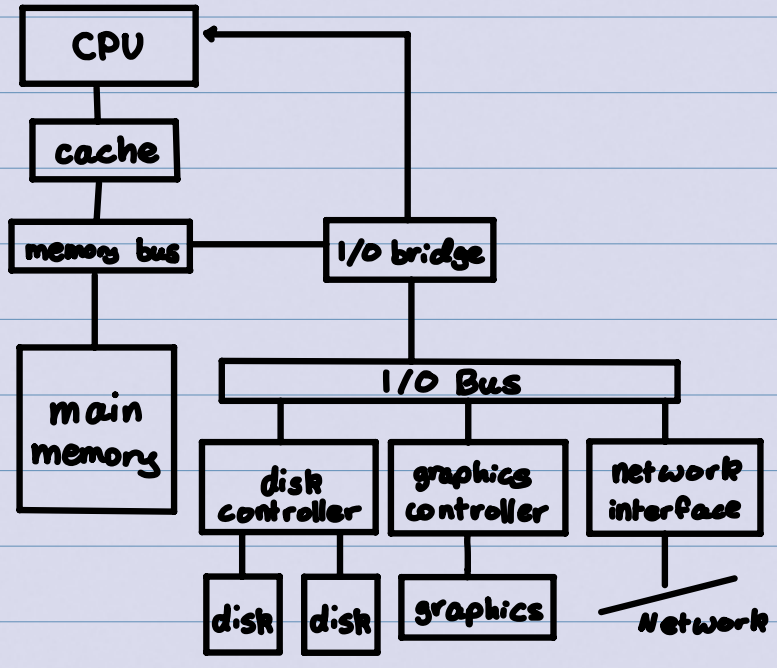
Lecture 1 - 4th Sept 2024

Instruction Set Architecture: how **hardware executes software**

- **What is a computer?**
  - ↳ just a machine that does whatever the software tells it to do.
  - ↳ computer > digital circuits > logic gates > transistors
  - ↳ software is just a series of instructions.
  - ↳ The five classic components of a computer:
    - CPU: Central Arithmetic (ALU) and Central Control
    - Memory System
    - Input and Output

↳ But, here's a more realistic view:



Architecture vs Microarchitecture

↳ **Processor architecture:**

- Functional appearance to software (in this case, compilers!) (ISA)
  - Exactly what instructions does it have?
  - Number of memory/storage locations it has
  - **Interface!**

↳ **Processor microarchitecture:**

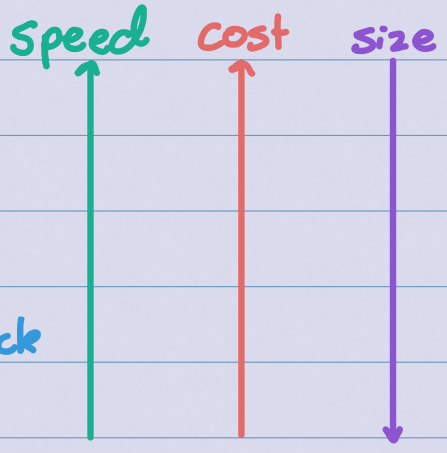
- **Logical structure that implements the architecture**
  - Number of functional units, interconnection, control
  - Size of the caches
  - Not visible to the software
  - **Implementation!**

Memory Hierarchy:

- CPU
- CPU Cache

↗ processor

- Physical Memory → RAM
- Solid State Memory → SSD/USB stick
- Virtual Memory → hard drive

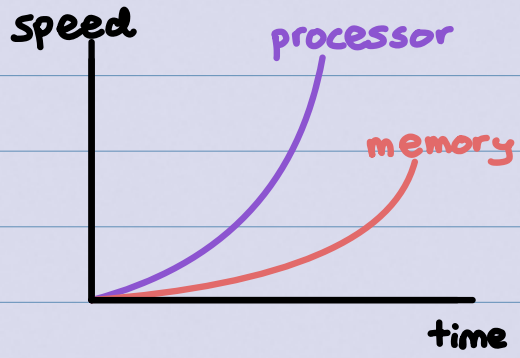


- **Moore's Law:** every **18-24 months,**
  - **2x transistors** on same chip area
  - **2x processor speed**
  - **2x memory capacity**
  - **½ energy/power consumption.**

↳ technically not a law, but it's a **self-fulfilling prophecy.**

Memory Wall: every 2 years,

- 2x Instructions / second
- 2x memory capacity
- 1.1x memory latency



↳ growing disparity between processor and memory performance!
∵ significant effort in reducing / hiding memory latency.

Latency: time from start to finish (aka response time).

Throughput: number of tasks completed per time unit (aka bandwidth)

↳ throughput can exploit parallelism, but latency cannot!

