SoC 101:

*a.k.a., "**Everything you wanted to know about a computer but were afraid to ask**"*

Lecture 2: The Microprocessor

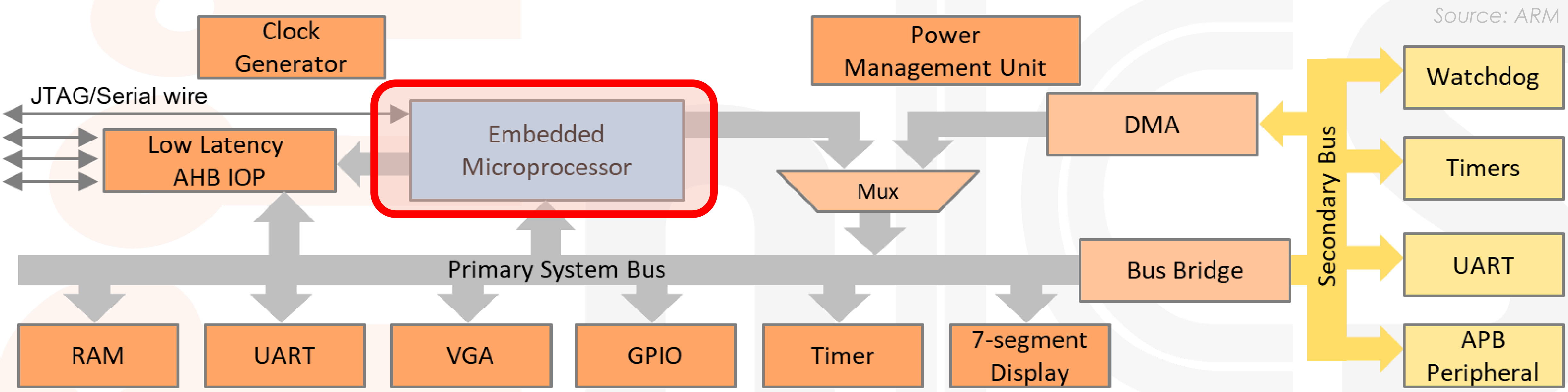
Prof. Adam Teman
EnICS Labs, Bar-Ilan University

29 March 2023

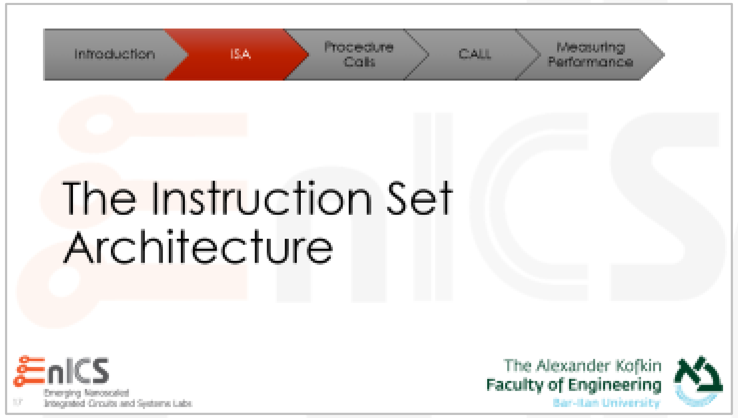
EnICS
Emerging Nanoscaled Integrated Circuits and Systems Labs

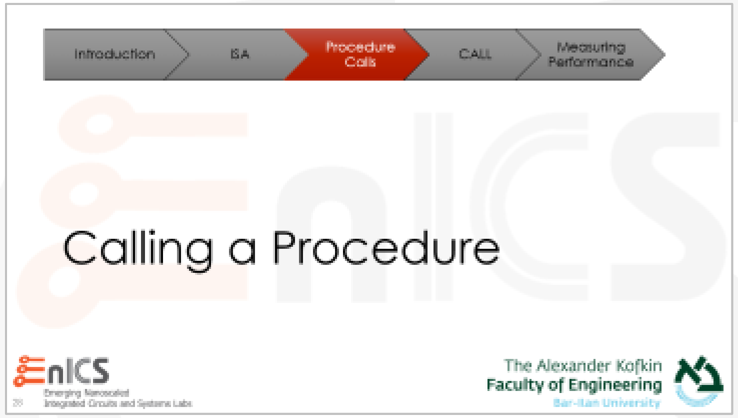
The Alexander Kofkin Faculty of Engineering
Bar-Ilan University

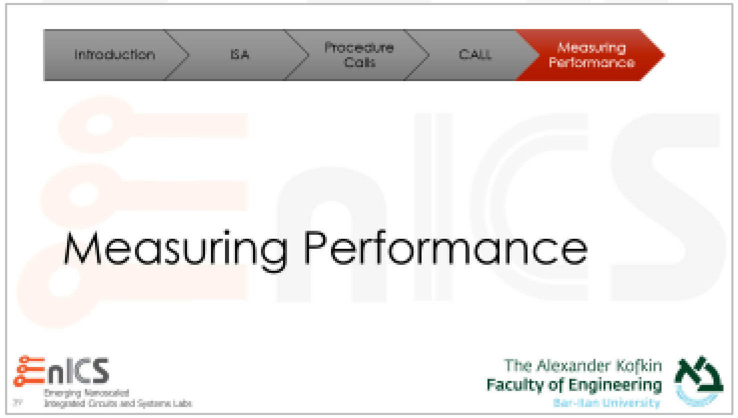
This Lecture



Lecture Overview



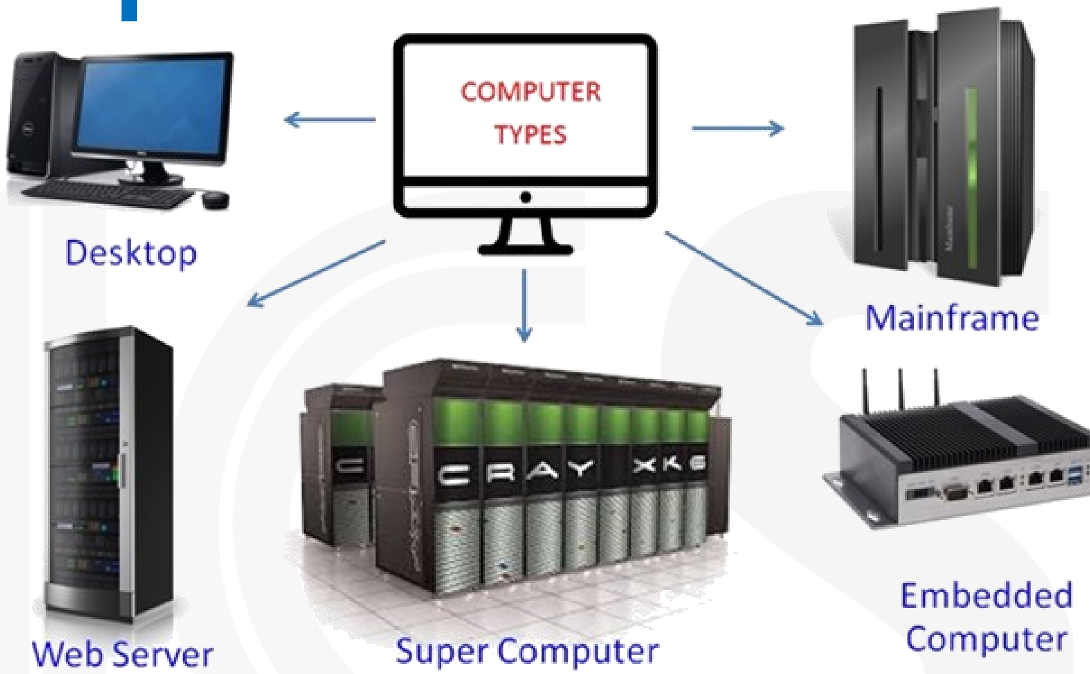




Introduction and History

Traditional Classes of Computers

- **Personal computers**
  - General purpose, variety of software
  - Subject to cost/performance tradeoff
- **Server computers**
  - Network based
  - High capacity, performance, reliability
  - Range from small servers to building sized
- **Supercomputers**
  - High-end scientific and engineering calculations
  - Highest capability but represent a small fraction of the overall computer market
- **Embedded computers**
  - Hidden as components of systems
  - Stringent power/performance/cost constraints



