Introduction to Assembly Language and RISC-V Instruction Set Architecture
Outline

- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- And in Conclusion …
Outline

• Assembly Language
• RISC-V Architecture
• Registers vs. Variables
• RISC-V Instructions
• C-to-RISC-V Patterns
• And in Conclusion …
Levels of Representation/Interpretation

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

Assembly Language Program (e.g., MIPS)

Machine Language Program (MIPS)

Compiler

Assembler

Machine Interpretation

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

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

temp = v[k];

v[k] = v[k+1];

v[k+1] = temp;

0000 1001 1100 0110 1010 1111 0101 1000

1010 1111 0101 1000 0000 1001 1100 0110

1100 0110 1010 1111 0101 1000 0000 1001

0101 1000 0000 1001 1100 0110 1010 1111

Anything can be represented as a number, i.e., data or instructions.
Instruction Set Architecture (ISA)

• Job of a CPU (Central Processing Unit, aka Core): execute instructions
• Instructions: CPU’s primitives operations
  • Instructions performed one after another in sequence
  • Each instruction does a small amount of work (a tiny part of a larger program).
  • Each instruction has an operation applied to operands,
    • and might be used change the sequence of instruction.
• CPUs belong to “families,” each implementing its own set of instructions
• CPU’s particular set of instructions implements an Instruction Set Architecture (ISA)
  • Examples: ARM, Intel x86, MIPS, RISC-V, IBM/Motorola PowerPC (old Mac), Intel IA64, ...
Assembly Language Programming

- Each assembly language is tied to a particular ISA (its just a human readable version of machine language).
- Why program in assembly language versus a high-level language?
  - Back in the day, when ISAs were complex and compilers were immature …. hand optimized assembly code could beat what the compiler could generate.
- These days ISAs are simple and compilers beat humans
  - Assembly language still used in small parts of the OS kernel to access special hardware resources
- For us … learn to program in assembly language
  - Best way to understand what compilers do to generate machine code
  - Best way to understand what the CPU hardware does
And
Roadmap To Future Classes...

• CS164: Compilers
  • All the processes in going from source code to assembly

• CS162: O/S
  • OS often needs a small amount of assembly for doing things the "high level" language doesn't support
    • Such as accessing special resources

• CS152: Computer Architecture
  • How to build the computer that supports the assembly

• CS161: Security
  • Exploit code ("shell code") is often in assembly and exploitation often requires understanding the assembly language of the target.
Inspired by the IBM 360 "Green Card"
Outline

• Assembly Language
• RISC-V Architecture
• Registers vs. Variables
• RISC-V Instructions
• C-to-RISC-V Patterns
• And in Conclusion …
What is RISC-V?

- Fifth generation of RISC design from UC Berkeley
- A high-quality, license-free, royalty-free RISC ISA specification
  - Implementors do not pay any royalties
  - But see Amdahl's Law:
    A decent 180 MHz 32b ARM chip costs $6 in quantity
    A Raspberry Pi (with a 1.2 GHz, quad core ARM and everything else) is $35:
    Licensing cost for the ISA can be in the noise
- Experiencing rapid uptake in both industry and academia
- Supported by growing shared software ecosystem
- Appropriate for all levels of computing system, from micro-controllers to supercomputers
  - 32-bit, 64-bit, and 128-bit variants
  - (we’re using 32-bit in class, textbook uses 64-bit)
- Standard maintained by non-profit RISC-V Foundation
Foundation Members (60+)

Platinum:

- Berkeley Architecture Research
- bluespec
- Google
- CORTEX
- DRAPE

Gold, Silver, Auditors:

- AMD
- Andes Technology
- Antmicro
- Blockstream
- ETH Zürich
Outline

• Assembly Language
• RISC-V Architecture
• Registers vs. Variables
• RISC-V Instructions
• C-to-RISC-V Patterns
• And in Conclusion …
Assembly Variables: Registers

- Unlike HLL like C or Java, assembly does not have variables as you know and love them
  - More primitive, instead what simple CPU hardware can directly support
- Assembly language operands are objects called **registers**
  - **Limited number** of special places to hold values, built directly into the hardware
  - Arithmetic operations can only be performed on these in a RISC!
    - Only memory actions are loads & stores
    - CISC can also perform operations on things **pointed to** by registers
- Benefit:
  - Since registers are directly in hardware, they are very fast to access
Registers live inside the Processor
Speed of Registers vs. Memory

