# COM SCI M151B Week 1

Aidan Jan

October 1, 2024

Using Abstraction

- We see a computer as a *box* with multiple layers of **abstraction**.

- Depending on which layer we want ot work on, we *abstract away* the irrelevant layers.

  - The *main benefit* is that we don't need to know the unnecessary details of the other layers in order to be able to work on our layer.

Computer Abstractions (simplified)
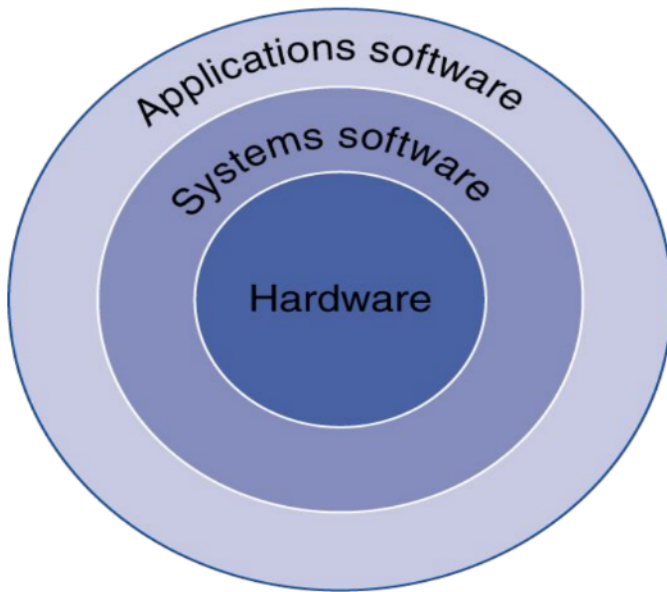
- Application software

  - Translation from algorithm to code
  - Written in high-level language (e.g., C, Java)

- System software

  - Compiler: translates high level language code to machine code
  - Operating system: service code
    * Handling input/output
    * Managing memory and storage
    * Scheduling tasks and sharing resources

- Hardware

  - Processor memory
  - I/O controllers

  `https://www.youtube.com/watch?v=_y-5nZAbgt4`

How do computers work?

- Theoretical and Historical Points of View

  - Turing machines and history of electronics and computers.

Level of Program Code

- High level language

  - Level of abstraction closer to problem domain
  - Provides for productivity and portability

- Assembly language

  - Textual representation of instructions
  - Architecture-dependent

- Hardware representation

  - Binary digits (bits)
  - Encoded instructions and data

How to maintain compatibility?

There are many different types of computer architecture, and many different languages applications are written in. How do we maintain compatibility?

- System software (OS) and software makes a contract to always give a program with **only a set of known instructions**, and the hardware promises to be able to run that.

- The **ISA** is the *interface* between hardware and software. This allows the HW and SW to change/evolve **independently**.

- Software sees:

  - Function description of hardware:
    1. Storage locations (e.g., memory)
    2. Operations (e.g., add)

- Hardware sees:

  - List of instructions and their order

- In this course, we will use RISC-V ISA.

Goals when designing computers

They must be efficient. What is *efficient*?

- Performance (often most important)

- Power consumption

- Cost

- Reliable and Secure

Past, Present, and Future

- Moore's Law:

  - We started with the ENIAC after World War II, and that computer had a size of 1000 cubic feet, used 125kW of power, only does 2000 additions per second, and had 48 kB of memory.

  - Now, we have computers taking up 1 cubic foot, consumes 250W of power, capable of 20B operations per second, with multiple GB of memory, for only less than 0.01% of the cost of the ENIAC.

  - Moore's Law states that every two years, the number of transistors (and therefore computational power) doubles. This suggests that the cost per transistor reduces each year.