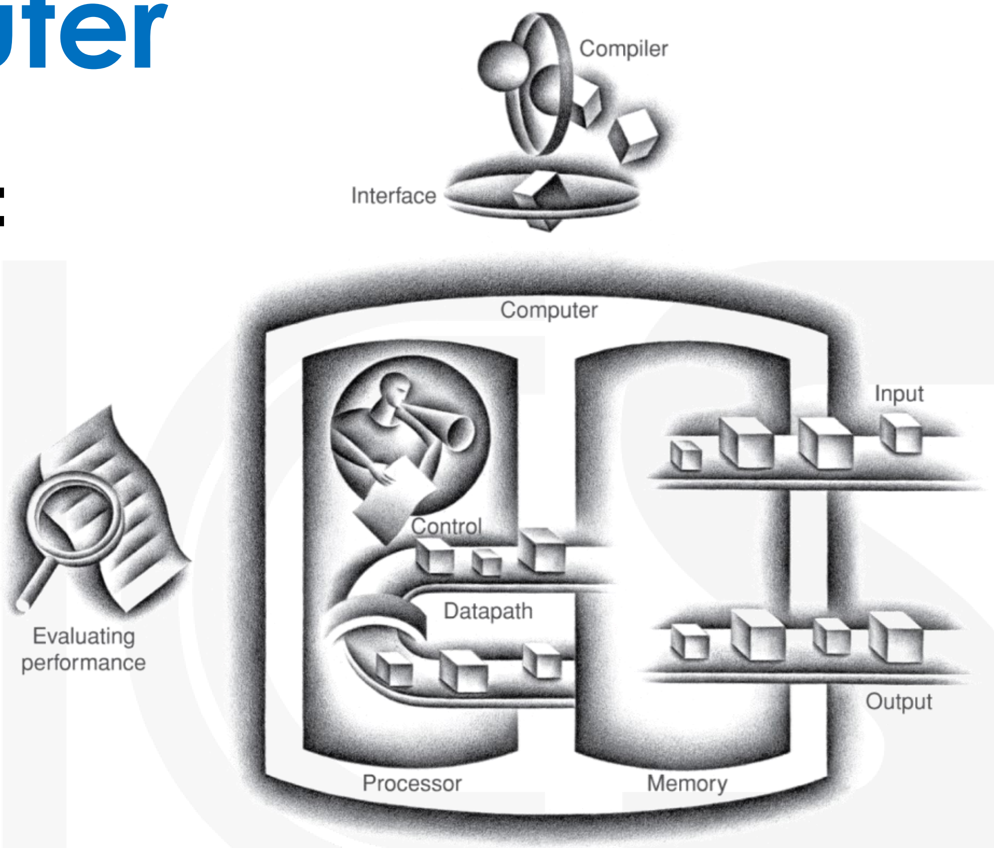
Source: www.learncomputerscienceonline.com

Embedded Computer:
A computer inside another device, used for running one predetermined application or collection of software.

Components of a Computer

- **The five classic components of a computer are:**
  - Input
  - Output
  - Memory
  - Datapath
  - Control

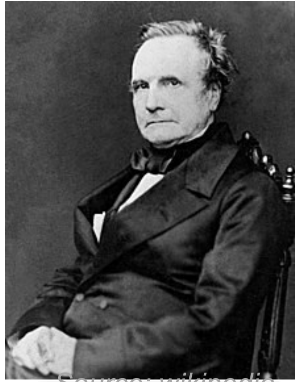
  The Processor
- **Same components for all kinds of computer**
  - Desktop, server, embedded
- **Input/output (I/O) includes**
  - **User-interface devices**: Display, keyboard, mouse
  - **Storage devices**: Hard disk, CD/DVD, flash
  - **Network adapters**: For communicating with other computers



Source: P&H, Chapter 1

The first computer

- **The "Analytical Engine" (1837-1871)** ***never completed***
  - Conceived by Charles Babbage, the "Father of Computing".
  - The first concept of a *programmable general-purpose* computer.
  - It would be able to perform any calculation set before it.
- **Included the main components of a modern computer:**
  - The Mill ↔ The CPU and ALU (conditional branching)
  - The Store ↔ Memory (1K 50-digit numbers)
  - The Reader ↔ Input (Jacquard Punch Cards)
  - The Printer ↔ Output (print out mathematical tables)
- **The first computer program**
  - Algorithm to calculate Bernoulli numbers.
  - Written by Ada Lovelace for the Analytical Engine.


Source: wikipedia


Source: Pointdexter's

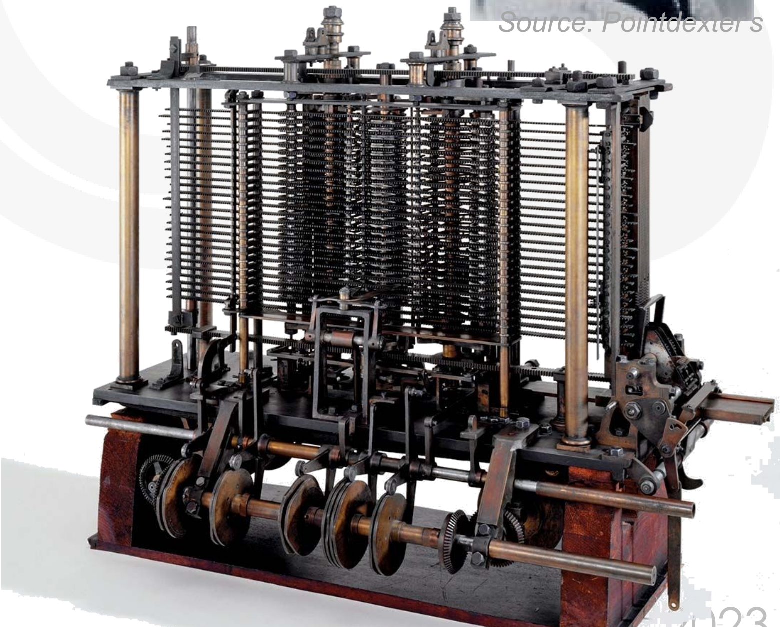


PHOTO: Britannica, Science Museum London

The Turing Machine

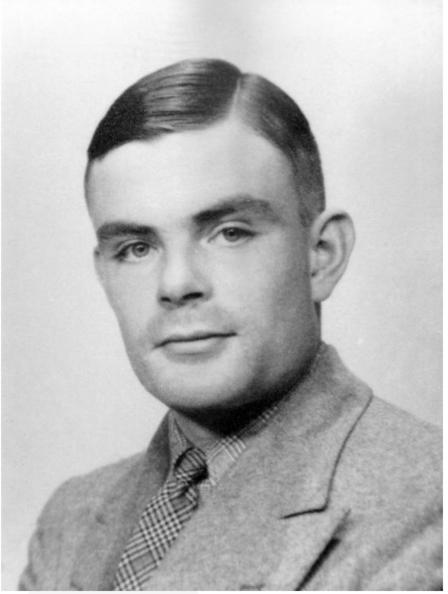
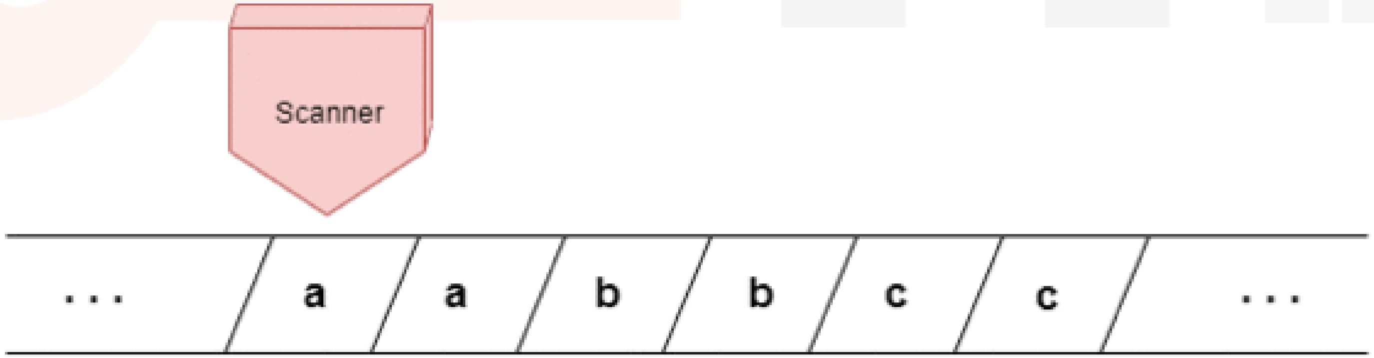


