CS61C
Great Ideas in Computer Architecture (a.k.a. Machine Structures)

RISC-V Assembly Language

cs61c.org
Great Idea #1: Abstraction (Levels of Representation/Interpretation)

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

```c
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw x3, 0(x10)</td>
<td>1000 1101 1110 0010 0000 0000 0000 0000</td>
</tr>
<tr>
<td>lw x4, 4(x10)</td>
<td>1000 1110 0001 0000 0000 0000 0000 0100</td>
</tr>
<tr>
<td>sw x4, 0(x10)</td>
<td>1010 1110 0001 0010 0000 0000 0000 0000</td>
</tr>
<tr>
<td>sw x3, 4(x10)</td>
<td>1010 1101 1110 0010 0000 0000 0000 0100</td>
</tr>
</tbody>
</table>

Anything can be represented as a number, i.e., data or instructions

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

Logic Circuit Description (Circuit Schematic Diagrams)
Assembly Language

- Basic job of a CPU: execute lots of *instructions*.
- Instructions are the primitive operations that the CPU may execute.
  - Like a sentence: operations (verbs) applied to operands (objects) processed in sequence ...
- Different CPUs implement different sets of instructions. The set of instructions a particular CPU implements is an *Instruction Set Architecture (ISA)*.
  - Examples: ARM (cell phones), Intel x86 (i9, i7, i5, i3), IBM Power, IBM/Motorola PowerPC (old Macs), MIPS, RISC-V, ...
"A new book was just released which is based on a new concept - teaching computer science through assembly language (Linux x86 assembly language, to be exact). This book teaches how the machine itself operates, rather than just the language. I've found that the key difference between mediocre and excellent programmers is whether or not they know assembly language. *Those that do* tend to understand computers themselves at a much deeper level. Although [almost!] unheard of today, this concept isn't really all that new -- there used to not be much choice in years past. Apple computers came with only BASIC and assembly language, and there were books available on assembly language for kids. This is why the old-timers are often viewed as 'wizards': they *had* to know assembly language programming."

-- slashdot.org comment, 2004-02-05
Instruction Set Architectures

- Early trend was to add more and more instructions to new CPUs to do elaborate operations
  - VAX architecture had an instruction to multiply polynomials!

- RISC philosophy (Cocke IBM, Patterson, Hennessy, 1980s) – Reduced Instruction Set Computing
  - Keep the instruction set small and simple, makes it easier to build fast hardware.
  - Let software do complicated operations by composing simpler ones.
  - This went against the convention wisdom of the time. (he who laughs last, laughs best)
Patterson and Hennessy win Turing!
RISC-V Architecture

- New open-source, license-free ISA spec
  - 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)

- Why RISC-V instead of Intel 80x86?
  - RISC-V is simple, elegant. Don’t want to get bogged down in gritty details.
  - RISC-V has exponential adoption

https://cs61c.org/resources/
RISC-V Origins

- Started in Summer 2010 to support open research and teaching at UC Berkeley
  - Lineage can be traced to RISC-I/II projects (1980s)
- As the project matured, it migrated to RISC-V foundation (www.riscv.org)
- Many commercial and research projects based on RISC-V, open-source and proprietary
  - Widely used in education
- Read more:
  - https://riscv.org/risc-v-history/
  - https://riscv.org/risc-v-genealogy/
Elements of Architecture: Registers
Instruction Set

Instruction set for a particular architecture (e.g., RISC-V) is represented by the Assembly language.

Each line of assembly code represents one instruction for the computer.

---

Preliminary discussion of the logical design of an electronic computing instrument

Arthur W. Burks / Herman H. Goldstine / John von Neumann

3.1. It is easy to see by formal-logical methods that there exist codes that are *in abstracto* adequate to control and cause the execution of any sequence of operations which are individually available in the machine and which are, in their entirety, conceivable by the problem planner. The really decisive considerations from the present point of view, in selecting a code, are more of a practical nature: simplicity of the equipment demanded by the code, and the clarity of its application to the actually important problems together with the speed of its handling of those problems. It would take us much too far afield to discuss these questions at all generally or from first principles. We will therefore restrict ourselves to analyzing only the type of code which we now envisage for our machine.
