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

RISC-V Assembly Language
Great Idea #1: Abstraction (Levels of Representation/Interpretation)

- **High Level Language Program (e.g., C)**
  - `temp = v[k];`
  - `v[k] = v[k+1];`
  - `v[k+1] = temp;`

- **Compiler**
  - `lw x3, 0(x10)`
  - `lw x4, 4(x10)`
  - `sw x4, 0(x10)`
  - `sw x3, 4(x10)`

- **Assembler**
  - `1000 1101 1110 0010 0000 0000 0000 0000`
  - `1000 1110 0001 0000 0000 0000 0000 0100`
  - `1010 1110 0001 0010 0000 0000 0000 0000`
  - `1010 1101 1110 0010 0000 0000 0000 0100`

- **Architecture Implementation**

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

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

---

```
A
\rightarrow
A
```

```
A
\rightarrow
B
```
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
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
Great Idea #3: Principle of Locality / Memory Hierarchy

- Extremely fast
- Extremely expensive
- Tiny capacity

Processor chip

CPU

Registers
Jim Gray’s Storage Latency Analogy: How Far Away is the Data?

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

This Campus

 Registers
1 [ns]

My Head 1 min

1

RISC-V (14)
Assembly Variables: Registers (2/3)

- 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)
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).
How do we do this?

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

Use intermediate temporary register

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

A good compiler may do:
Immediates

- 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 \( x_3, x_4 \) are associated with C variables \( f, g \)
- Syntax similar to add instruction, except that last argument is a number instead of a register.
Immediates

- 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
  - \textbf{addi} ..., \(-x = \text{“subi} \ldots, x” \Rightarrow \text{so no “subi”}
    \begin{align*}
    \text{addi } x_3, x_4, -10 & \quad \text{(in RISC-V)} \\
    f = g - 10 & \quad \text{(in C)}
    \end{align*}
  - where RISC-V registers \(x_3, x_4\) 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 (\(x_0\)) is ‘hard-wired’ to value 0; e.g.
  
  \[
  \text{add } x_3, x_4, x_0 \quad \text{(in RISC-V)}
  \]

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

  where RISC-V registers \(x_3, x_4\) are associated with C variables \(f, g\)

- Defined in hardware, so an instruction
  
  \[
  \text{add } x_0, x_3, x_4 \quad \text{will not do anything!}
  \]