PHOTO: Britannica

- **Described by Alan Turing (1936)**
  - A machine with finite states, an infinite tape (memory) of symbols and a scanner that can read and write to the current position.
  - At any moment, the current symbol is scanned, and depending on the state, it may be replaced, and the scanner is moved to a new position.
- **Such a machine can compute all computable problems.**
- ***Turing completeness* is the ability for a system of instructions to simulate a Turing machine.**



Source: DEV

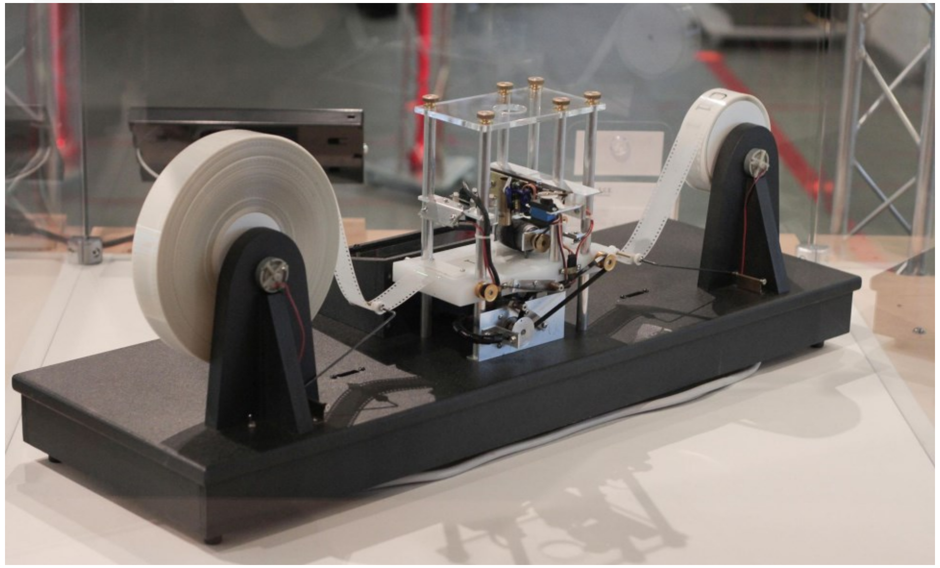


PHOTO: Rocky Acosta

Early Computers

- **The ENIAC is often considered "the first computer"**
  - Actually, it was the first *"Electronically Programmable General Purpose Computer"*.
  - In other words, it was the first electronic Turing Machine*.

    ** The Analytical Engine was the first Turing Machine.*
  - But programming it required rewiring hundreds of cables
  - This process took anywhere from days to weeks.
- **A new concept was necessary**
  - Mauchly and Eckert were working on EDVAC and came up with the idea of treating the *instructions* as just another piece of *data*.
  - Von Neumann heard about it and wrote the nominal paper about the "stored-program"

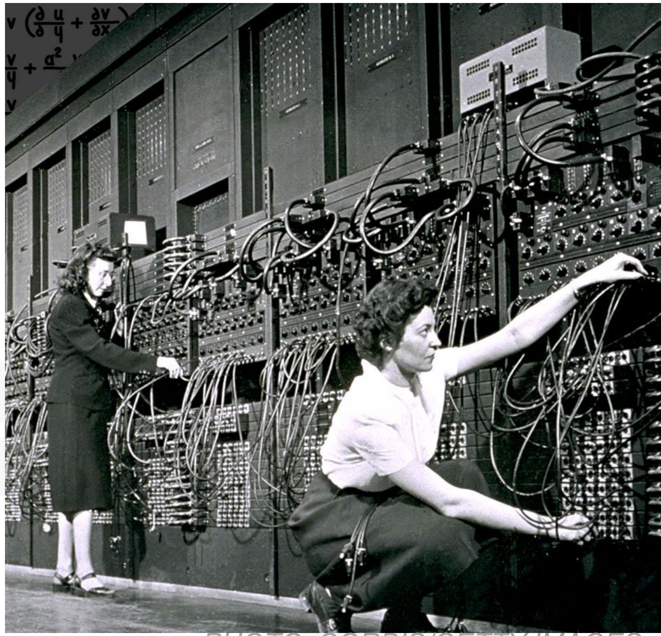
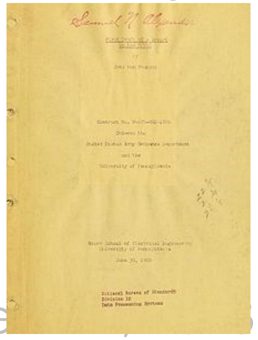


PHOTO: CORBIS/GETTY IMAGES

PHOTO: Life Magazine



First draft of a report on the EDVAC

Stored-Program Computers

- **In a *stored-program* *computer*, a.k.a. a *Von Neumann Machine*:**
  - Instructions represented in binary, just like data
  - Programs are stored in memory, just like data
  - The memory can be read and written when given an address
  - The *program counter* (**PC**) holds the address of the current instruction
  - The program flow is achieved by incrementing the **PC** or branching.

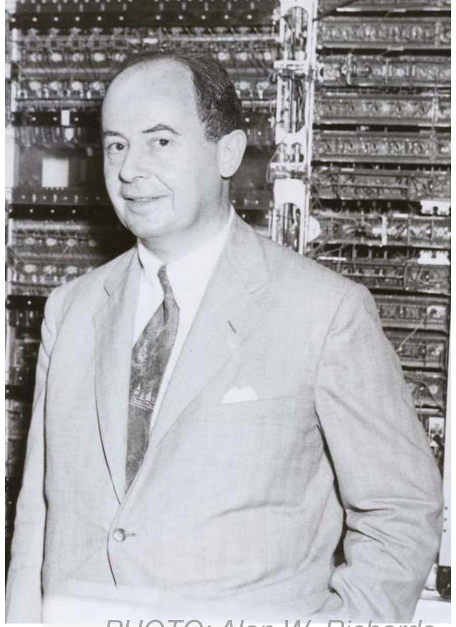
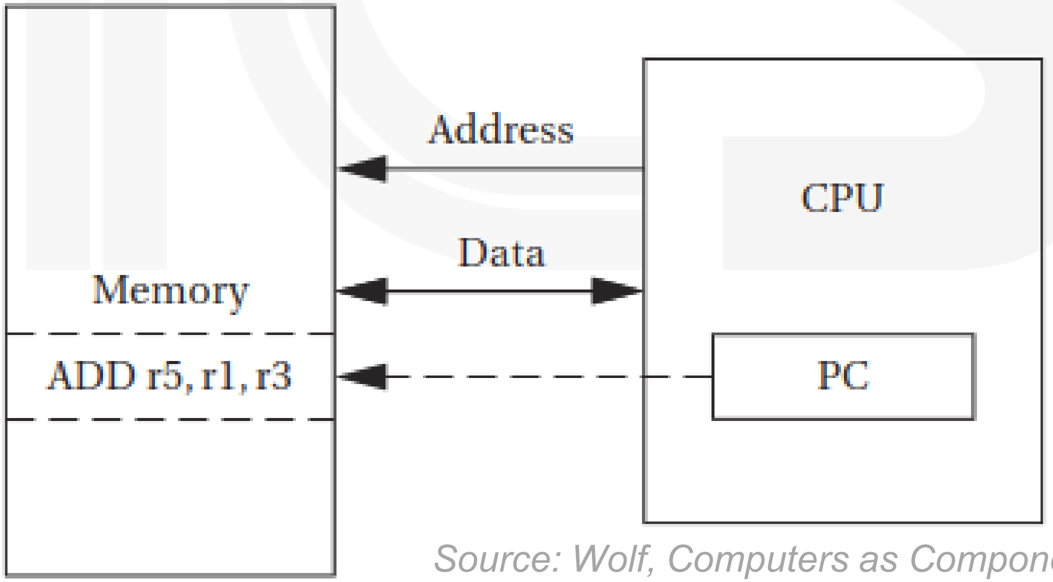