Assembly Variables: Registers (1/3)

- Unlike HLL like C or Java, assembly cannot use variables
  - Why not? Keep Hardware Simple
- Assembly operands are registers
  - Limited number of special locations built directly into the hardware
  - Operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they’re very fast (faster than 0.25ns)
  - Recall light is \(3 \times 10^8\) m/s = 0.3 m/ns = 30 cm/ns = 10 cm/0.3 ns!!!... where 0.3 ns is the clock period of a 3.33 GHz computer
Aside: Registers are **Inside** the Processor

- **Processor**
  - Control
  - Datapath
    - Program Counter (PC)
    - Registers
    - Arithmetic-Logic Unit (ALU)
  - Memory Interface
- **Memory**
  - Address
  - Write Data
  - Read Data
  - Memory Interface
- **I/O**
  - Input
  - Output
  - Processor-Memory Interface
  - I/O-Memory Interfaces

---

**Berkeley UNIVERSITY OF CALIFORNIA**

Garcia, Nikolić

**RISC-V (12)**
Great Idea #3: Principle of Locality / Memory Hierarchy

- Extremely fast
- Extremely expensive
- Tiny capacity
Jim Gray’s Storage Latency Analogy: 
How Far Away is the Data?

This Campus

1 [ns] Registers

My Head 1 min

Jim Gray
Turing Award
B.S. Cal 1966
Ph.D. Cal 1969

Garcia, Nikolić

RISC-V (14)
Drawback: Since registers are in hardware, there is a predetermined number of them
  - Solution: RISC-V code must be very carefully put together to efficiently use registers

32 registers in RISC-V
  - 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 (in RV32 variant)
  - Groups of 32 bits called a word in RV32
  - P&H textbook uses the 64-bit variant RV64
Assembly Variables: Registers (3/3)

- Registers are numbered from 0 to 31
  - Referred to by number \( x_0 \) – \( x_{31} \)
- \( x_0 \) is special, always holds value zero
  - So only 31 registers able to hold variable values
- Each register can be referred to by number or name
  - Will add names later
C, Java variables vs. registers

- In C (and most high-level languages) variables declared first and given a type. E.g.,
  ```c
  int fahr, celsius;
  char a, b, c, d, e;
  ```
- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match `int` and `char` variables).
- In assembly language, the registers have no type
  - Operation determines how register contents are treated
Comments in Assembly

- Make your code more readable: comments!
- Hash (#) is used for RISC-V comments
  - anything from hash mark to end of line is a comment and will be ignored
  - This is just like the C99 //
- Note: Different from C.
  - C comments have format /* comment */
  - so they can span many lines
Aside: Apollo Guidance Computer

Margaret Hamilton
(Wikimedia commons)

Assembly code with comments
(ABC News, 2018)
Assembly Instructions

- In assembly language, each statement (called an Instruction), executes exactly one of a short list of simple commands
- Unlike in C (and most other high-level languages), each line of assembly code contains at most 1 instruction
- Instructions are related to operations (=, +, -, *, /) in C or Java
- Ok, enough already...gimme my RV32!
RISC-V Add/Sub Instructions
RISC-V Addition and Subtraction (1/4)

- Syntax of Instructions:
  - one  two, three, four

  add      x1, x2, x3

  where:
  - one = operation by name
  - two = operand getting result ("destination," x1)
  - three = 1st operand for operation ("source1," x2)
  - four = 2nd operand for operation ("source2," x3)

- Syntax is rigid:
  - 1 operator, 3 operands
  - Why? Keep hardware simple via regularity
Addition and Subtraction of Integers (2/4)

- **Addition in Assembly**
  - Example: `add x1, x2, x3` (in RISC-V)
  - Equivalent to: `a = b + c` (in C)
  - where C variables ⇔ RISC-V registers are:
    - `a ⇔ x1`, `b ⇔ x2`, `c ⇔ x3`

- **Subtraction in Assembly**
  - Example: `sub x3, x4, x5` (in RISC-V)
  - Equivalent to: `d = e - f` (in C)
  - where C variables ⇔ RISC-V registers are:
    - `d ⇔ x3`, `e ⇔ x4`, `f ⇔ x5`
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
```

