CS 61C: Great Ideas in Computer Architecture
Introduction to Assembly Language and RISC-V Instruction Set Architecture

Instructors:
Nick Weaver & John Wawrzynek
http://inst.eecs.Berkeley.edu/~cs61c
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., RISC-V)**
- **Machine Language Program (RISC-V)**

### Compiler
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

### Assembler
```
lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
```

### Machine Interpretation

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

**Architecture Implementation**

**Logic Circuit Description (Circuit Schematic Diagrams)**

**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, ...
Instruction Set Architectures

• Early trend: add more instructions to new CPUs for elaborate operations
  – Made assembly language programming easier.
  – VAX architecture had an instruction to compute polynomials!
    \[
    \text{result} = C[0] + x^{**0} + x*(C[1] + x*(C[2] + ... x*C[d]))
    \]

• 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 compiler generate software do complicated operations by composing simpler ones
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 where complex and compilers where 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
  1. Best way to understand what compilers do to generate machine code
  2. Best way to understand what the CPU hardware does
  3. *Plus its great fun!*
### RISC-V Green Card

(in textbook)

---

http://inst.eecs.berkeley.edu/~cs61c/resources/RISCV_Green_Sheet.pdf
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
- 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
- Cortus
- Samsung
- Microsemi
- Qualcomm
- Microsoft
- Rambus
- Mellanox Technologies
- NVIDIA
- Western Digital
- HUAWEI

Gold, Silver, Auditors:
- AMD
- Andes Technology
- Esperanto Technologies
- ESPRESSIF
- antmicro
- Blockstream
- J-TECH
- MII
- intrinsX
- RDA Logic
- Rumble Development
- CADENCE
- Syntacore
- lowRISC
- VectorBlox
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, 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
  – Operations can only be performed on these!
• Benefit: Since registers are directly in hardware, they are very fast to access (faster than 1 ns - light travels 1 foot in 1 ns!!!)
Registers live inside the Processor

Processor

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

Registers

Memory

- Data
- Bytes
- Program

Input

- Enable?
- Read/Write
- Address
- Write
- Data
- Read
- Data

Output

Processor-Memory Interface

I/O-Memory Interfaces
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 $x_0 - x_{31}$**
  - 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)
- **$x_0$ 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:
      ```
      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
Outline

• Assembly Language
• RISC-V Architecture
• Registers vs. Variables
• RISC-V Instructions
• C-to-RISC-V Patterns
• And in Conclusion ...
• 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: \( \text{add } x_1, x_2, x_3 \) (in RISC-V)
  – Equivalent to: \( a = b + c \) (in C)
    where C variables \( \leftrightarrow \) RISC-V registers are:
    \( a \leftrightarrow x_1, b \leftrightarrow x_2, c \leftrightarrow x_3 \)

• Subtraction in Assembly
  – Example: \( \text{sub } x_3, x_4, x_5 \) (in RISC-V)
  – Equivalent to: \( d = e - f \) (in C)
    where C variables \( \leftrightarrow \) RISC-V registers are:
    \( d \leftrightarrow x_3, e \leftrightarrow x_4, f \leftrightarrow x_5 \)
Addition and Subtraction of Integers Example 1

• How to do the following C statement?

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

• Break into multiple instructions

\[
\begin{align*}
\text{add } & \ x10, \ x1, \ x2 & \# \ temp = b + c \\
\text{add } & \ x10, \ x10, \ x3 & \# \ temp = \ temp + d \\
\text{sub } & \ x10, \ x10, \ x4 & \# \ a = \ temp - e
\end{align*}
\]

• A single line of C may turn into several RISC-V instructions

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

- **Immediates are used to provide numerical constants**
- Constants appear often in code, so there are special instructions for them:
- Ex: Add Immediate:
  \[
  \text{addi } x3, x4, -10 \quad \text{(in RISC-V)} \\
  f = g - 10 \quad \text{(in C)}
  \]
  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
  \[
  \text{addi } x3, x4, 0 \quad \text{(in RISC-V) same as} \\
  f = g \quad \text{(in C)}
  \]
Data Transfer: Load from and Store to memory

Processor - Memory Interface

I/O-Memory Interfaces

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

Fast but limited place
To hold values

Program = $24$

Input

Output
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
- 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)

Words in memory must start at byte addresses that are even multiples of 4, i.e., words must be aligned. Note: aligned words have the low 2 bits of their address = 0.
Transfer from Memory to Register

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

- **Using Load Word (lw) in RISC-V:**
  ```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**
  
  ```
  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.

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

x10: 

...is copied to “sign-extend”

This bit loaded
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>
The Project 1 deadline extended to Thursday, 11:59pm!
Send DSP letters to Pejie <li_paige@berkeley.edu>.
There will be a guerrilla section Thursday 7-9PM.
Two weeks to Midterm #1!
Project 2-1 release later this week or early next, due 2/16.
Project 2-2 release right after midterm and due 2/23.
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><strong>and</strong></td>
</tr>
<tr>
<td>Bit-by-bit OR</td>
<td></td>
<td></td>
<td><strong>or</strong></td>
</tr>
<tr>
<td>Bit-by-bit XOR</td>
<td>^</td>
<td>^</td>
<td><strong>xor</strong></td>
</tr>
<tr>
<td>Shift left logical</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td><strong>sll</strong></td>
</tr>
<tr>
<td>Shift right logical</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;</td>
<td><strong>srl</strong></td>
</tr>
</tbody>
</table>
Logical Shifting

• Shift Left Logical: \texttt{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: \texttt{0000 0002}_{\text{hex}}
  
  \texttt{0000 0000 0000 0000 0000 0000 0000 0010}_{\text{two}}
  
  After: \texttt{0000 0008}_{\text{hex}}
  
  \texttt{0000 0000 0000 0000 0000 0000 0000 1000}_{\text{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* (srai) moves *n* bits to the right (insert high-order sign bit into empty bits)

• For example, if register x10 contained
  \[1111 1111 1111 1111 1111 1111 1110 0111\]  
two  =  -25_{ten}

• If execute sra x10, x10, 4, result is:
  \[1111 1111 1111 1111 1111 1111 1111 1111 1110\]  
two  =  -2_{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*)
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

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

\[
\text{if (i == j)} \quad \text{bne x}_{13}, x_{14}, \text{skip} \\
f = g + h; \quad \text{add x}_{10}, x_{11}, x_{12} \\
\text{. skip: .} \\
\]

38
Example *if-else* Statement

- Assuming assignments below, compile

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

  \[
  \text{if} \ (i == j) \quad \text{bne} \ x_{13}, x_{14}, \text{else}
  \]

  \[
  f = g + h; \quad \text{add} \ x_{10}, x_{11}, x_{12}
  \]

  \[
  \text{else:} \quad \text{sub} \ x_{10}, x_{11}, x_{12}
  \]

  \[
  \text{done:}
  \]

  \[
  \text{else:}
  \]

  \[
  \text{done:}
  \]
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 (reg1 < reg2) // treat registers as signed integers
            goto label;

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

  “Branch on Greater Than or Equal” (and it’s unsigned version) also exists.
int A[20];
int sum = 0;
for (int i=0; i<20; i++)
    sum += A[i];

# Assume x8 holds pointer to A
# Assign x9=A, x10=sum, x11=i
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:
lw x12, 0(x9)   # x12=A[i]
add x10,x10,x12 # sum+=
addi x9,x9,4    # &A[i++]
addi x11,x11,1  # i++
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