*Plot was taken from Hennessy Patterson Book [1].
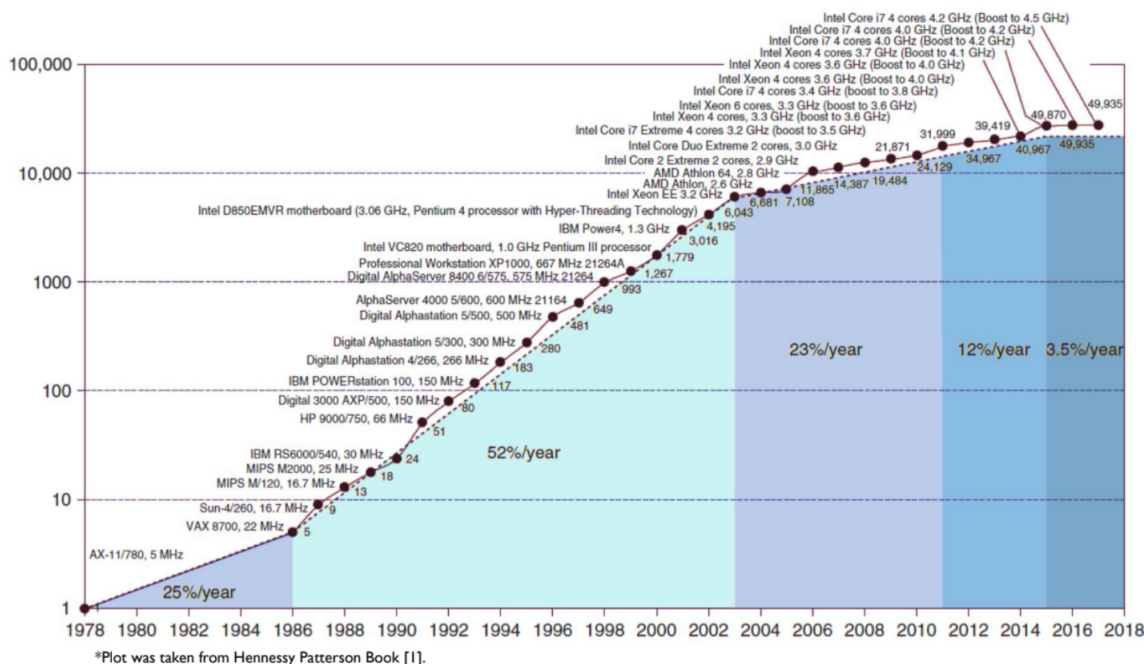
  - From 1986 to 2003, Moore's Law was theorized
  - At 2003 to 2011, transistors grew so close together that it became near-impossible to make them smaller; refer to Power Wall below.
  - From 2011 to 2015, Amdahl's Law started taking effect. (Refer below).
  -

- Dennard's Scaling Law:

  - According to Moore's law, the number of transistors on a chip doubles every two years, thus each transistor's area is reduced by 50%, or every dimension by 0.7x.

- As a result, voltage is reduced by -30% (0.7x) to keep the electric field constant. $V = EL$
- L is reduced, thus delays are reduced by -30%. ($x = Vt$)
- Frequency is increased by +40% ($f = 1/t$)
- Capacitance is reduced by -30% ($C = kA/L$)

- Scaling Power and Energy

$$P = CV^2 f$$

- Power consumption per transistor is decreased by -50%.
- Power consumption of the entire chip stays the same! Except now it has more transistors.

- Where are these improvements coming from?

  1. Advancements in microelectronics and fabrication technologies.
  2. Advancements in architectural techniques
     - What this course is about!
     - This lead to an improvement by a factor of 25 versus if we had only relied on (1.)

- Power Wall

  - Up to 2005, manufacturers could double the processor performance while keeping the power constant.
  - Since 2005, due to smaller transistor sizes, **static power leakage** becomes so dominant that the power consumption did not stay constant. (Moore's Law slowed down.)
  - However, the rate still grew since multicore systems began to emerge.

- Amdahl's Law

  - Performance improvement (speedup) is limited by the part you cannot improve (called "sequential part").

  $$\text{speed up} = \frac{1}{(1 - p) + \frac{p}{s}}$$

    * P is the part that can be improved (e.g., **parallelized**)
    * S is the factor for improvement (e.g., more cores for parallelization)

- Dark Silicon

  - Where we are right now
  - Due to thermal management limitations, some parts of a chip cannot be turned on.
  - This is the end of the multi-core era.
  - New technologies and techniques are being used to continue the scaling. We need cross-stack approaches!

Instruction Set Architecture (ISA)

What is an ISA

- The contract between software and hardware is the ISA. Typically described by giving all the programmer-visible state (register + memory) plus the semantics of the instructions that operate on that state.

- IBM 360 was the first line of machines to separate ISA from implementation (aka. *microarchitecture*)

- There are **many implementations possible** for a given ISA

- AMD, Intel, VIA processors run the AMD64 ISA
- Many cell phones use the ARM ISA with implementations from many different ompanies including Apple, Qualcomm, Samsung, Huawei, etc.

- We use RISC-V as the standard ISA in class.