- Improving the latency of a component always improves the overall system throughput.

$$\text{Speedup} = \frac{\text{Performance}(x)}{\text{Performance}(y)}.$$

"x is "speedup" times faster than y"

↳ can be broken down into $\frac{\text{throughput}(x)}{\text{throughput}(y)}$ or $\frac{\text{latency}(y)}{\text{latency}(x)}$.

don't forget - latency will likely be < 1, so need to invert fraction!

Iron Law of Performance:

$$\text{CPU Time} : \frac{\text{instructions}}{\text{program}} \cdot \frac{\text{cycles}}{\text{instruction}} \cdot \frac{\text{seconds}}{\text{cycle}}$$

aka CPI

IPC is inverse - instructions per cycle

Amdahl's Law: $\text{Speedup} = \dfrac{1}{(1 - \text{Frac}_{EHN}) + \dfrac{\text{Frac}_{EHN}}{\text{Speedup}_{EHN}}}$

Example:

- Enhancement 1: speedup of 20 on 10% of time

  ↳ $\dfrac{1}{(1-0.1) + \frac{0.1}{20}} = \dfrac{1}{0.9 + 0.005} = 1.105.$

- Enhancement 2: speedup of 1.6 on 80% of time.

  ↳ $\dfrac{1}{(1-0.8) + \frac{0.8}{1.6}} = \dfrac{1}{0.2 + 0.5} = \dfrac{1}{0.7} = 1.43.$

∴, the second enhancement is better → even though the first enhancement has a speedup of 20, it's only helpful 10% of the time.

↳ "make the common case fast" !!

Power Consumption: transistors consume / dissipate power in two ways:

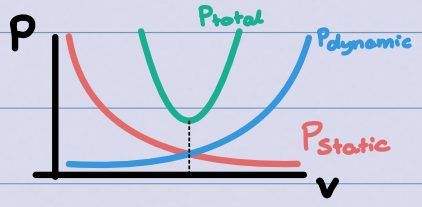
- **Dynamic Power:** Consumed by activities in a circuit (switching transistors)

  ↳ $P = \frac{1}{2} C \cdot V^2 \cdot f \cdot a$

  where: C = capacitance (~ chip area)
  V = power supply voltage
  f = clock frequency
  a = activity factor

- **Static Power**: consumed when powered on but idle (leaking current)
  - ↳ as voltage decreases, leakage increases.

- we strive for the "sweet spot" between static and dynamic power:



- **Power Wall**: supply voltage (generally) decreasing over time.
  - ↳ emphasis on power starting ~2000.
    - Since ~2000, we've "hit a wall" and aren't making as much progress.

- **What is an ISA?**
  - ↳ Functional and precise specification of a computer.

- An ISA is a "contract" between the software and the hardware.
  - Specifies what hardware promises to do when it sees certain instructions, but not how it does it.

CISC vs RISC:

- Early trend was to add more and more instructions to new CPUs to do elaborate operations
  - ↳ CISC (Complex Instruction Set Computing)

- Now, we keep the instruction set small and simple and let the software do the complicated operations by composing simpler ones.
- This makes it easier to build fast hardware.
  ↳ RISC (Reduced Instruction Set Computing)

- Instruction Set:
  - Arithmetic and logic: add, subtract, multiply, divide, AND, OR, XOR, ...
  - Data movement: move, load, store
  - Control flow: branch and jump
  - System (privileged): used to manage processor state, handle exceptions, etc.
  - Each instruction has a specific format and encoding

Registers
- General-Purpose Registers: can be used for various purposes as determined by the programmer or compiler.
- Special-Purpose Registers: have specific roles, such as:
  ↳ Program Counter (PC): holds the address of the next instruction to be executed.
  ↳ Stack Pointer (SP): points to the top of the stack in memory
  ↳ Status Register: holds flags that indicate the results of operations (eg, zero flag).

Memory Model:

this is virtual memory, not physical!

- **Address Space:** defines the **range of memory locations** that can be addressed
  - ↳ **Eg:** a 32-bit address space can address $2^{32} = 4\,GB$.
- **Byte Ordering:** specifies whether multi-byte values are stored with the most **significant byte first** (**big-endian**) or **last** (**little-endian**)
- **Alignment Requirements:** some ISAs require **data to be aligned** in memory **in specific ways.**
  - ↳ **Eg:** 32-bit values may need to be stored at addresses that are multiples of four.

