CSC 411

Computer Organization (Spring 2022)
Lecture 5: ISAs and Basic RISC-V Programming

Prof. Marco Alvarez, University of Rhode Island

Quick notes

- From previous lectures
  - prefixes (base 10 and base 2)
  - understanding performance
  - response time, throughput, and speed-up
  - CPU clocking, CPI, and execution time
  - the power wall
  - benchmarking, geometric mean
  - amdahl's law

Quick notes

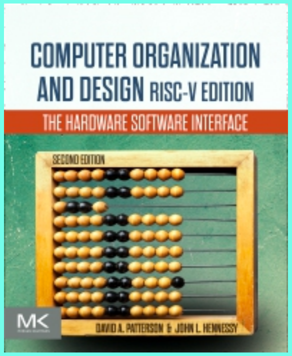
- Assignment 1 out (due Feb 21st)
  - problem set
  - …
- Required reading for next lectures
  - chapter 2 (P&H)

Disclaimer

Some of the following slides are adapted from:

Computer Organization and Design (Patterson and Hennessy)

The Hardware/Software Interface

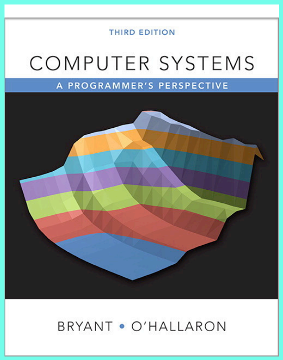


Disclaimer

Some of the following slides are copied from:

Computer Systems (Bryant and O'Hallaron)

A Programmer's Perspective



Instruction Sets

Abstraction layers (computing system)

	Problem
	Algorithm
	Program / Language
	System Software (OS, VM)
ISA (Instruction Set Architecture)	SW / HW Interface
Implementation of the ISA	Microarchitecture
Building blocks (gates)	Logic
	Devices
	Electrons

Levels of program code

- High-level language
  - level of abstraction closer to problem domain
  - provides for productivity and portability
- Assembly language
  - textual representation of instructions
- Hardware representation
  - binary digits (bits)
  - encoded instructions and data

High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Compiler

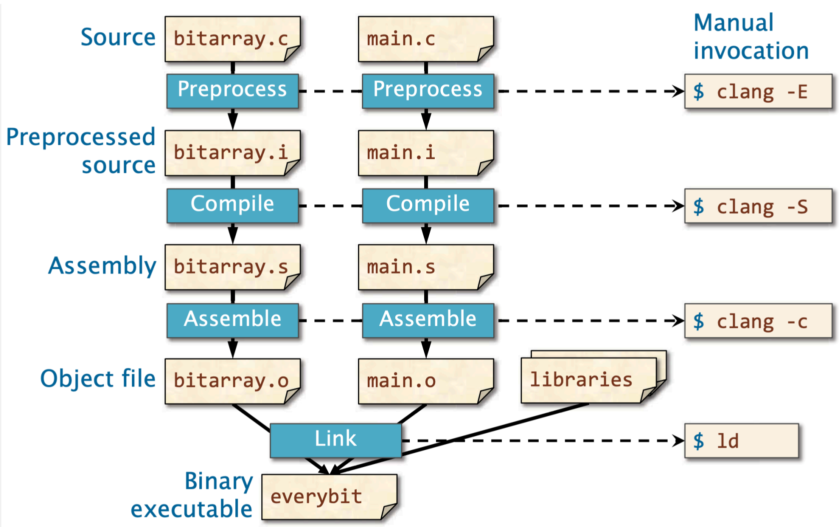
Assembly language program (for MIPS)

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```

Assembler

Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```



© 2008–2018 by the MIT 6.172 Lecturers

Instruction set

- The **repertoire** of instructions of a computer
  - different computers have different instruction sets
  - … with many aspects in common
  - the set of instructions a particular CPU implements is an **Instruction Set Architecture (ISA)**
- Many modern computers also have simple instruction sets
  - ARM (cell phones), Intel x86 (i9, i7, i5, i3), MIPS, RISC-V, …

ISA design principles

- Keep hardware simple
  - chip must only implement basic primitives and run fast
  - simplicity enables higher performance at lower cost
- Keep the instructions regular
  - regularity makes implementation simpler
    - simplifies decoding/scheduling of instructions