Design Methodology

- ISA often designed with particular microarchitectural style in mind, e.g.,

  - Accumulator $\rightarrow$ hardwired, unpipelined
  - CISC $\rightarrow$ microcoded
  - RISC $\rightarrow$ hardwired, pipelined
  - VLIW $\rightarrow$ fixed-latency in-order parallel pipelines
  - JVM $\rightarrow$ software interpretation

- But can be implemented with any microarchitectural style

  - Intel Ivy Bridge: hardwired pipelined CISC (x86) machine (with some microcode support)
  - Spike: Software-interpreted RISC-V machine
  - ARM Jazelle: A hardware JVM processor

RISC-V

What is RISC-V?

- An *open-source* "RISC"-based ISA (*royalty-free*)

- Developed in the 2010s at Berkeley

- Mostly maintained by the open-source community.

- Different extensions and models. We will focus on the 32-bit "base" mode (i.e., "RV321").

- RISC = "Reduced Instruction Set Computers". Unlike "CISC", every instruction in RISC is the same size.

RISC vs. CISC

- Instruction size of RISC is fixed, CISC is variable

- RISC has simple (one-by-one) operations, rather than packed operations

- RISC is less complex than CISC in terms of instruction set.

Widely-Used ISAs

- All "RISC" except x86 (Intel's ISA)

- Most popular RISC ISAs:

  - MIPS, ARM, PowerPC, and RISC-V

Stored Program Computer (von Neumann)

- Computer hardware is a machine that reads instructions one-by-one and executes them sequentially. It continues this until the program finishes.

- Memory holds *both* program and data

  – Instructions and data in a linear memory array
  – Instructions can be modified as data

- Sequential instruction processing

  1. Program Counter (PC) identifies current instruction.
  2. Fetch instruction from memory.
  3. Update state (e.g., PC and memory) as a function of current state according to instruction
  4. Repeat.

How to build an ISA?

- Transition from HLL to assmebly

- Take CS132 (Compiler Construction)

What do instructions look like?

There are two main parts: commands and operands.

- In 32-bit RISC-V, each instruction is fixed-size and is 32-bit.

- Possible Types of Operands:

  1. Registers

     – A small storage unit *inside* the processor to quickly access data.
     – Faster than memory, but much smaller (smaller is faster!)
     – Can be seen by Software*! (This is something that we will clarify later.)
     – Typically between 16 and 64 registers in modern ISAs.
        * Larger width means easier data transfer but more power.
        * More registers means less access to the main memory, but requires more area and more power.
        * Low-end processors use smaller registers. High-Performance processors use wider and more registers.
     – See section on RISC-V Registers

  2. Immediate

     – Constant numbers to be used in an instruction
     – Example: `addi x2, x1, 5`
     – Registers are 32-bits but immediates may not. (This is because they must fit into the instruction. Use pointer instead if it doesn't fit!)
     – See section on RISC-V Immediates.

  3. Memory

     – Values are stored in a memory and could be accessed.
     – Memory shouldbe byte-addressable
     – See section on Memory.

RISC-V Registers

- RISC-V (RV32) uses 32 registers, 32-bit each (called "word").

- RV64 uses 32 registers, 64-bit each (called double-word)

- Each register shown as xi.

- $x0$ is hardwired to zero.

- Registers are stored in a data structure called the **Register File** (this is a hardware unit that will be discussed later.)

- For more high-end processors, there is a separate set of registers for floating point operations.

RISC-V Immediates

- RISC-V does Sign-Extension and Padding

- Sign-Extension

  - Two's Complement for signed values

- Padding

  - For LSBs (least significant bit), padding zeros is sufficient.

Memory

Format and addressing

- Byte-addressability vs. 32-bit data (little endian vs. big endian)

- Alignment:

MSB	byte-3	byte-2	byte-1	byte-0	LSB
	byte-7	byte-6	byte-5	byte-4	
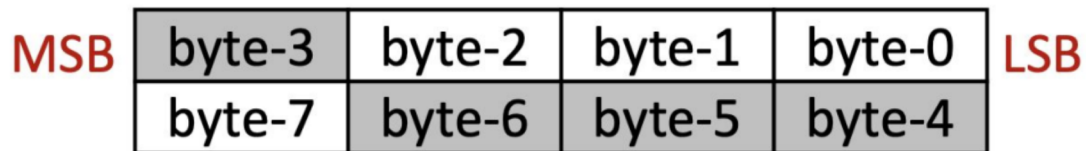
- Load and Store Variants