Notice: A single line of C may break up into several lines of RISC-V.

Notice: Everything after the hash mark on each line is ignored (comments).
Addition and Subtraction of Integers (4/4)

- How do we do this?
  \[ f = (g + h) - (i + j); \]

- Use intermediate temporary register
  
  \[
  \begin{align*}
  \text{add } & \text{x5, x20, x21} & \# \text{a_temp} = g + h \\
  \text{add } & \text{x6, x22, x23} & \# \text{b_temp} = i + j \\
  \text{sub } & \text{x19, x5, x6} & \# f = (g + h) - (i + j)
  \end{align*}
  \]

- A good compiler may do:
RISC-V
Immediates
Immediates

- Immediates are numerical constants.
- They appear often in code, so there are special instructions for them.
- 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.
There is no Subtract Immediate in RISC-V: Why?
- There are add and sub, but no addi counterpart

Limit types of operations that can be done to absolute minimum
- if an operation can be decomposed into a simpler operation, don’t include it
- addi ..., -x = “subi ... , x” => so no “subi”
  - 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, respectively
Register Zero

- One particular immediate, the number zero (0), appears very often in code.
- So the register zero (x0) is ‘hard-wired’ to value 0; e.g.
  - `add x3, x4, x0` (in RISC-V)
  - `f = g` (in C)
  - where RISC-V registers `x3, x4` are associated with C variables `f, g`
- Defined in hardware, so an instruction `add x0, x3, x4` will not do anything!
Storing Data in Memory
### RV32 So Far…

- **Addition/subtraction**
  
  ```
  add rd, rs1, rs2  
  \[ R[rd] = R[rs1] + R[rs2] \]
  
  sub rd, rs1, rs2  
  \[ R[rd] = R[rs1] - R[rs2] \]
  ```

- **Add immediate**
  
  ```
  addi rd, rs1, imm  
  \[ R[rd] = R[rs1] + \text{imm} \]
  ```
Data Transfer: **Load from and Store to memory**

**Processor**
- Control
- Datapath
  - Program Counter (PC)
  - Registers
  - Arithmetic-Logic Unit (ALU)

**Memory**
- Program
- Data
  - Bytes

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

**Output**
- Very fast, but limited space to hold values!

---

**Control**
- Enable?
- Read/Write

**Datapath**
- Address
- Write Data
- Read Data

**Program**
- = Store to Memory

**Data**
- = Load from Memory

---

Garcia, Nikolić

**Berkeley UNIVERSITY OF CALIFORNIA**
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)
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 \((1 \text{ word} = 4 \text{ bytes})\)
- 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)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Least-significant byte gets the smallest address
The adjective endian has its origin in the writings of 18th century writer Jonathan Swift. In the 1726 novel Gulliver's Travels, he portrays the conflict between sects of Lilliputians divided into those breaking the shell of a boiled egg from the big end or from the little end. He called them the "Big-Endians" and the "Little-Endians".

- The order in which BYTES are stored in memory
- Bits always stored as usual (E.g., 0xC2=0b 1100 0010)

Consider the number 1025 as we typically write it:

```
00000000 00000000 00000100 00000001
```

Examples

Names in the US (e.g., Bora Nikolić)
Internet names (e.g., cs.berkeley.edu)
Dates written in Europe DD/MM/YYYY (e.g., 07/09/2020)
Eating Pizza crust first

Names in China or Hungary (e.g., Nikolić Bora)
Java Packages (e.g., org.mypackage.HelloWorld)
Dates in ISO 8601 YYYY-MM-DD (e.g., 2020-09-07)
Eating Pizza skinny part first
Data Transfer Instructions
Great Idea #3: Principle of Locality / Memory Hierarchy

- Processor chip
  - Extremely fast
  - Extremely expensive
  - Tiny capacity

- DRAM chip (e.g., DDR3/4/5, HBM/HBM2/3)
  - Fast
  - Priced reasonably
  - Medium capacity

Physical Memory

Random-Access Memory (RAM)
Speed of Registers vs. Memory

- Given that
  - Registers: 32 words (128 Bytes)
  - Memory (DRAM): Billions of bytes (2 GB to 64 GB on laptop)

- and physics dictates...
  - Smaller is faster

- How much faster are registers than DRAM??
  - About 50-500 times faster! (in terms of latency of one access - tens of ns)
    - But subsequent words come every few ns
Jim Gray’s Storage Latency Analogy: How Far Away is the Data?

1.5 hr
Sacramento

1 min
My Head

100
Memory

1 [ns]
Registers

Jim Gray
Turing Award
B.S. Cal 1966
Ph.D. Cal 1969

Berkeley
UNIVERSITY OF CALIFORNIA

RISC-V (39)
C code

```c
int A[100];
g = h + A[3];
```

Using Load Word (lw) in RISC-V:

```asm
lw  x10,12(x15)  # Reg x10 gets A[3]
add x11,x12,x10  # g = h + A[3]
```

Note: x15 – base register (pointer to A[0])
12 – offset in bytes

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

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

- Using Store Word (sw) in RISC-V:
  ```
  lw  x10,12(x15)  # Temp reg x10 gets A[3]
  add x10,x12,x10 # Temp reg x10 gets h + A[3]
  ```

Note:  
- x15 – base register (pointer)
- 12,40 – offsets in bytes
- x15+12 and x15+40 must be multiples of 4
In addition to word data transfers (\texttt{lw}, \texttt{sw}), RISC-V has \textbf{byte} data transfers:

- load byte: \texttt{lb}
- store byte: \texttt{sb}

Same format as \texttt{lw}, \texttt{sw}

E.g., \texttt{lb x10,3 (x11)}

- contents of memory location with address = sum of “3” + contents of register \texttt{x11} is copied to the \underline{low byte position} of register \texttt{x10}.