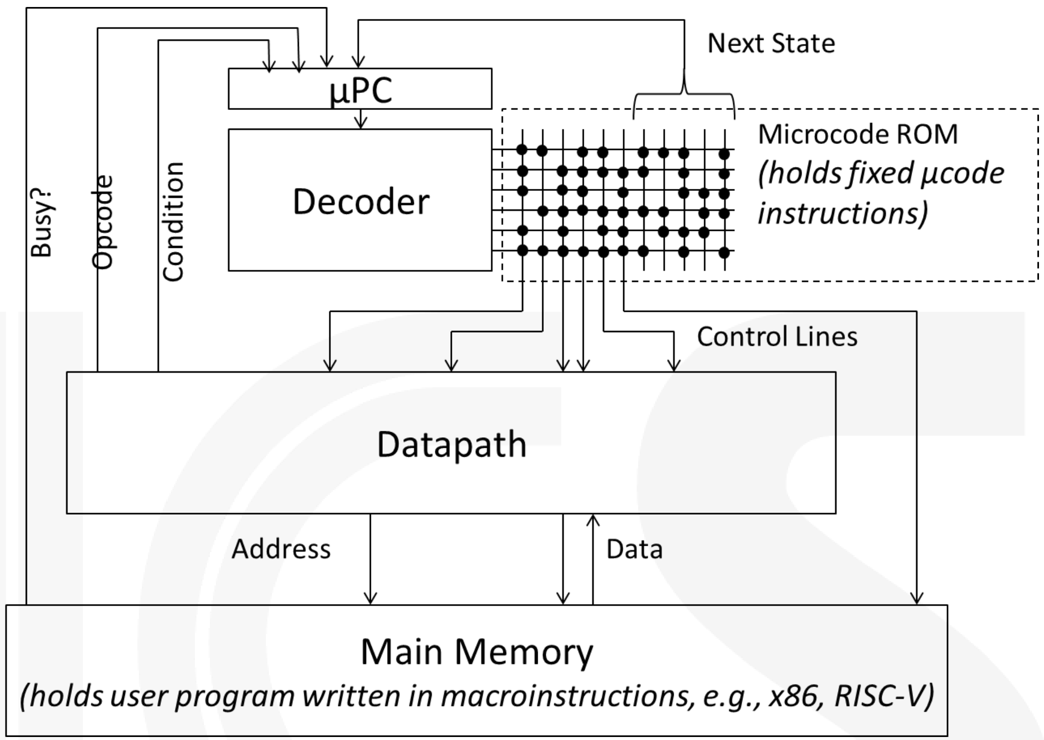
PHOTO: Alan W. Richards and Brittanica

- **Programs can operate on programs**
  - e.g., compilers, linkers, …
- **Programs are shipped as files of binary numbers ("binaries")**
  - "Binary compatibility" allows compiled programs to work on different computers



Source: Wolf, Computers as Components

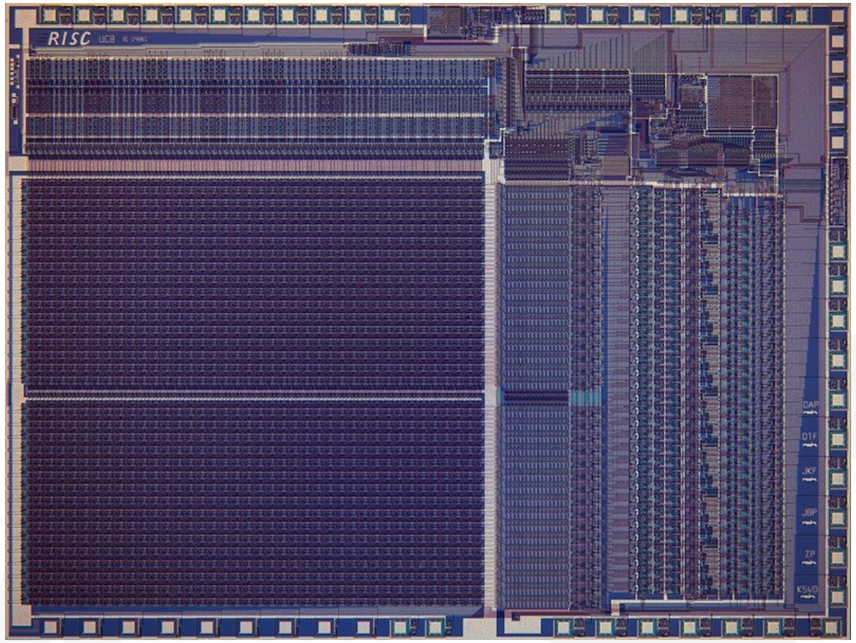
Microcoded Computers



- **In early processors:**
  - ROM was cheaper and faster than RAM.
  - Logic was expensive compared to ROM.
  - Getting the *control unit* right was very hard.
- **In 1958, Maurice Wilkes came up with the idea of *microcoding*.**
  - Implement the *"microcoded" control unit* with ROM to make it programmable.
  - Microcode turns complex instructions into a set of datapath control signals.
  - Microcode is part of micro-architecture and not visible to the programmer.
- **First used to design the control unit of EDSAC-II.**
  - Easier to design, fix bugs, support new instructions without changing datapath.
  - But does not benefit from µarch innovations, better compilers, reprogramming.

From CISC to RISC

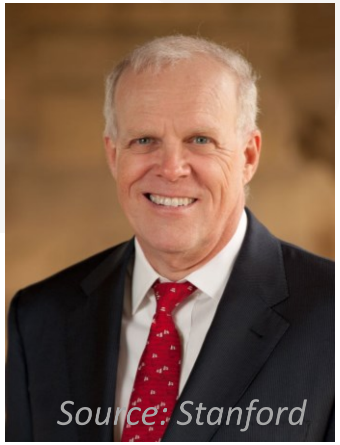
- **Complex instruction set computers (CISC)**
  - Large variety of instructions
  - Instructions may perform very complex tasks
    - e.g., string searching
  - Very common for early computer architectures
- **Reduced instruction set computers (RISC)**
  - Fewer and simpler instructions
    - Most compiled code only used a few of the available CISC instructions
  - Load/Store instruction sets
    - Operations cannot be performed directly on memory locations, only on registers
  - Relatively straightforward to pipeline
  - Virtually all ISAs invented since the eighties are RISC.