  - LW, LB, LH (load different amounts of bits - word, byte, high) (this one does sign extension, e.g., adding 1's instead of 0's if number is negative)

  - LBU, LHU (byte unsigned, high unsigned) (this one does not do sign extension, which means higher bits are unaltered)

  - SW, SB, SH (store word, byte, high - doesn't care about the sign)

Instruction Types

1. Arithmetic/ALU

   - Do an arithmetic operation on two registers or one register and an immediate

   - Result is saved in another register

   - Example: `ADDI x5, x1, 10` - adds `x1` and 10, stores in `x5`

2. Memory

- Load and Store
- Addressing modes:
  - Base (register) + Offset (immediate)
  - Base could be zero or it could be a value stored in a register.
- Data Size and format
  - Word, half-word, byte
  - Signed and unsigned

3. Control-Flow

RISC-V Arithmetic operators

ADDI	addi rd, rs1, constant	Add Immediate	reg[rd] <= reg[rs1] + constant
SLTI	slti rd, rs1, constant	Compare < Immediate (Signed)	reg[rd] <= (reg[rs1] $<_s$ constant) ? 1 : 0
SLTIU	sltiu rd, rs1, constant	Compare < Immediate (Unsigned)	reg[rd] <= (reg[rs1] $<_u$ constant) ? 1 : 0
XORI	xori rd, rs1, constant	Xor Immediate	reg[rd] <= reg[rs1] ^ constant
ORI	ori rd, rs1, constant	Or Immediate	reg[rd] <= reg[rs1] | constant
ANDI	andi rd, rs1, constant	And Immediate	reg[rd] <= reg[rs1] & constant
SLLI	slli rd, rs1, constant	Shift Left Logical Immediate	reg[rd] <= reg[rs1] « constant
SRLI	srli rd, rs1, constant	Shift Right Logical Immediate	reg[rd] <= reg[rs1] $»_u$ constant
SRAI	srai rd, rs1, constant	Shift Right Arithmetic Immediate	reg[rd] <= reg[rs1] $»_s$ constant
ADD	add rd, rs1, rs2	Add	reg[rd] <= reg[rs1] + reg[rs2]
SUB	sub rd, rs1, rs2	Subtract	reg[rd] <= reg[rs1] - reg[rs2]
SLL	sll rd, rs1, rs2	Shift Left Logical	reg[rd] <= reg[rs1] « reg[rs2]
SLT	slt rd, rs1, rs2	Compare < (Signed)	reg[rd] <= (reg[rs1] $<_s$ reg[rs2]) ? 1 : 0
SLTU	sltu rd, rs1, rs2	Compare < (Unsigned)	reg[rd] <= (reg[rs1] $<_u$ reg[rs2]) ? 1 : 0
XOR	xor rd, rs1, rs2	Xor	reg[rd] <= reg[rs1] ^ reg[rs2]
SRL	srl rd, rs1, rs2	Shift Right Logical	reg[rd] <= reg[rs1] $»_u$ reg[rs2]
SRA	sra rd, rs1, rs2	Shift Right Arithmetic	reg[rd] <= reg[rs1] $»_s$ reg[rs2]
OR	or rd, rs1, rs2	Or	reg[rd] <= reg[rs1] | reg[rs2]
AND	and rd, rs1, rs2	And	reg[rd] <= reg[rs1] & reg[rs2]

Memory-Type Instructions

LB	lb rd, offset(rs1)	Load Byte	reg[rd] <= signExtend(mem[addr])
LH	lh rd, offset(rs1)	Load Half Word	reg[rd] <= signExtend(mem[addr + 1: addr])
LW	lw rd, offset(rs1)	Load Word	reg[rd] <= mem[addr + 3: addr]
LBU	lbu rd, offset(rs1)	Load Byte (Unsigned)	reg[rd] <= zeroExtend(mem[addr])
LHU	lhu rd, offset(rs1)	Load Half Word (Unsigned)	reg[rd] <= zeroExtend(mem[addr + 1: addr])
SB	sb rs2, offset(rs1)	Store Byte	mem[addr] <= reg[rs2][7:0]
SH	sh rs2, offset(rs1)	Store Half Word	mem[addr + 1: addr] <= reg[rs2][15:0]
SW	sw rs2, offset(rs1)	Store Word	mem[addr + 3: addr] <= reg[rs2]