Immediates

- There is no subtract immediate in RISC-V because we can perform any subtract immediate operation with an add immediate operation.

- RISC-V limits the operations it supports to the bare minimum:
  - If there is an operation that can be decomposed into a simpler operation, its not included.
Addi immediates are limited to 12 bits (we’ll see why later)
When you perform an operation with an immediate, it is sign extended to 32-bits
Sign Extension creates the same value

0b0000 0000 0011 => 3

0b0000 0000 0000 0000 0000 0000 0000 0011 => 3

0b1111 1111 1101 => -3
Flip Bits 0b000 0000 00010
Add 1 0b0000 0000 0011

0b1111 1111 1111 1111 1111 1111 1111 1101 => -3
Flip Bits 0b0000 0000 0000 0000 0000 0000 0000 00010
Add 1 0b0000 0000 0000 0000 0000 0000 0000 00011
Memory

Control

Datapath

Registers

Arithmetic & Logic Unit (ALU)

Processor-Memory Interface

Enable?
Read/Write

Address

Write Data = Store to memory

Read Data = Load from memory

Memory

Data

Bytes

Program

Input

 Much larger place to hold values, but slower than registers!

Output

Processor

Fast but limited place to hold values
Memory Addresses

- 1 byte = 8 bits
- 1 word = 32 bits (In a 32-bit architecture)
- We need a way to specify which bits in the memory we want to access

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>31</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word 3</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Word 2</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Word 1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>31</td>
<td>24</td>
<td>23</td>
<td>16</td>
</tr>
</tbody>
</table>
Memory Addresses

How many bits would we need if we wanted to address this memory by words?

$$\log_2(4) = 2$$
Memory Addresses

How many bits would we need if we wanted to address this memory by bytes?

\[ \log_2(16) = 4 \]
## Memory Addresses

| 127 | 126 | 125 | 124 | 123 | 122 | 121 | 120 | 119 | 118 | 117 | 116 | 115 | 114 | 113 | 112 | 111 | 110 | 109 | 108 | 107 | 106 | 105 | 104 | 103 | 102 | 101 | 100 | 99 | 98 | 97 | 96 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 95  | 94  | 93  | 92  | 91  | 90  | 89  | 88  | 87  | 86  | 85  | 84  | 83  | 82  | 81  | 80  | 79  | 78  | 77  | 76  | 75  | 74  | 73  | 72  | 71  | 70  | 69  | 68  | 67  | 66  | 65  | 64  |     |
| 63  | 62  | 61  | 60  | 59  | 58  | 57  | 56  | 55  | 54  | 53  | 52  | 51  | 50  | 49  | 48  | 47  | 46  | 45  | 44  | 43  | 42  | 41  | 40  | 39  | 38  | 37  | 36  | 35  | 34  | 33  | 32  |     |
| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |     |

How many bits would we need if we wanted to address this memory by bits?

\[ \log_2(128) = 7 \]
Memory Addresses

- Data is typically smaller than or equal to 32 bits, but is rarely smaller than 8 bits (e.g. char type)
- Applying the goldilocks principle again: memory is addressed in terms of bytes
- Can access specific bits using bitwise operators
  - & , | , >> , <<
Accessing Arrays

- If I have a pointer to an array (ptr) and I want to access the 3rd element of the array, there are two ways that I can do this
  - ptr[3]
  - *(ptr + 3)

- When writing C, you should always use the first method
- It’s important to understand the 2nd method to perform array accesses in RISC-V
Pointer Arithmetic

- C will automatically do the pointer arithmetic for you
  - If \( \text{ptr} \) is a pointer to an int array, then when you do \( \text{ptr} + 3 \), it knows to multiply 3 by \( \text{sizeof(int)} \) to get the correct address
- In RISC-V, you have to manually do the pointer arithmetic

\[ \text{ptr} \rightarrow \begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
\end{array} \]

\[ \text{ptr} + 3 \]
Loading Data from Memory into Processor Registers

Load word (lw)