RISC-I (1982)
5 µm NMOS, 1 MHz, 44K transistors

John Cocke
IBM-801 (1980)

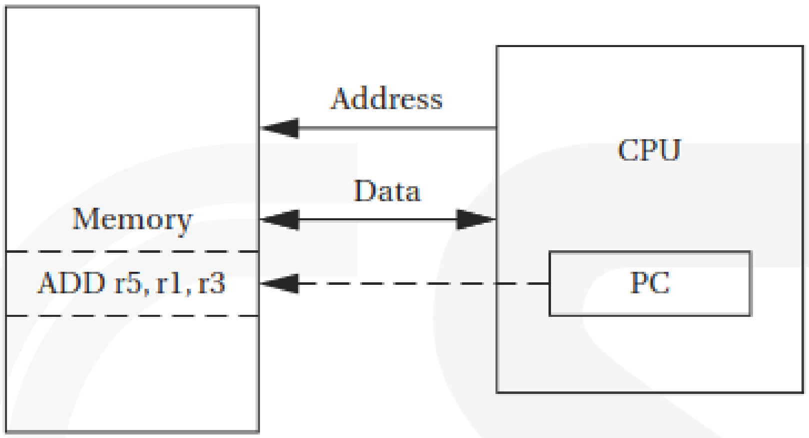
Dave Patterson
RISC (1982)



John Hennessy
MIPS (1983)

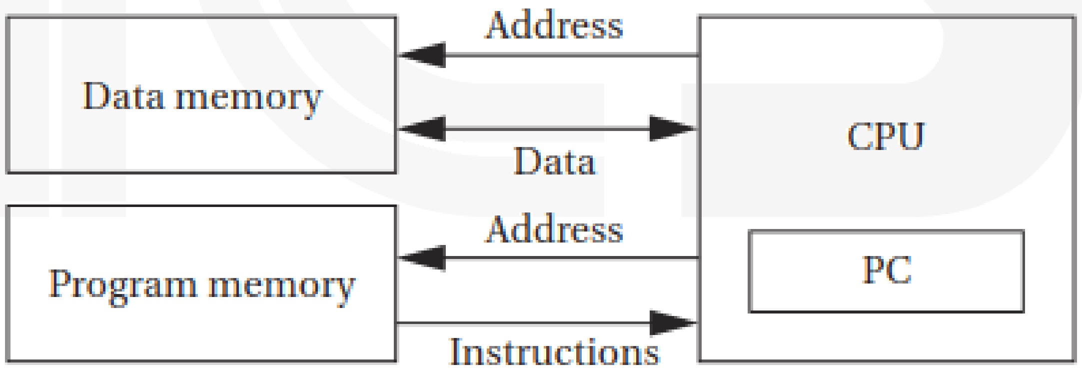
Princeton/Harvard Architecture

- ***Von Neumann*** **or** ***Princeton Architecture***
  - Instructions and data share a single memory space
  - Can be **either** reading an instruction or reading/writing data from/to the memory
  - Limits operating bandwidth



Source: Wolf, Computers as Components

- ***Harvard Architecture***
  - Uses **two separate** memory spaces for instructions and data
  - The CPU can **both** read an instruction and access data memory at the same time
    - Improved operating throughput
  - RISC designs are also more likely to feature this model



The Computer Architecture Monopoly

- **While thousands of instruction set architectures (ISAs) have been invented and used over the years, the vast majority have died off and disappeared.**
- **Only ~~two~~ three general purpose ISAs are commonly found today:**
  - Intel x86 (a.k.a., AMD64): A CISC architecture, which (still) dominates the laptop, desktop and server domains.
  - ARM: A (formerly) RISC architecture, which dominates the embedded computing domain.
  - RISC-V: An open-source RISC architecture that is gaining popularity.
- **Other ISAs survive mainly as legacy or for application specific purposes.**
- **We will discuss this in detail later in the course, but for now, we introduce the ISA as a concept, and use RISC-V, where examples are required.**

Can't we be ISA agnostic?

- **Compilation is the translation of high-level code to machine code (ISA).**
  - Converts a program from the source language (e.g., C) to an equivalent program in another language (e.g., RISC-V assembly).
- **But, we don't compile a Python script…**
  - An Interpreter directly executes a program in the source language.
  - An Interpreter is a program that executes other programs
- **In general, we interpret a high-level language when <u>efficiency is not critical</u> and translate to a lower-level language to <u>increase performance</u>**

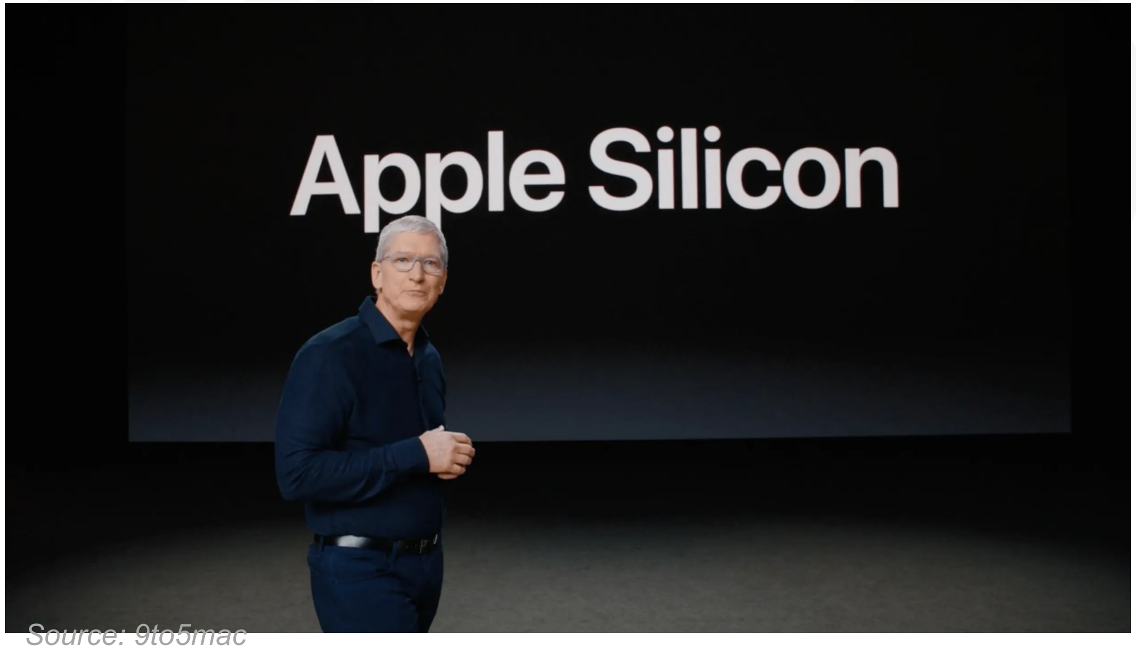
The language execution continuum:

Python Java C++ — C — Assembly — Java bytecode — Machine code

← Easy to program / Inefficient to interpret ——— Difficult to program / Efficient to interpret →

Machine language interpretation?

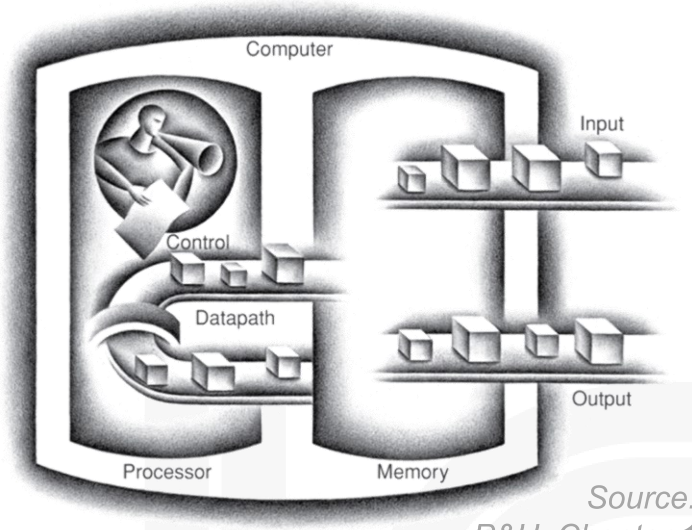
- **Interpreting high-level code is understandable, but *would we ever interpret machine language*?**
- **Emulation/Simulation:**
  - e.g., Whisper – a RISC-V simulator for learning/debugging
- **Backwards Compatibility**
  - e.g., Apple Macintosh conversion from Motorola 680x0 to PowerPC to x86
  - e.g., Apple Mac conversion (revisited…) from x86 to ARM (Apple Silicon M1)



Source: 9to5mac

The Instruction Set Architecture

Instructions



Source: P&H, Chapter 1

- Problem
- Algorithm
- Program Language
- Runtime System (OS/VM)
- Architecture (ISA)
- Microarchitecture
- Logic
- Transistors
- Electrons

PHOTO: Shuttershock

- **The objectives of a microprocessor:**
  - Transfer or store data
  - Operate on data
  - Make decisions based on the values or outcomes of operation
- **Correspondingly, there are three categories of instructions:**
  - **Data Transfer**: Move data within the system and exchange data with external devices.
  - **Flow of Control**: Determine the execution order of instructions.
  - **Arithmetic and Logic**: computational capabilities and functionality of the microprocessor.
- **The *instruction set architecture* (ISA) is the set of instructions and concepts that provide an interface ("a contract") between the software and hardware.**

Instructions (ctnd.)