• Given that
  • Registers: 32 words (128 Bytes)
  • Memory (DRAM): Billions of bytes (2 GB to 8 GB on laptop)

• and physics dictates…
  • Smaller is faster

• How much faster are registers than DRAM??
• About 100-500 times faster!
  • in terms of latency of one access
Number of RISC-V Registers

- Drawback: Registers are in hardware. To keep them really fast, their number is limited:
  - Solution: RISC-V code must be carefully written to use registers efficiently
- 32 registers in RISC-V, referred to by number x0 – x31
  - Registers are also given symbolic names, described later
  - Why 32? Smaller is faster, but too small is bad.
    - Plus need to be able to specify 3 registers in operations...
  - Each RISC-V register is 32 bits wide (RV32 variant of RISC-V ISA)
  - Groups of 32 bits called a word in RISC-V ISA
  - P&H CoD textbook uses the 64-bit variant RV64 (explain differences later)
- x0 is special, always holds value zero
  - So really only 31 registers able to hold variable values
C, Java Variables vs. Registers

• In C (and most HLLs):
  • Variables declared and given a type
    • Example:
      ```c
      int fahr, celsius;
      char a, b, c, d, e;
      ```
    • Each variable can ONLY represent a value of the type it was declared (e.g., cannot mix and match int and char variables)
      • If types are not declared, the object carries around the type with it. EG in python:
        ```python
        a = "fubar"  # now a is a string
        a = 121      # now a is an integer
        ```

• In Assembly Language:
  • Registers have no type;
  • Operation determines how register contents are interpreted
RISC-V Memory Alignment...

- RISC-V does not **require** that integers be word aligned...
  - But it is very **very bad** if you don't make sure they are...

- Consequences of unaligned integers
  - Slowdown: The processor is allowed to be a lot slower when it happens
    - In fact, a RISC-V processor may natively only support aligned accesses, and do unaligned-access in **software**!
      An unaligned load could take **hundreds of times longer**!
  - Lack of **atomicity**: The whole thing doesn't happen at once...
    can introduce lots of very subtle bugs
RISC-V Instructions

- Instructions are fixed, 32b long
  - Must be word aligned, or half-word aligned if the 16b optional (C) instruction set is also enabled
- Only a few formats (we'll go into detail later)...

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>25</th>
<th>24</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td>R-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td>I-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm[31:12]</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td>U-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Outline

• Assembly Language
• RISC-V Architecture
• Registers vs. Variables
• RISC-V Instructions
• C-to-RISC-V Patterns
• And in Conclusion …
RISC-V Instruction Assembly Syntax

• Instructions have an opcode and operands

  E.g., \texttt{add x1, x2, x3} # x1 = x2 + x3

  - Operation code (opcode)
  - Destination register
  - First operand register
  - Second operand register
  - # is assembly comment syntax
Addition and Subtraction of Integers

• Addition in Assembly
  • Example: \texttt{add x1,x2,x3} (in RISC-V)
  • Equivalent to: \( a = b + c \) (in C)
    where C variables \( \leftrightarrow \) RISC-V registers are:
    \[ a \leftrightarrow x1, \ b \leftrightarrow x2, \ c \leftrightarrow x3 \]

• Subtraction in Assembly
  • Example: \texttt{sub x3,x4,x5} (in RISC-V)
  • Equivalent to: \( d = e - f \) (in C)
    where C variables \( \leftrightarrow \) RISC-V registers are:
    \[ d \leftrightarrow x3, \ e \leftrightarrow x4, \ f \leftrightarrow x5 \]
No-Op

• A No-op is an instruction that does nothing...
  • Why?
    You may need to replace code later: No-ops can fill space, align data, and perform other options

• By convention RISC-V has a specific no-op instruction...
  • add x0 x0 x0

• Why?
  • Writes to x0 are always ignored...
    RISC-V uses that a lot as we will see in the jump-and-link operations
  • Making a "standard" no-op improves the disassembler and can potentially improve the processor
    • Special case the particular conventional no-op.
Addition and Subtraction of Integers
Example 1

- How to do the following C statement?
  \[ a = b + c + d - e; \]