Register x15 contains the pointer to an int array stored in memory. How do I store the value located at index 3 into register x10?

\[
\text{sizeof(int)} = 4
\]

\[
lw\ x10,\ 12(x15)
\]

Offset (in bytes)  Destination Register  Base register

offset must be a constant, it cannot be a register
Storing Data from Processor Registers into Memory

Store word (sw)

Register x15 contains the pointer to an int array stored in memory. How do I store the value located in register x10 to the 3rd index of the array?

\[
\text{sw } x10, 12(x15)
\]

\[
\text{sizeof(int)} = 4
\]

offset must be a constant, it cannot be a register

Source Register

Offset (in bytes)

Base register

x15 + 3(4)
Load and Store Example

Register x15 contains the pointer to an int array stored in memory. How do I increment the value stored in the 3rd index of the array by 1?

```
lw x10, 12(x15)
addi x10, x10, 1
sw x10, 12(x15)
```

![Diagram showing the process with register x15 pointing to an array and the 3rd index being incremented by 1.]
Pause
Loading and Storing Bytes

• You can also transfer data at a byte granularity

Load Byte

Store Byte

(offset is still in bytes)
Loading Bytes

• When you load a byte from memory, it is placed into the lowest byte of the destination register and sign extended.

```
lb x10, 3(x15)
```

Destination Register  Offset  Base register

x10: 

```
xxxx xxxx xxxx xxxx xxxx xxxx xxxx
```

...is copied to “sign-extend”

This bit

zzzz zzzzz

...byte loaded
Loading Bytes

- If you don’t want the number to be sign extended, you can use the instruction `lbu` instead which will zero extend to fill the register.

```
  0000 0000 0000 0000 0000 0000 0000
```

```
x10: lbu x10, 3(x15)
```

Destination Register

Offset

Base register

`zzzz zzzz` byte loaded

`x10:` zero extended
Storing bytes

- When you store a byte, only the lower 8 bits of the register is copied to memory, so there is no sign-extension.
- This means that we don’t need a store byte unsigned (sbu) instruction.
Loading vs Storing Sign Extension

- Loads are sign extended because the destination is 32 bits while the data being stored is 8 bits.
- Stores are not sign extended because the location where we are putting the data is the same size as the data we want to store (8 bits).
Recall: Immediates

- Immediates are limited to 12 bits (we’ll see why later)
- When you perform an operation with an immediate, it is sign extended to 32-bits
Sign Extension Example

What value is stored in x12 after the following code runs?

```
addi x11, x0, 0x3F5
sw x11, 0(x5)
lb x12, 1(x5)
```

- x11 = 0x000003F5
- Byte 0 = 0xF5
- Byte 1 = 0x03
- x12 = 0x00000003

```
addi x11, x0, 0x3F5
sw x11, 0(x5)
lb x12, 0(x5)
```

- x11 = 0x000003F5
- Byte 0 = 0xF5
- 0xF5 gets sign extended
- x12 = 0xFFFFFFF5

```
addi x11, x0, 0x8F5
sw x11, 0(x5)
lb x12, 1(x5)
```

- x11 = 0xFFFFF8F5
- Byte 0 = 0xF5
- Byte 1 = 0xF8
- 0x85F gets sign extended
- x12 = 0xFFFFFFF8
## RISC-V Logical Instructions

<table>
<thead>
<tr>
<th>Logical operations</th>
<th>C operators</th>
<th>Java operators</th>
<th>RISC-V instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitwise AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td><code>and</code></td>
</tr>
<tr>
<td>Bitwise OR</td>
<td></td>
<td></td>
<td><code>or</code></td>
</tr>
<tr>
<td>Bitwise XOR</td>
<td>^</td>
<td>^</td>
<td><code>xor</code></td>
</tr>
<tr>
<td>Shift left logical</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td><code>sll</code></td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;</td>
<td><code>srl/sra</code></td>
</tr>
</tbody>
</table>
Shifting