- **In a Von Neumann architecture, instructions are "*just another piece of data*".**
  - Therefore, an instruction is just a data word (a vector of bits).
  - For example, in RISC-V, instructions are 32-bits wide.
- **The bits of data encode all the data necessary to carry out the instruction, e.g.:**
  - **Opcode**: An encoding of the name/type of the instruction
  - **Operands**: *Sources* (inputs) and *Destinations* (outputs) of the operation
  - **Immediates**: Hard-coded constants to be used as inputs to the operation
  - Other information (**metadata**): Additional encodings that can affect the control flow, operation, and additional features.
- **For example: `ADD RD,RS1,RS2` (RD←RS1+RS2) could be implemented**

OPCODE	RD	RS1	RS2

← *N*-bits →

Side note: Simplified view of Memory

- **Before starting, let's briefly introduce registers vs. memory**
  - Registers are flip-flops, while memory (for now) is SRAM.
  - Both are typically synchronous with "one-cycle latency", so we need a single clock edge to read or write from/to both registers and memory.
- **However, registers are advantageous over memory, since:**
  - Access to an SRAM takes (close to) a whole clock cycle, while other operations can be applied before or after register access.
  - Several registers can be accessed simultaneously, while only a single SRAM address can be accessed for read/write during a given cycle.
  - Register access requires lower power than memory access.
- **Therefore, operation on registers is typically preferred to operations on memory.**
  - However, due to size, only a limited number of registers are available.

General Purpose Register ISA

- **Most modern architectures have general purpose register (GPR) ISAs**
  - The architecture uses datapath registers <u>as operands</u>.
  - As opposed to *stack* architectures or *accumulator* architectures.
  - Generally faster and easier for compilers.
- **The ISA (and ABI*) provides a defined set of processor registers, including:**
  - General Registers (i.e., `R1`, `R2`, `R3`)
  - Program Counter (`PC`)
  - Stack Pointer (`SP`) and Frame Pointer (`FP`)
  - Others
- **Different ISAs have a different number of GPRs**
  - x86: 8 regs, ARMv7: 16 regs, RISC-V: 32 regs

Register	ABI Name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary/alternate link register
x6–7	t1–2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10–11	a0–1	Function arguments/return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries
f0–7	ft0–7	FP temporaries
f8–9	fs0–1	FP saved registers
f10–11	fa0–1	FP arguments/return values
f12–17	fa2–7	FP arguments
f18–27	fs2–11	FP saved registers
f28–31	ft8–11	FP temporaries

RISC-V Registers

**ABI* = Application Binary Interface*

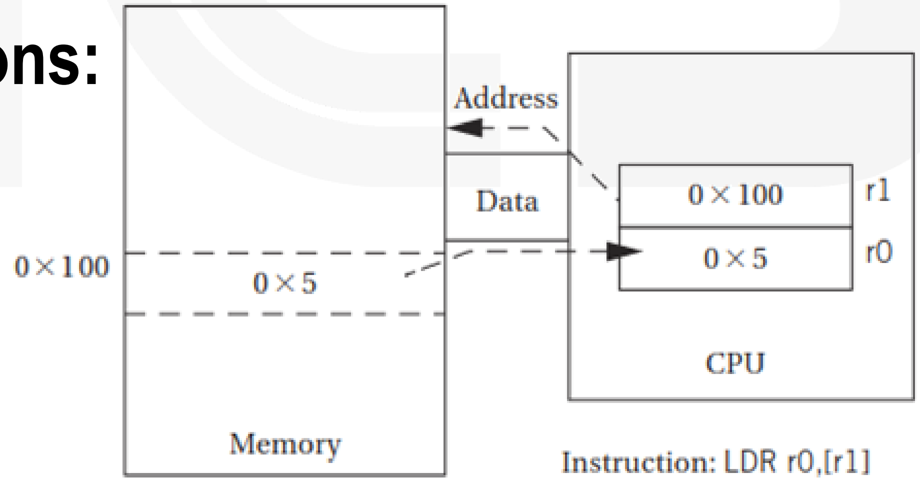
Data Transfer Instructions

- **Data transfer instructions are responsible for *moving data around* inside the processor and for *bringing data in* from the outside world or *sending data out*.**
- **Many modern RISC machines are "*load-store*" architectures:**
  - Data is loaded from memory to registers for computation
  - Results are written to registers, which can later be stored back in memory.
  - Only accesses the memory through explicit LOAD/STORE instructions
- **These are carried out with two types of instructions:**
  - Load data from memory to a register, e.g.:

```
LOAD R1, ADDRESS
```

  - Store data from a register to memory, e.g.:

```
STORE R1, ADDRESS
```



Source: Computers as Components

Memory Access Addressing Modes

- **Some of the most common addressing modes are:**
  - *Direct* Addressing

    `LOAD R1, const`   `R1←MEM[const]`

    Memory address immediately available, but size limited.
  - *Register Indirect* Addressing

    `LOAD R1, R2`   `R1←MEM[R2]`

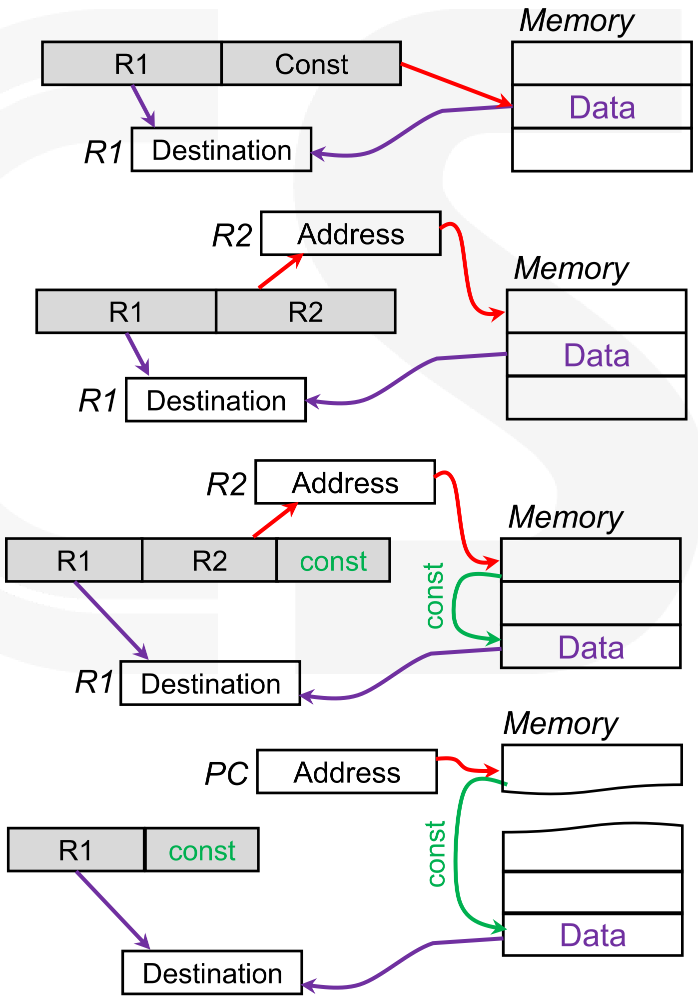
    Store full bit-width (e.g., 32-bits) inside register.
  - *Displacement* or *Indexed* Addressing