- Break into multiple instructions
  
  ```c
  add x1, x2, x3  # temp = b + c
  add x1, x1, x4  # temp = temp + d
  sub x1, x1, x5  # a = temp - e
  ```

- A single line of C may turn into several RISC-V instructions
  
  ```c
  add x3, x4, x0 (in RISC-V) same
  f = g (in C)
  ```
Immediates

- **Immediates are used to provide numerical constants**
- Constants appear often in code, so there are special instructions for them:
- Ex: Add Immediate:
  
  $\text{addi } x3, x4, -10$  \hspace{1cm} \text{(in RISC-V)}

  \[ f = g - 10 \]  \hspace{1cm} \text{(in C)}

  where RISC-V registers $x3, x4$ are associated with C variables $f, g$

- Syntax similar to $\text{add}$ instruction, except that last argument is a number instead of a register
  
  $\text{addi } x3, x4, 0$  \hspace{1cm} \text{(in RISC-V)}  \hspace{1cm} \text{same as}$

  \[ f = g \]  \hspace{1cm} \text{(in C)}
Immediates & Sign Extension...

- Immediates are necessarily small
  - An I-type instruction can only have 12 bits of immediate
- In RISC-V immediates are "sign extended"
  - So the upper bits are the same as the largest bit
- So for a 12b immediate...
  - Bits 31:12 get the same value as Bit 11
Data Transfer: Load from and Store to memory

- Processor
  - Control
  - Datapath
    - Registers
    - Arithmetic & Logic Unit (ALU)
  - PC

- Memory
  - Data
  - Program
  - Address
  - Enable?
  - Read/Write
  - Write Data = Store to memory
  - Read Data = Load from memory

- Processor-Memory Interface
- I/O-Memory Interfaces
  - Input
  - Output

Fast but limited place To hold values

Much larger place To hold values, but slower than registers!
Memory Addresses are in Bytes

- Data typically smaller than 32 bits, but rarely smaller than 8 bits (e.g., char type)
  - So everything is a multiple of 8 bits
- Remember, 8 bit chunk is called a byte (1 word = 4 bytes)
- Memory addresses are really in bytes, not words
- Word addresses are 4 bytes apart
  - Word address is same as address of rightmost byte – least-significant byte (i.e. Little-endian convention)
Transfer **from** Memory to Register

- **C code**
  
  ```c
  int A[100];
  g = h + A[3];
  ```

- **Using Load Word (lw) in RISC-V:**
  
  ```
  lw $x10, 12($x13)  # Reg $x10 gets A[3]
  add $x11, $x12, $x10  # g = h + A[3]
  ```

  **Assume:**  $x13$ – base register (pointer to A[0])
  
  **Note:**  12 – offset in **bytes**

  Offset must be a constant known at assembly time
Transfer from Register to Memory

- C code
  ```c
  int A[100];
  ```

- Using Store Word (sw) in RISC-V:
  ```
  lw x10,12(x13)  # Temp reg x10 gets A[3]
  add x10,x12,x10  # Temp reg x10 gets h + A[3]
  ```

Assume:  
- x13 – base register (pointer)

Note:  
- 12, 40 – offsets in bytes

x13+12 and x13+40 must be multiples of 4
Loading and Storing Bytes

- In addition to word data transfers ($lw$, $sw$), RISC-V has **byte** data transfers:
  - load byte: $lb$
  - store byte: $sb$
- Same format as $lw$, $sw$
- E.g., $lb$ $x10,3(x11)$
  - contents of memory location with address = sum of “3” + contents of register $x11$ is copied to the low byte position of register $x10$.

$lb$ $x10,3(x11)$ is copied to “sign-extend”

RISC-V also has “unsigned byte” loads ($lbu$) which zero extend to fill register. Why no unsigned store byte $sbu$?
Your turn - clickers

addi x11, x0, 0x3f5
sw x11, 0 (x5)
lb x12, 1 (x5)

What’s the value in x12?

<table>
<thead>
<tr>
<th>Answer</th>
<th>x12</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0x5</td>
</tr>
<tr>
<td>B</td>
<td>0xf</td>
</tr>
<tr>
<td>C</td>
<td>0x3</td>
</tr>
<tr>
<td>D</td>
<td>0xffffffff</td>
</tr>
</tbody>
</table>
Your turn - clickers

```
addi x11,x0,0x3f5
sw x11,0(x5)
lb x12,1(x5)
```

What's the value in x12?