\texttt{x10}: 
\begin{verbatim}
xxxx xxxx xxxx xxxx xxxx xxxx xxxx
\end{verbatim}

\texttt{...is copied to “sign-extend”}

This bit \texttt{zzzz zzzz} is copied to “sign-extend”.

RISC-V also has “unsigned byte” loads (\texttt{lbu}) which zero extends to fill register. Why no unsigned store byte ‘\texttt{sbu}?”
Example: What is in $x_{12}$?

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

Memory
Substituting \textit{addi}

The following two instructions:

\begin{verbatim}
lw x10,12(x15)  # Temp reg x10 gets A[3]
add x12,x12,x10 # reg x12 = reg x12 + A[3]
\end{verbatim}

Replace \textit{addi}:

\begin{verbatim}
addi x12, value # value in A[3]
\end{verbatim}

But involve a load from memory!

Add immediate is so common that it deserves its own instruction!
Decision Making
RV32 So Far…

- **Addition/subtraction**
  - add rd, rs1, rs2
  - sub rd, rs1, rs2

- **Add immediate**
  - addi rd, rs1, imm

- **Load/store**
  - lw rd, rs1, imm
  - lb rd, rs1, imm
  - lbu rd, rs1, imm
  - sw rs1, rs2, imm
  - sb rs1, rs2, imm
Based on computation, do something different
In programming languages: *if*-statement

RISC-V: *if*-statement instruction is

\[
\text{beq reg1, reg2, L1}
\]

means: go to statement labeled L1 if (value in reg1) == (value in reg2)
....otherwise, go to next statement

\[
\text{beq}
\]
stands for *branch if equal*

Other instruction: \[
\text{bne}
\] 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 (\texttt{beq}) or branch if not equal (\texttt{bne})
  - Also branch if less than (\texttt{blt}) and branch if greater than or equal (\texttt{bge})
  - And unsigned versions (\texttt{bltu}, \texttt{bgeu})

- Unconditional Branch – always branch
  - a RISC-V instruction for this: \texttt{jump (j)}, as in \texttt{j label}
Example *if* Statement

- Assuming translations below, compile *if* block
  
  \[
  f \rightarrow x_{10} \quad g \rightarrow x_{11} \quad h \rightarrow x_{12} \\
  i \rightarrow x_{13} \quad j \rightarrow x_{14}
  \]

  
  \[
  \text{if} \ (i == j) \quad \text{bne} \ x_{13},x_{14},\text{Exit}\\n  f = g + h; \quad \text{add} \ x_{10},x_{11},x_{12}
  \]

  
  \[
  \text{Exit:}
  \]

- May need to negate branch condition
Example *if-else* Statement

- Assuming translations below, compile

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

  \[\begin{align*}
  \text{if } (i == j) & \quad \text{bne } x_{13},x_{14},\text{Else} \\
  f & = g + h; \quad \text{add } x_{10},x_{11},x_{12} \\
  \text{else} & \quad j \text{ Exit} \\
  f & = g - h; \quad \text{Else: sub } x_{10},x_{11},x_{12} \\
  \text{Exit:}
  \end{align*}\]
Magnitude Compares in RISC-V

- 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) goto Label;`

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

Also “Branch on Greater or Equal” `bge` and `bgeu`
Note: No `bgt` or `ble` instructions
There are three types of loops in C:
- `while`
- `do ... while`
- `for`

Each can be rewritten as either of the other two, so the same branching method can be applied to these loops as well.

Key concept: Though there are multiple ways of writing a loop in RISC-V, the key to decision-making is conditional branch
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
add x11, x0, x0 # i
addi x13, x0, 20 # x13
Loop:
  bge x11, x13, Done
  lw x12, 0(x9) # x12 A[i]
  add x10, x10, x12 # sum+=
  addi x9, x9, 4 # &A[i+1]
  addi x11, x11, 1 # i++
  j Loop
Done:
```

Introduction (53)