    `LOAD R1, R2, const`   `R1←MEM[R2+const]`

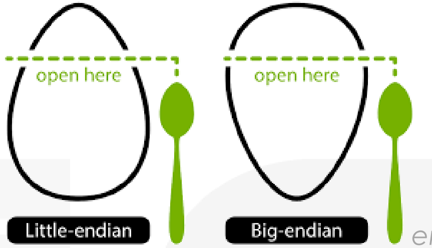
    Enables access relative to base pointer (e.g., array, stack).
  - *Program Counter Relative* Addressing

    `LOAD R1, (PC), const`   `R1←MEM[PC+const]`

    Enables relative addressing based on code placement.
- **RISC-V only supports Displacement Addressing!**



Elaboration: Big Endian vs. Little Endian



Source: endian.com

- **From Gulliver's Travels**
  - *Big Endians* broke their eggs at the large end ("the primitive way")
  - The Lilliputian King made his subjects (*Little Endians*) break their eggs at the small end.

Consider the number 1025 as we normally write it:

BYTE3 BYTE2 BYTE1 BYTE0

00000000 00000000 00000100 00000001

Address	[31:24]	[23:16]	[15:8]	[7:0]	
0x00000000	Byte0	Byte1	Byte2	Byte3	Word 1
0x00000004	Byte0	Byte1	Byte2	Byte3	Word 2
0x00000008	Byte0	Byte1	Byte2	Byte3	

Address	[31:24]	[23:16]	[15:8]	[7:0]	
0x00000000	Byte3	Byte2	Byte1	Byte0	Word 1
0x00000004	Byte3	Byte2	Byte1	Byte0	Word 2
0x00000008	Byte3	Byte2	Byte1	Byte0	Word 3

Big Endian

ADDR3 ADDR2 ADDR1 ADDR0

BYTE0 BYTE1 BYTE2 BYTE3

00000001 00000100 00000000 00000000

Examples

Names in China (e.g., Teman Adam)

ISO 8601 Dates YYYY-MM-DD (e.g., 2019-03-20)

Eating Pizza crust first

Little Endian

ADDR3 ADDR2 ADDR1 ADDR0

BYTE3 BYTE2 BYTE1 BYTE0

00000000 00000000 00000100 00000001

Examples

Names in the West (e.g., Adam Teman)

Dates in England DD/MM/YYYY (e.g., 20/03/2019)

Eating Pizza skinny part first (the normal way)

Source: www.gutenberg.org

Adapted from: Berkeley CS61C

Elaboration: Big Endian vs. Little Endian

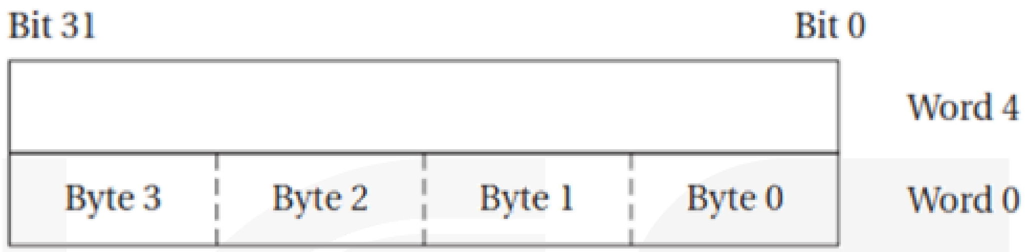
- **Assume the memory holds the following bytes in Little Endian:**
  - Loading a byte from `0xE5000004` will give:

    `0x0000001A`
  - Loading a halfword from `0xE5000004` will give:

    `0x00002B1A`
  - Loading a word from `0xE5000004` will give:

    `0x4D3C2B1A`
- **We can view this as holding a sequence of aligned words:**

address (in hex)	data (in hex)
...	...
E5000004	4D3C2B1A
E5000008	81706F5E
E500000C	C5B4A392
...	...

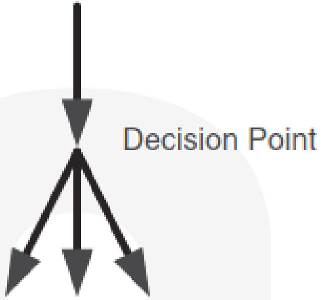


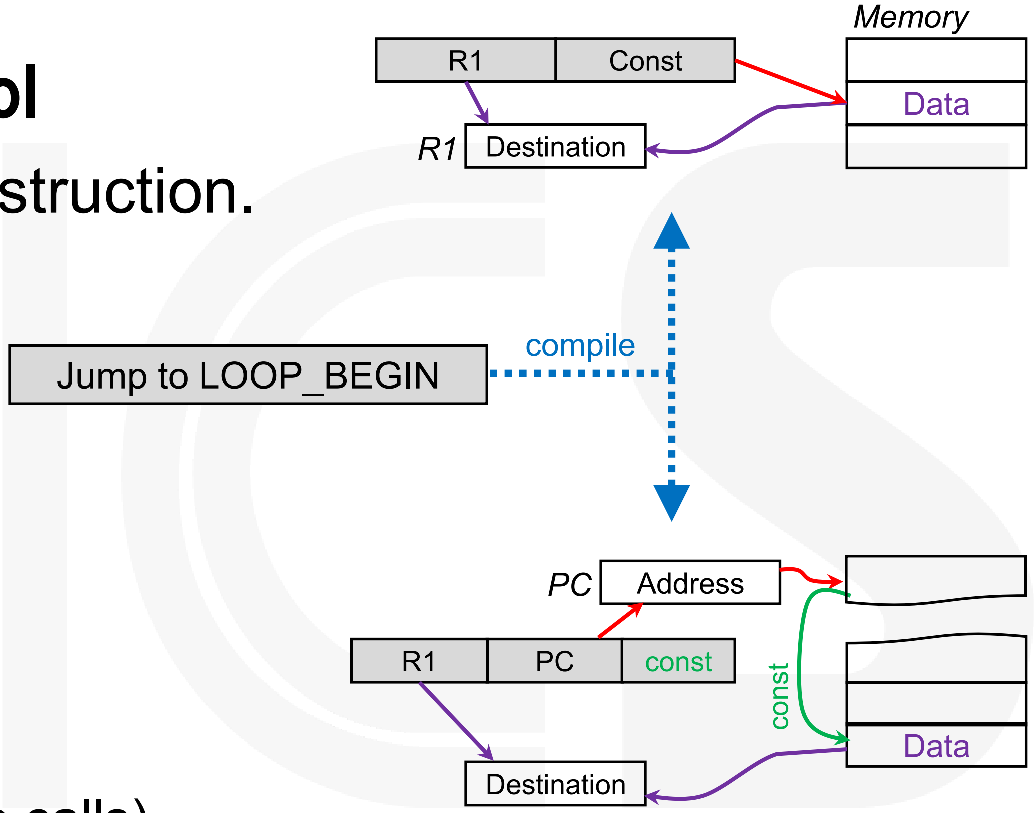
Little-endian

address (in hex)	data (in hex)
...	...
E5000004	1A
E5000005	2B
E5000006	3C
E5000007	4D
E5000008	5E
E5000009	6F
E500000A	70
E500000B	81
E500000C	92
E500000D	A3
E500000E	B4
E500000F	C5
...	...