<table>
<thead>
<tr>
<th>Answer</th>
<th>x12</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0x5</td>
</tr>
<tr>
<td>B</td>
<td>0xf</td>
</tr>
<tr>
<td>C</td>
<td>0x3</td>
</tr>
<tr>
<td>D</td>
<td>0xffffffff</td>
</tr>
</tbody>
</table>
addi x11, x0, 0x8f5
sw x11, 0(x5)
lb x12, 1(x5)

What’s the value in x12?

<table>
<thead>
<tr>
<th>Answer</th>
<th>x12</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0x8</td>
</tr>
<tr>
<td>B</td>
<td>0xf8</td>
</tr>
<tr>
<td>C</td>
<td>0x3</td>
</tr>
<tr>
<td>D</td>
<td>0xffffffff8</td>
</tr>
</tbody>
</table>
Your turn - clickers

```
addi x11, x0, 0x8f5
sw x11, 0(x5)
lb x12, 1(x5)
```

What's the value in x12?

<table>
<thead>
<tr>
<th>Answer</th>
<th>x12</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0x8</td>
</tr>
<tr>
<td>B</td>
<td>0xf8</td>
</tr>
<tr>
<td>C</td>
<td>0x3</td>
</tr>
<tr>
<td>D</td>
<td>0xffffffff8</td>
</tr>
</tbody>
</table>
• Load Balancing for labs:
  • When the new lab starts, all those in the room from the previous lab have to leave
  • Can then come back if there is more space left

• Tutoring (and lots of it!)
  • Can sign up for CS 370 tutoring now
    • Link on Piazza
  • CSM tutoring starts next week
  • As soon as you think you are starting to struggle, get help!
RISC-V Logical Instructions

Useful to operate on fields of bits within a word
e.g., characters within a word (8 bits)
Operations to pack /unpack bits into words
Called logical operations

<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>Bit-by-bit AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and</td>
</tr>
<tr>
<td>Bit-by-bit OR</td>
<td></td>
<td></td>
<td>or</td>
</tr>
<tr>
<td>Bit-by-bit XOR</td>
<td>^</td>
<td>^</td>
<td>xor</td>
</tr>
<tr>
<td>Shift left logical</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;</td>
<td>srl/sra</td>
</tr>
</tbody>
</table>
Logical Shifting

- **Shift Left Logical:** `slli x11, x12, 2` # `x11 = x12 << 2`
  - Store in `x11` the value from `x12` shifted 2 bits to the left (they fall off end), inserting 0’s on right; `<<` in C
    - Before: \(0000\ 0002\) \(_\text{hex}\)
    - \(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010\) \(_\text{two}\)
    - After: \(0000\ 0008\) \(_\text{hex}\)
    - \(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1000\) \(_\text{two}\)

What arithmetic effect does shift left have?

- **Shift Right Logical:** `srli` is opposite shift; `>>`
  - Zero bits inserted at left of word, right bits shifted off end
Arithmetic Shifting

- *Shift right arithmetic* (srai) moves \( n \) bits to the right (insert high-order sign bit into empty bits)
- For example, if register x10 contained 
  \[ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0111 \] 
  \( \text{two} = -25 \) \( \text{ten} \)
- If execute sra x10, x10, 4, result is:
  \[ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0111 \] 
  \( \text{two} = -2 \) \( \text{ten} \)
- Unfortunately, this is NOT same as dividing by \( 2^n \)
  - Fails for odd negative numbers
  - C arithmetic semantics is that division should round towards 0
Computer Decision Making

- Based on computation, do something different
- Normal operation on CPU is to execute instructions in sequence
- Need special instructions for programming languages: if-statement

RISC-V: if-statement instruction is

```
beq register1, register2, L1
```

means: go to instruction labeled L1 if (value in register1) == (value in register2)

....otherwise, go to next instruction

- `beq` stands for branch if equal
- Other instruction: `bne` for branch if not equal
Types of Branches

• **Branch** – change of control flow

• **Conditional Branch** – change control flow depending on outcome of comparison
  - branch *if* equal (**beq**) or branch *if not* equal (**bne**)
  - Also branch if less than (**blt**) and branch if greater than or equal (**bge**)