- Shift by the contents of a register
  \[ sll \ x10, \ x11, \ x12 \ # \ x10 = x11 \ll x12 \]

- Shift by a constant value
  \[ slli \ x10, \ x11, \ 2 \ # \ x10 = x11 \ll 2 \]
Left Shifting

• When you shift to the left, the bits on the left “fall off” and you insert zeros at the end

  \[
  \begin{align*}
  \text{addi} & \ x11, \ x0, \ 6 \quad x11 = \text{0b} \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0110 \\
  \text{slli} & \ x12, \ x11, \ 2 \quad x11 = \text{0b} \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0001 \ 0000 \\
  \end{align*}
  \]

• Left shifting by \( n \) is equivalent to multiplying by \( 2^n \)
• To shift to the left, we use the shift left logical (sll) instruction
• Shift left arithmetic would perform the same operation, but we don’t support it since it would be redundant
Right Shifting

- **Logical right shift**
  - The bits on the right “fall off” and zeros are inserted on the left
  - Probably don’t want to use this with negative numbers

- **Arithmetic right shift**
  - The bits on the right “fall off” and the left bits are sign extended
Right Shifting

If x10 contains 40  
`srl` x11,x10,3

x10 = 0b 0000 0000 0000 0000 0000 0000 0010 1000 = 40  
x11 = 0b 0000 0000 0000 0000 0000 0000 0000 0101 = 5

If x10 contains 41  
`srai` x11,x10,3

x10 = 0b 0000 0000 0000 0000 0000 0000 0010 1001 = 41  
x11 = 0b 0000 0000 0000 0000 0000 0000 0000 0101 = 5

If x10 contains -32  
`srai` x12,x10,4

x10 = 0b 1111 1111 1111 1111 1111 1111 1110 0000 = -32  
x12 = 0b 1111 1111 1111 1111 1111 1111 1111 1110 = -2

If x10 contains -25  
`srai` x12,x10,4

x10 = 0b 1111 1111 1111 1111 1111 1111 1110 0111 = -25  
x12 = 0b 1111 1111 1111 1111 1111 1111 1111 1110 = -2
Right Shifting

- Right shifting positive numbers and even numbers is equivalent to dividing by \(2^n\) with the fractional part of the result being truncated.
- Right shifting negative odd numbers is equivalent to dividing by \(2^n\) and rounding the result towards negative infinity.
  - This is not the behavior that we want.
  - C arithmetic semantics is that division should round towards 0.
Register x15 contains the pointer to an int array stored in memory. How do I store the value located in register x10 to the 3rd index of the array?

`sizeof(int) = 4`

`sw x10, 12(x15)`

Offset must be a constant, it cannot be a register.
Shifting to Compute Address

- What if the index that we want to reach is stored in a register? (we don’t know the value until run time)
  - We can shift to the left to calculate the index

- Ex: Register x15 contains the pointer to an int array stored in memory. How do I store the value located in register x10 to the index that is stored in x11 of the array? \( \text{sizeof(int)} = 4 \)

\[
\begin{align*}
slli & \ x12, \ x11, \ 2 \quad \# \ \text{compute offset (x12 = x11 \times 4)} \\
\text{add} & \ x12, \ x12, \ x15 \quad \# \ \text{compute address (x12 = arr + offset)} \\
\text{sw} & \ x10, \ 0(x12)
\end{align*}
\]
Decision Making Instructions

• How do we construct if-statements in RISC-V?
  • Branch instructions

• Branch instructions change the control flow of the program
  • Instead of executing instructions sequentially, you execute them in a different order (in other words, you branch to a new location)

• Types of branch instructions
  • Conditional branch
    • Only branch if some condition is met
  • Unconditional branch
    • Always branch
Labels