Addressing Modes:

- **Immediate:** the **operand** is included **directly** in the instruction itself.
  - ↳ often used for **small constants** and **quick arithmetic operations.**
- **Register:** the **operand** is stored in a CPU register
  - ↳ often used for **frequently-accessed variables** and **intermediate results.**
- **Direct:** the **instruction contains the memory address** where the operand is located.
  - ↳ often used for accessing **static variables** or **fixed memory locations.**
- **Indirect:** the **instruction specifies a register that contains the memory address** of the operand
  - ↳ often used for accessing **pointer-based structures.**

RISC-V

- New open-source and license-free ISA spec.
- Appropriate for all levels of computing system, from microcontrollers to supercomputers.
- Simple and elegant!
- Designed with modularity and extensibility in mind
- Many optional extensions targetting different use cases:
  - M: integer multiplication and division
  - A: atomic instructions
  - F: single-precision floating point
  - D: double-precision floating point
  - C: compressed instructions
  - V: vector operations
  - ↳ Eg: "RV64MC" would be a RISC V 64-bit implementation with the extensions M and C.

- There are 32 registers in RISC-V, numbered from 0 to 31: x0 - x31
  - ↳ x0 is special, it always holds the value zero.
- Each register is 32-bits wide ↝ note, 32 bits is called a "word"
- Some registers have conventional uses:
  - ↳ x1: return address (ra)
    x2: stack pointer (sp)
    x10-x17: function arguments/return values (a0-a7)

- Since registers are close to the processor, they're very fast (faster than 0.25 ns)!

RISC-V arithmetic instruction syntax:

opname, rd, rs1, rs2

opname: operation, by name
rd: "destination", operand getting result
rs1: "source 1", 1st operand for operation
rs2: "source 2", 2nd operand for operation

Eg: add, x1, x2, x3
↳ a = b + c, where a is stored at register 1, b in register 2, and c in register 3.

Example: how would we implement the following statement: a = b + c + d - e? a is x10, b is x1, c is x2, d is x3, and x4.

↳
```
add  x10, x1, x2     # a = b+c
add  x10, x10, x3    # a = a+d
sub  x10, x10, x4    # a = a-e
```

Example: f = (g+h) - (i+j); , where f is x19, g is x20, h is x21, i is x22, and j is x23.
↳ we'll have to use intermediate temporary registers!

↳
```
add  x5, x20, x21    # temp1 = g+h
add  x6, x22, x23    # temp2 = i+j
```

sub x19, x5, x6 # f = temp1 - temp2

Immediates: numerical constants

↳ **add immediate instruction**: similar to add instruction, but the last operand must be a number, not a register:

addi rd, rs1, number

↳ Eg: add 6 to the value in register x3.

↳ addi x3, x3, 6 # x3 = x3 + 6.

↗ RISC philosophy is to reduce the operations to an absolute minimum!

- There is no subtract immediate instruction in RISC-V → just pass a negative number!

- **Register zero (x0)**: RISC-V hardwires the register 0 (x0) to value 0.

↳ Eg: add x3, x4, x0 : f = g

Eg: addi x3, x0, 0xff : f = 0xff

- ∴, the instruction add x0, x1, x2 won't do anything!

- Also, x0 is useful for clearing a register or negating a number.

RISC-V is a load-store architecture!

↳ Load-store (aka register-register) architecture: memory access is limited to load and store instructions. All other instructions (arithmetic, logical, etc) operate only on registers.

RISC-V Load Instructions Syntax:

```
l[size] rd, imm(rs1)
```

l: stands for load

[size]: specifies the size of the data to load

↳ w for word, h for halfword, b for byte

rd: destination register

imm: immediate offset value

rs1: the base register containing a memory address

Example: translate the following code to RISC-V:

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

g ↑ 12 bits into A → aka A[3] ↑ address where A is stored

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

RISC-V Store Instructions Syntax:

```
S[size] rs2, imm(rs1)
```

S: stands for store

[size]: specifies the size of the data to store

↳ w for word, h for halfword, b for byte

rs2: the source register containing the data to store

imm: an immediate offset value

rs1: the base register containing a memory address

Example: translate the following code to RISC-V:

```
int A[100];
A[10] = h + A[3];
```

```
lw   x10, 12(x15)      # temp gets A[3]
add  x10, x12, x10     # temp = h + temp = h + A[3]
sw   x10, 40(x15)      # store temp in A[10]
```

(temp = x10, A[3] = 12(x15), register of A = x15; h = x12, temp = x10; temp = x10, 40 bits into A aka A[10], register of A = x15)

Loading and Storing Bytes:

- Using lb and sb.
- the most significant bit of the byte is extended to fill the rest of the word → aka "sign-extend" the byte.

↳ Example: what does lb x10, 3(x11) do?

↳ contents of memory location with address = 3 + contents of register x11 is copied to the low byte position of register x10.

Example: what will be in x12 after these instructions?

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

1) x11 becomes 100100031F5 → this is in hex for simplicity
↳ each 2 digits represents a byte
↳ 4 bytes is a word!

2) x5 becomes x11 = 100100031F5
sign-extended → skip first byte, because 1(x5)

3) x12 becomes 100100100031

Pseudoinstructions: instructions that are not actually implemented in hardware but are recognized by the assembler and translated into one or more real hardware instructions.