• **Unconditional Branch** – always branch
  - a RISC-V instruction for this: *jump* (**j**)
    - *We will see later than j doesn't exist (its a "pseudo-instruction")*
Outline

- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- And in Conclusion …
Example if Statement

• Assuming assignments below, compile if block

\[ f \rightarrow x_{10} \quad g \rightarrow x_{11} \quad h \rightarrow x_{12} \]
\[ i \rightarrow x_{13} \quad j \rightarrow x_{14} \]

\[
\text{if} \ (i == j) \quad \text{bne} \ x_{13}, x_{14}, \text{done} \\
f = g + h; \quad \text{add} \ x_{10}, x_{11}, x_{12} \\
\text{done:}
\]
Example *if-else* Statement

- Assuming assignments below, compile

\[
\begin{align*}
    f & \rightarrow x_{10} &
    g & \rightarrow x_{11} &
    h & \rightarrow x_{12} &
    i & \rightarrow x_{13} &
    j & \rightarrow x_{14} \\

    \text{if } (i == j) & & \text{bne } x_{13},x_{14},\text{else} \\
    f & = g + h; & \text{add } x_{10},x_{11},x_{12} \\
\text{else} & & \text{j done} \\
    f & = g - h; & \text{else: sub } x_{10},x_{11},x_{12} \\
    \text{done:} & & \\
\end{align*}
\]

Magnitude Compares in RISC-V

• Until now, we’ve only tested equalities (== and != in C); General programs need to test < and > as well.

• RISC-V magnitude-compare branches:
  “Branch on Less Than”
  Syntax: \texttt{blt \ reg1,reg2, label}
  Meaning: if (\texttt{reg1 < reg2}) \ // \ Registers are signed
            \texttt{goto label;}

• “Branch on Less Than Unsigned”
  Syntax: \texttt{bltu \ reg1,reg2, label}
  Meaning: if (\texttt{reg1 < reg2}) \ // \ treat registers as unsigned integers
            \texttt{goto label;}

“Branch on Greater Than or Equal” (and it’s unsigned version) also exists.
But RISC philosophy...

- A CISC might also have "branch if greater than"...
  - But RISC-V doesn't.

- Instead you can switch the argument
  - branch if greater then reg1 reg2...
  - branch if less than reg2 reg1
C Loop Mapped to RISC-V Assembly

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

```assembly
   # Assume x8 holds pointer to A
   # Assign x10=sum, x11=i
   add x10, x0, x0 # sum=0
   add x11, x0, x0 # i=0
   addi x12, x0, 20 # x12=20
   Loop:
   bge x11, x12, exit:
   sll x13, x11, 2 # i * 4
   add x13, x13, x8 # & of A + i
   lw x13, 0(x13) # *(A + i)
   add x10, x10, x13 # increment sum
   addi x11, x11, 1 # i++
   j Loop # Iterate
   exit:
```
Comments...

- The simple translation is suboptimal!
  - A more efficient way:
- Inner loop is now 4 instructions rather than 7
  - And only 1 branch/jump rather than two:
    Because first time through is always true so can move check to the end!
- The compiler will often do this automatically for optimization
  - See that i is only used as an index in a loop

```assembly
# Assume x8 holds pointer to A
# Assign x10=sum
add  x10, x0, x0  # sum=0
add  x11, x8, x8  # Copy of A
addi x12,x11, 80  # x12=80 + A
Loop:
lw   x13, 0(x11)
add  x10, x10, x13
addi x11, x11, 4
blt  x11, x12, loop:
```
And Premature Optimization...

- In general we want **correct** translations of C to RISC-V
- It is **not** necessary to optimize
  - Just translate each C statement on its own
- Why?
  - Correctness first, performance second
    - Getting the wrong answer fast is not what we want from you...
  - We're going to need to read your assembly to grade it!
    - Multiple ways to optimize, but the straightforward translation is mostly unique-ish.
Outline

- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- And in Conclusion …
In Conclusion,…

- Instruction set architecture (ISA) specifies the set of commands (instructions) a computer can execute
- Hardware registers provide a few very fast variables for instructions to operate on
- RISC-V ISA requires software to break complex operations into a string of simple instructions, but enables faster, simple hardware
- Assembly code is human-readable version of computer’s native machine code, converted to binary by an assembler