• Labels are used to give control flow instructions places to go
• When you are writing code, you don’t know the memory address of the location you want to go to
• You can place a label in the assembly at the place that you want to branch to and then specify that label in your code
Conditional Branches

- Branch if equal
  - `beq reg1, reg2, L1`
  - If `reg1 == reg2`, jump to code at the location of label `L1`, otherwise continue executing the code in sequence

```
if (a != b)
    e = c + d;
```

```
x10 = a
x11 = b
x12 = c
x13 = d
x14 = e
beq x10,x11,Exit
add x14,x13,x12
Exit:
```
Conditional Branches

• Branch if not equal
  • *bne reg1, reg2, L1*
  • If reg1 != reg2, jump to code at the location of label L1, otherwise continue executing the code in sequence

  ```
  if (a == b)
      e = c + d;
  bne x10,x11,Exit
  add x14,x13,x12
  Exit:
  ```

  x10 = a
  x11 = b
  x12 = c
  x13 = d
  x14 = e
Conditional Branches

• Branch on less than
  • blt reg1, reg2, L1
  • If reg1 < reg2, jump to code at the location of label L1, otherwise continue executing the code in sequence

```plaintext
if (a >= b)
    e = c + d;
    x10 = a
    x11 = b
    x12 = c
    x13 = d
    x14 = e
    blt x10,x11,Exit
    add x14,x13,x12
Exit:
```
Conditional Branches

- Branch on greater than or equal
  - \texttt{bge reg1, reg2, L1}
  - If \( \text{reg1} \geq \text{reg2} \), jump to code at the location of label \( \text{L1} \), otherwise continue executing the code in sequence

\[
\text{if (} a < b \text{)}
\]
\[
\text{e} = c + d;
\]
\[
\begin{align*}
x10 &= a \\
x11 &= b \\
x12 &= c \\
x13 &= d \\
x14 &= e
\end{align*}
\]
\[
\text{bge x10,x11,Exit} \\
\text{add x14,x13,x12} \\
\text{Exit:}
\]
Conditional Branches

- blt and bge perform signed comparisons of the numbers
- To perform unsigned comparisons, use bltu and bgeu
- RISC-V doesn’t have “branch if greater than” or “branch if less than or equal”. Instead you can reverse the arguments:
  - $A > B$ is equivalent to $B < A$
  - $A \leq B$ is equivalent to $B \geq A$
Aside: Pseudo Instructions

- Instructions that are available for the programmer’s use but are not implemented in the ISA
- These instructions are translated by the assembler to real RISC-V instructions
- Example
  - RISC-V doesn’t define bgt to avoid redundancy; however there is a bgt pseudo instruction
  - \texttt{bgt x2 x3 foo -> blt x3 x2 foo}
Unconditional Branches

- **Jump**
  - `j label`
  - Always jump to the code located at label
If-Else Statement

\[
\text{if } (a == b) \\
\quad e = c + d; \\
\text{else} \\
\quad e = c - d;
\]

\[
x10 = a \\
x11 = b \\
x12 = c \\
x13 = d \\
x14 = e
\]

\[
\text{bne } x10,x11,\text{else} \\
\text{add } x14,x12,x13 \\
j \text{ done} \\
\text{else: sub } x14,x12,x13 \\
\text{done:}
\]
Loop Example

```
int A[20];
int sum = 0;
for (int i=0; i < 20; i++)
    sum += A[i];
```

Assume x8 holds the address of the array

```
add x9,x8,x0  # x9=&A[0]
add x10,x0,x0  # sum=0
add x11,x0,x0  # i=0
addi x13,x0,20 # x13=20
Loop: bge x11,x13,Done
lw x12,0(x9)  # x12=A[i]
add x10,x10,x12 # sum+=A[i]
addi x9,x9,4   # x9=&A[i+1]
addi x11,x11,1 # i++
j Loop
Done:
```
More Instructions!

• See the 61C RISC-V Reference Card
  • https://cs61c.org/sp22/pdfs/resources/reference-card.pdf