Decision making: based on computation, do (or don't do) something different. Eg: if-else
↳ Assembly instructions support decision making called branch.

- **Branch:** change of control flow

- **Conditional Branch:** change control flow based on outcome of comparison.
  ↳ beq, bne, blt(u), bgt(u)

- **Unconditional Branch:** always branch
  ↳ j, jal, jr

RISC-V Conditional Branch Instructions Syntax:

b[cond] rs1, rs2, L

b: stands for **branch**

[cond]: specifies the **condition** of the branch.

↳ eq for equal, ne for not equal, etc.

rs1, rs2: registers containing **the 2 operands for comparison.**

L: a **label** (symbolic address for another position in the code)

Example: translate the following code to RISC-V instructions:

```
if (i==j){
    f=g+h; }
```

, assume i → x13, j → x14, f → x10,
g → x11, h → x12.

```
    bne  x13, x14, Exit     # if (i≠j), branch to Exit
    add  x10, x11, x12      # ∴, i=j, so f=g+h
Exit:                       # Exit label with nothing after it.
```

↳ note: it's common to need to negate the if statement.

Example: translate the following code to RISC-V instructions:

```
if (i==j) { f=g+h; }
else { f=g-h }
```

, assume i → x13, j → x14, f → x10,
g → x11, h → x12.

```
    bne  x13, x14, Else     # if (i≠j), branch to Else
    add  x10, x11, x12      # ∴ i=j, so f=g+h
    j    Exit               # end if i=j
Else: sub x10, x11, x12     # ∴ i≠j, so f=g-h
Exit:                       # end
```

- General programs **need to test < and >** as well

RISC-V Magnitude Comparison Instruction Syntax:

```
blt reg1, reg2, L
```

blt: stands for **branch on less than** — if (reg1 < reg2)

reg1, reg2: registers containing the **operands** to compare

L: **label** to branch to

- There is also **bltu**, which acts **exactly like blt**, but **does not consider negative numbers.**
  - ↳ **bltu:** **branch on less than unsigned.**

Loops in Assembly: **while** / **do ... while** / **for**

Example: translate the C code to RISC-V instructions:

```
int A[20];
int sum = 0;
for (int i = 0; i < 20 ; i++) { sum += A[i]; }.
```

Assume x8 is A[0], x10 is sum, x11 is i

```
  add  x9, x8, x0      # copy A[0] to x9
  add  x10, x0, x0     # sum (x10) = 0
  add  x11, x0, x0     # int i = 0
  addi x13, x0, 20     # x13 = 20 (loop bound)
Loop:
  bge  x11, x13, Done  # end if i > 20
  lw   x12, 0(x9)      # load A[i] into x12
  add  x10, x10, x12   # sum += A[i]
```

```
    addi x9, x9, 4      # A[i+1] in x9
    addi x11, x11, 1    # i++
    j Loop              # loop again
Done:                   # end!
```

Logical Operators:

- **Bit-by-bit AND**
  - ↳ C: &
  - RISC-V: and
- **Bit-by-bit OR**
  - ↳ C: |
  - RISC-V: or
- **Bit-by-bit XOR**
  - ↳ C: ^
  - RISC-V: xor
- **Bit-by-bit shift left**
  - ↳ C: <<
  - RISC-V: sll
- **Bit-by-bit shift right**
  - ↳ C: >>
  - RISC-V: srl

↳ useful for **moving, extracting, and inserting groups of bits!**

↳ **Bitwise and** used for **'masks'**

- ↳ andi with 0x000000FF isolates the **least significant byte**
- ↳ and: with 0xFF000000 isolates the **most significant byte**

↳ there is no **logical NOT** in RISC-V, because it can be implemented using XOR.

↗ XOR gate with x and 1 gives $\bar{x}$!

Shift Left Logical (sll) and immediate (slli):

```
slli rd, rs1, number
```

slli: stands for shift left logical immediate
rd: destination register
rs1: source register
number: amount to shift

Example: `slli x11, x12, 2`

↳ Store in x11 the value from x12 shifted by 2 bits to the left, inserting 0s on the right. the bits on the left dissapear (not wrap).

↳ before : 0000 ... 0010
after : 0000 ... 1000

srl and srli do the exact opposite shift.

Arithmetic Shifting:

Shift right arithmetic: sra, srai
↳ moves n bits to the right
- inserts high-order sign bit into empty bits

```
srai rd, rs1, number
```

srai: stands for shift right arithmetic immediate

rd: destination register

rs1: source register

number: amount to shift

Example: srai x10, x10, 4

↳ replace the contents of register x10 by bit shifting 4 bits to the right with the same sign.

before: 1111 1111 1111 1111 1111 1111 1110 0111 = -25

after: 1111 1111 1111 1111 1111 1111 1111 1110 = -2

↳ srai is NOT the same as dividing by 2^n.