Outline

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

Levels of Representation/Interpretation

<table>
<thead>
<tr>
<th>High Level Language Program (e.g., C)</th>
<th>Machine Language Program (RISC-V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp = v[k]; v[k+1] = v[k+1];</td>
<td>1010 1111 0101 1000 0000 1001 1100 0110</td>
</tr>
<tr>
<td>Anything can be represented as a number.</td>
<td>0001 1010 1110 0110 1001 0000 1011 0110</td>
</tr>
<tr>
<td></td>
<td>1110 0110 1010 1111 0100 1100 1000 1001</td>
</tr>
</tbody>
</table>

Instruction Set Architecture (ISA)

- Job of a CPU (Central Processing Unit, aka Core): execute instructions
- Instructions: CPU’s primitives operations
  - Like a sentence: operations (verbs) applied to operands (objects) processed in sequence ...
  - With additional operations to change the sequence
- 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 i64, ...
Instruction Set Architectures

- Early trend: add more instructions to new CPUs for elaborate operations
  - VAX architecture had an instruction to multiply polynomials!
- RISC philosophy (Cocke IBM, Patterson UCB, Hennessy Stanford, 1980s) — Reduced Instruction Set Computing
  - Keep the instruction set small and simple, in order to build fast hardware
  - Let software do complicated operations by composing simpler ones

RISC-V Green Card (in textbook)

Inspired by the IBM 360 “Green Card”

http://inst.eecs.berkeley.edu/~cs61c/resources/RISCV_Green_Sheet.pdf

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
- Experiencing rapid uptake in both industry and academia
- Both proprietary and open-source core implementations
- Supported by growing shared software ecosystem
- Appropriate for all levels of computing system, from microcontrollers 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
Assembly Variables: Registers

- Unlike HLL like C or Java, assembly does not have variables as you know and love them.
  - More primitive, closer what simple hardware can directly support.
- Assembly operands are objects called *registers*.
  - Limited number of special places to hold values, built directly into the hardware.
  - Operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they are very fast (faster than 1 ns - light travels 1 foot in 1 ns!!!)

Registers live inside the Processor

C, Java Variables vs. Registers

- In C (and most HLLs):
  - Variables declared and given a type
    - Example: `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)
- In Assembly Language:
  - Registers have no type;
  - Operation determines how register contents are interpreted.

Number of RISC-V Registers

- Drawback: Since registers are in hardware, there are a limited number of them
  - Solution: RISC-V code must be carefully written to efficiently use registers
  - 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. Goldilocks principle (“This porridge is too hot; This porridge is too cold; this porridge is just right”)
  - 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
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., **add x1, x2, x3 # x1 = x2 + x3**

Addition and Subtraction of Integers

• Addition in Assembly
  - Example: `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: `sub x3, x4, x5` (in RISC-V)
  - Equivalent to: $d = e - f$ (in C)
  - Where **C** variables $\leftrightarrow$ RISC-V registers are:
    - $d \leftrightarrow x3$, $e \leftrightarrow x4$, $f \leftrightarrow x5$

Addition and Subtraction of Integers

Example 1

• How to do the following **C** statement?
  - $a = b + c + d - e$;
• Break into multiple instructions
  - `add x10, x1, x2` # $a_{temp} = b + c$
  - `add x10, x10, x3` # $a_{temp} = a_{temp} + d$
  - `sub x10, x10, x4` # $a = a_{temp} - e$
• A single line of **C** may turn into several RISC-V instructions

Immediate

• Immediates are numerical constants
• They appear often in code, so there are special instructions for them
• Add Immediate:
  - `addi x3, x4, -10` (in RISC-V)
  - $f = g - 10$ (in C)
  - Where RISC-V registers $x3, x4$ are associated with **C** variables $f, g$
• Syntax similar to add instruction, except that last argument is a number instead of a register
  - `add x3, x4, x0` (in RISC-V)
  - $f = g$ (in C)

Data Transfer:

Load from and Store to memory

Much larger place To hold values, but

Fast but limited place To hold values, but
Memory Addresses are in Bytes

- Data typically smaller than 32 bits, but rarely smaller than 8 bits (e.g., char type)—works fine if everything is a multiple of 8 bits
- 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:
  ```c
  lw x10,12(x13) # Reg x10 gets A[3]
  add x11,x12,x10 # g = h + A[3]
  ```
- 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:
  ```c
  add x10,x12,x10 # Temp reg x10 gets A[3]
  ```
- 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.
  ```c
  x10 = ...is copied to “sign-extend”
  ```
  This bit
  ```c
  byte
  ```
  RISC-V also has “unsigned byte” loads (`lbu`) which zero extend to full register. Why no `unsigned store byte `sbu`?