CISC vs RISC

- Design “philosophies” for ISAs
  - RISC vs. CISC
- Complex Instruction Set Computer (CISC)
  - X86, X86_64 (Intel and AMD, desktop/laptop/server)
  - X86* internally are still RISC
- Reduced Instruction Set Computer (RISC)
  - ARM (smartphone/pad)
  - **RISC-V** (free ISA, closer to MIPS than other ISAs)
  - Others: Power, SPARC, etc

CISC vs RISC

- Early trend was to add more and more instructions
  - VAX architecture had an instruction to multiply polynomials
- RISC philosophy
  - keep the instruction set small and simple
  - makes it easier to build fast hardware.
  - let software do complicated operations by composing simpler ones

ACM has named **John L. Hennessy**, former President of Stanford University, and **David A. Patterson**, retired Professor of the University of California, Berkeley, recipients of the 2017 ACM A.M. Turing Award for pioneering a systematic, quantitative approach to the design and evaluation of computer architectures with enduring impact on the microprocessor industry.

https://www.youtube.com/watch?v=3LVeEjsn8Ts

Industry Context and ISAs

Intel x86 Processors

- **Dominate laptop/desktop/server market**
- **Evolutionary design**
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
- **x86 is a Complex Instruction Set Computer (CISC)**
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
- **Compare: Reduced Instruction Set Computer (RISC)**
  - RISC: *very few* instructions, with *very few* modes for each
  - RISC can be quite fast (but Intel still wins on speed!)
  - Current RISC renaissance (e.g., ARM, RISCV), especially for low-power

x86 Clones: Advanced Micro Devices (AMD)

- **Historically**
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- **Then**
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension to 64 bits
- **Recent Years**
  - Intel got its act together
    - 1995-2011: Lead semiconductor "fab" in world
    - 2018: #2 largest by $$ (#1 is Samsung)
    - 2019: reclaimed #1
  - AMD fell behind
    - Relies on external semiconductor manufacturer GlobalFoundaries
    - ca. 2019 CPUs (e.g., Ryzen) are competitive again

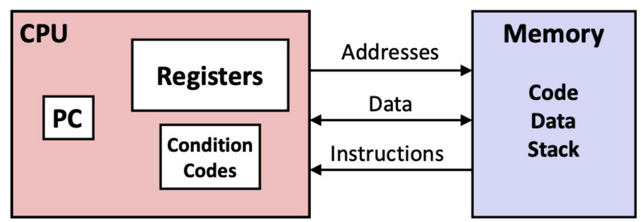
Intel's 64-Bit History

- **2001: Intel Attempts Radical Shift from IA32 to IA64**
  - Totally different architecture (Itanium, AKA "Itanic")
  - Executes IA32 code only as legacy
  - Performance disappointing
- **2003: AMD Steps in with Evolutionary Solution**
  - x86-64 (now called "AMD64")
- **Intel Felt Obligated to Focus on IA64**
  - Hard to admit mistake or that AMD is better
- **2004: Intel Announces EM64T extension to IA32**
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!
- **Virtually all modern x86 processors support x86-64**
  - But, lots of code still runs in 32-bit mode

Definitions

- **Architecture: (also ISA: instruction set architecture) The parts of a processor design that one needs to understand for writing correct machine/assembly code**
  - Examples: instruction set specification, registers
  - **Machine Code**: The byte-level programs that a processor executes
  - **Assembly Code**: A text representation of machine code
- **Microarchitecture: Implementation of the architecture**
  - Examples: cache sizes and core frequency
- **Example ISAs:**
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Used in almost all mobile phones
  - RISC V: New open-source ISA