Loading and Storing Bytes

- In addition to word data transfers (`lw`, `sw`), RISC-V has `byte` data transfers:
  - load byte: `lb`
  - store byte: `sb`
- 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.
  ```c
  x10 = ...
  ```
  This bit
  ```c
  byte
  ```
  RISC-V also has “unsigned byte” loads (`lbu`) which zero extend to full register. Why no `unsigned store byte `sbu`?

Your turn

<table>
<thead>
<tr>
<th>Code</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi x11,x0,0x3f5</td>
<td>x12</td>
</tr>
<tr>
<td>sw x11,0(x5)</td>
<td>RED 0x5</td>
</tr>
<tr>
<td>lb x12,1(x5)</td>
<td>GREEN 0xf</td>
</tr>
<tr>
<td></td>
<td>ORANGE 0x3</td>
</tr>
<tr>
<td></td>
<td>YELLOW 0xffffffff</td>
</tr>
</tbody>
</table>

Your turn

<table>
<thead>
<tr>
<th>Code</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi x11,x0,0x3f5</td>
<td>x12</td>
</tr>
<tr>
<td>sw x11,0(x5)</td>
<td>RED 0x5</td>
</tr>
<tr>
<td>lb x12,1(x5)</td>
<td>GREEN 0xf</td>
</tr>
<tr>
<td></td>
<td>ORANGE 0x3</td>
</tr>
<tr>
<td></td>
<td>YELLOW 0xffffffff</td>
</tr>
</tbody>
</table>
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??
  – in terms of latency of one access

Administrivia

• HW #0 due tomorrow night!
• HW #1 will be published soon
  – Two-part C programming assignment
• Small Group Tutoring sign ups will be out right after today’s lecture
• Three weeks to Midterm #1!

Break!

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>&lt;</td>
<td>&lt;</td>
<td>xor</td>
</tr>
<tr>
<td>Shift left logical</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;</td>
<td>srl</td>
</tr>
</tbody>
</table>

Logical Shifting

• Shift Left Logical: \texttt{slli} \texttt{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: \texttt{0000 0000 0000 0000 0000 0000 0000 0010} two
  After: \texttt{0000 0000 0000 0000 0000 0000 0000 1000} two
  
  What arithmetic effect does shift left have?

• Shift Right Logical: \texttt{srli} is opposite shift; >>
  – Zero bits inserted at left of word, right bits shifted off end

Arithmetic Shifting

• Shift right arithmetic (\texttt{sra}) moves \(n\) bits to the right (insert high-order sign bit into empty bits)
  
  – For example, if register x10 contained \texttt{1111 1111 1111 1111 1111 1111 1110 0111} two\(= -25_{\text{ten}}\)
  
  – If execute \texttt{sra} x10, x10, 4, result is: \texttt{1111 1111 1111 1111 1111 1111 1111 1110} 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
- In programming languages: if-statement
- RISC-V: if-statement instruction is `beq register1, register2, L1`
  - means: go to statement labeled L1 if (value in register1) == (value in register2)
  - otherwise, go to next statement
- `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: `jmp` (j)

Outline

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

Example if Statement

- Assuming translations below, compile if block
  - f → x10, g → x11, h → x12
  - i → x13, j → x14
  - if (i == j) bne x13,x14,Exit
  - f = g + h; add x10,x11,x12
  - Else: sub x10,x11,x12 Exit:

Example if-else Statement

- Assuming translations below, compile
  - f → x10, g → x11, h → x12
  - i → x13, j → x14
  - if (i == j) bne x13,x14,Exit
  - f = g + h; add x10,x11,x12
  - else j Exit
  - f = g - h; Else: sub x10,x11,x12 Exit:

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: `blt reg1, reg2, label`
    - Meaning: if (reg1 < reg2) // treat registers as signed integers goto label;
  - “Branch on Less Than Unsigned”
    - Syntax: `bltu reg1, reg2, label`
    - Meaning: if (reg1 < reg2) // treat registers as unsigned integers goto label;
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
add x9, x8, x0 # x9=A[0]
add x10, x0, x0 # sum=0
add x11, x0, x0 # i=0
Loop:
    lw x12, 0(x9) # x12=A[i]
    add x10,x10,x12 # sum+=
    addi x9,x9,4 # &A[i++]
    addi x11,x11,1 # i++
    addi x13,x0,20 # x13=20
    blt x11,x13,Loop
```

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