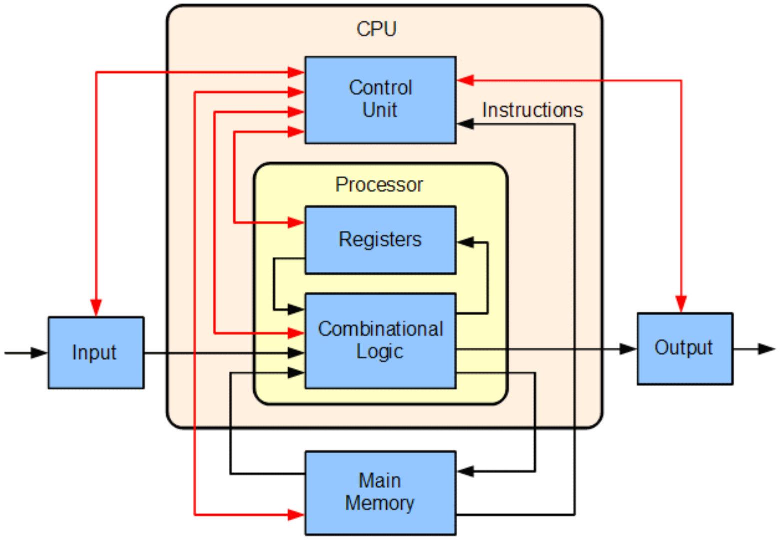
Assembly/Machine Code View



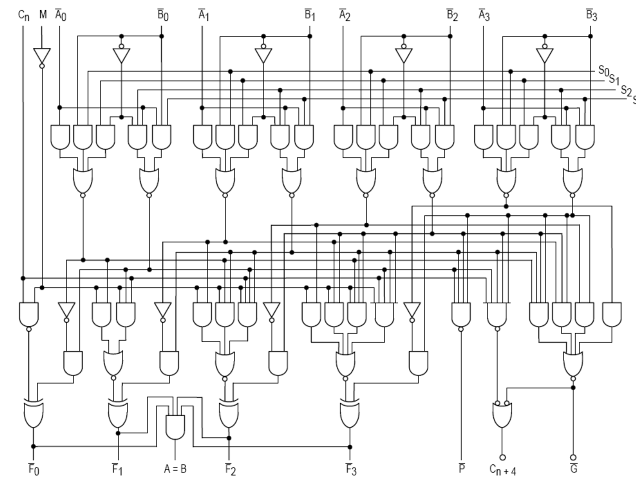
Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called "RIP" (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures



https://en.wikipedia.org/wiki/Central_processing_unit

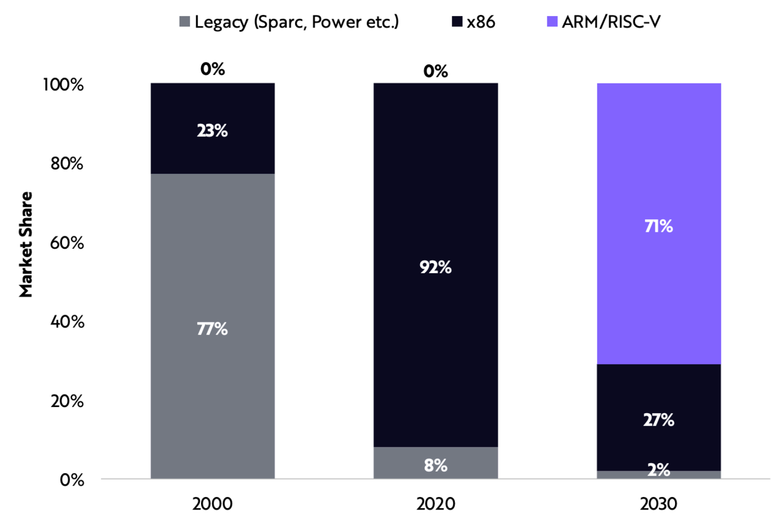


The 74181 is a 7400 series medium-scale integration (MSI) TTL integrated circuit, containing the equivalent of 75 logic gates[2] and most commonly packaged as a 24-pin DIP. The 4-bit wide ALU can perform all the traditional add / subtract / decrement operations with or without carry, as well as AND / NAND, OR / NOR, XOR, and shift. The 74181 represents an evolutionary step between the CPUs of the 1960s, which were constructed using discrete logic gates.

https://en.wikipedia.org/wiki/74181

RISC-V Basics

The Data Center Architecture



Big Ideas 2021, ARK Investment Management LLC

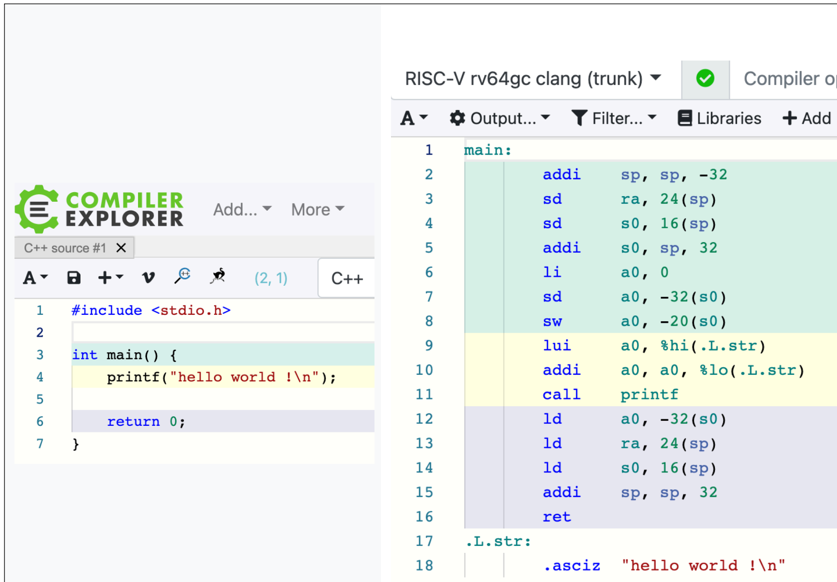
Looking into an executable

```c
#include <stdio.h>

int main() {
    printf("hello world !\n");

    return 0;
}
```

```
$ objdump -d
```



https://godbolt.org/

The RISC-V instruction set

- Developed at UC Berkeley as open ISA
  - now managed by the RISC-V Foundation (riscv.org)
- Similar ISAs have a large share of embedded core market
  - applications in consumer electronics, network/storage equipment, cameras, printers, …
- Used as the example throughout this course

Arithmetic operations

- Three operands
  - one destination and two sources
  - **all arithmetic operations** have this form

```
add a, b, c       // a gets b + c
```

- How would you translate the following C code?

```c
a = b + c + d + e;
```

  - single line of C converts into multiple lines in Assembly
  - some sequences are better than others (temporary variables?)

Arithmetic example

- C code

```
f = (g + h) - (i + j);
```

- Assembly code

```
add t0, g, h     // temp t0 = g + h
add t1, i, j     // temp t1 = i + j
sub f, t0, t1    // f = t0 - t1
```

Register operands

- Arithmetic instructions use **register** operands
- RISC-V has a 32×64-bit **register file** (RV64 variant)
  - use for frequently accessed data
  - 32x64-bit general purpose registers numbered x0 to x31
  - `x5–x7, x28–x31` for temporaries
  - `x9, x18–x27` for saved registers
- Words
  - 64-bit data is called a "***doubleword***"
  - 32-bit data is called a "***word***"

RISC-V registers

Register name	Symbolic name	Description	Saved by
32 integer registers			
x0	Zero	Always zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary / alternate return address	Caller
x6–7	t1–2	Temporary	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function argument / return value	Caller
x12–17	a2–7	Function argument	Caller
x18–27	s2–11	Saved register	Callee
x28–31	t3–6	Temporary	Caller

https://en.wikipedia.org/wiki/RISC-V

RISC-V registers

32 floating-point extension registers			
f0–7	ft0–7	Floating-point temporaries	Caller
f8–9	fs0–1	Floating-point saved registers	Callee
f10–11	fa0–1	Floating-point arguments/return values	Caller
f12–17	fa2–7	Floating-point arguments	Caller
f18–27	fs2–11	Floating-point saved registers	Callee
f28–31	ft8–11	Floating-point temporaries	Caller

https://en.wikipedia.org/wiki/RISC-V

Example with register operands

- C code

```
f = (g + h) - (i + j);
```

- Compiled RISC-V code

```
// assume f,...,j in x19,...,x23

add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

- Can you do it differently?

In fact, if the variables are floating-point values, different sequences of instructions may produce slightly different results. Floating-point operations are not necessarily associative or commutative …

… stay tuned

Immediate operands

- Constant data specified in an instruction

```
addi x3, x4, 4
```

- No subtract immediate instruction
  - there are `add` and `sub` instructions but only `addi` for immediate operands
  - just use a negative constant

```
addi x3, x4, -1
```

- Immediate operands
  - immediate operand avoids a load instruction

The constant zero

- Register x0 is the constant 0
- Defined in hardware
  - **cannot be overwritten**
- Useful for common operations
  - e.g., move between registers

```
add x3, x4, x0
```