Flow-Control Instructions

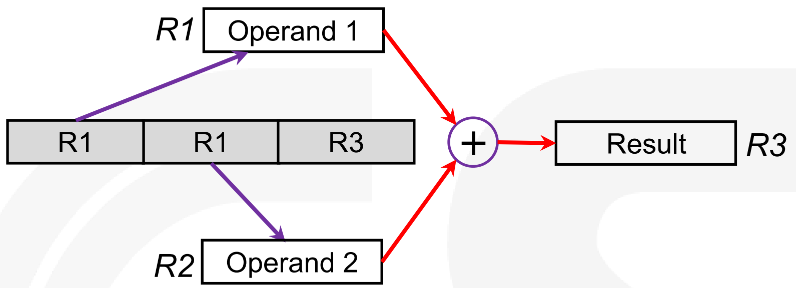
- **Utilize addressing modes to change flow-control**
  - As opposed to incrementing the PC by one instruction.
- **Conditional Branches**
  - Branch if equal/not equal
  - Branch if greater/less than
- **Nonconditional**
  - Jump
  - Jump and Link
    - Stores the return address in a register (for procedure calls)
- **Labels are commonly used to make the assembly code more readable**
  - Will be replaced by <u>absolute</u> (immediate addressing) or <u>relative</u> (often `PC+offset`) addresses during compilation

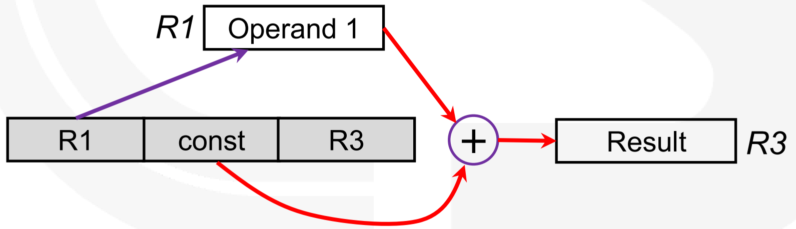




Arithmetic-Logic Instructions

- **Register-Register Arithmetic**
  - `ADD R3, R1, R2` → `R3 = R1+R2`
- **Register-Immediate Arithmetic**
  - `ADD R3, R1, const` → `R3 = R1+const`
- **Common Arithmetic-Logic Commands:**
  - Basic: Add, Subtract
  - Extended: Multiply, Divide
  - Floating Point Arithmetic
  - Logical: AND, OR, XOR
  - Shift: Shift left/right, logical/arithmetic
  - Compare: “Set if” less/greater than

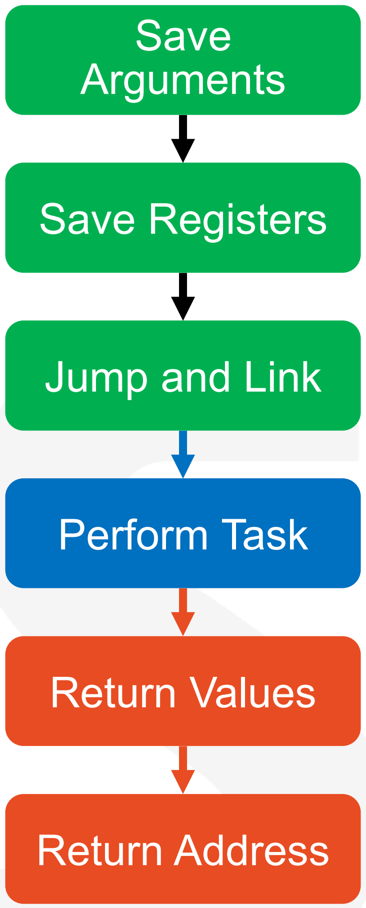




Calling a Procedure

Procedure Calls

- **There are six stages in calling a function (a.k.a. procedure):**
  1. Place the arguments where the function can access them
  2. Acquire storage and save the registers that are needed
  3. Save return address and Jump to the function
  4. Perform the desired task
  5. Return from the function:
     - Place the result values where the calling function can access them
     - Restore any saved registers
     - Release any local storage resources
  6. Return control to the point of origin (return address)
- **Procedures use the stack to store registers, variables, etc.**



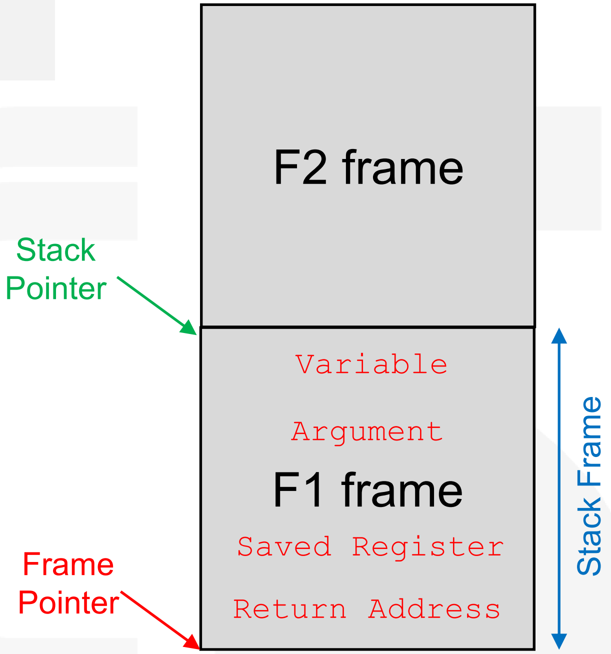
The Stack

```
main() {
   f1(xyz);
}

void f1(int a) {
   f2(a,25);
}
```

Variable
Saved Register
F1 SP and FP
Return Address

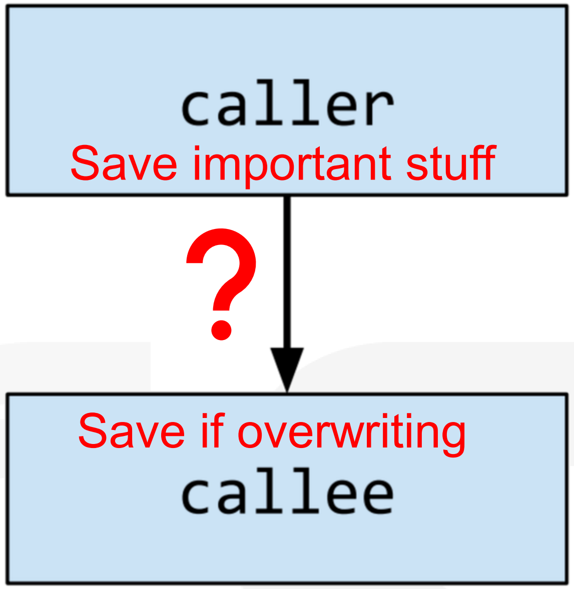
- **Building a Stack allows nested procedure calls**
  - The stack contains one **stack frame** (or **activation record**) for each active procedure
  - The stack frame contains the *return address*, *saved register* values and *parameters* (arguments)
  - The **stack pointer** points to the top of the stack and grows when additional data is pushed.
  - The **frame pointer** points to the beginning of the frame, which is the stack pointer value when the procedure is called.



The "***prologue***" of a function call:
- Moves the stack pointer up by framesize
- Stores the current return address on the stack
- Stores other saved registers on the stack

The "***epilogue***" of a function call:
- Restores save regs and return address
- Move the stack pointer back down by framesize
- Change the PC to the return address

Calling Conventions



- **A procedure is initiated from within another piece of code.**
  - Initiating function = "caller", Subroutine = "callee"
- **In order to ensure that the caller's state is not changed during the subroutine, important data must be saved.**
  - The caller can save important registers before calling the subroutine.
  - The callee can save registers that are going to be overwritten during execution.
  - Since the two functions are written independently, redundant saves may be applied.
- **The calling convention will define which registers should be saved by the callee and which by the caller and this can lead to improved efficiency.**
  - The compiler/programmer should (almost) always adhere to the calling convention.
  - The calling convention is part of the Application Binary Interface (ABI)

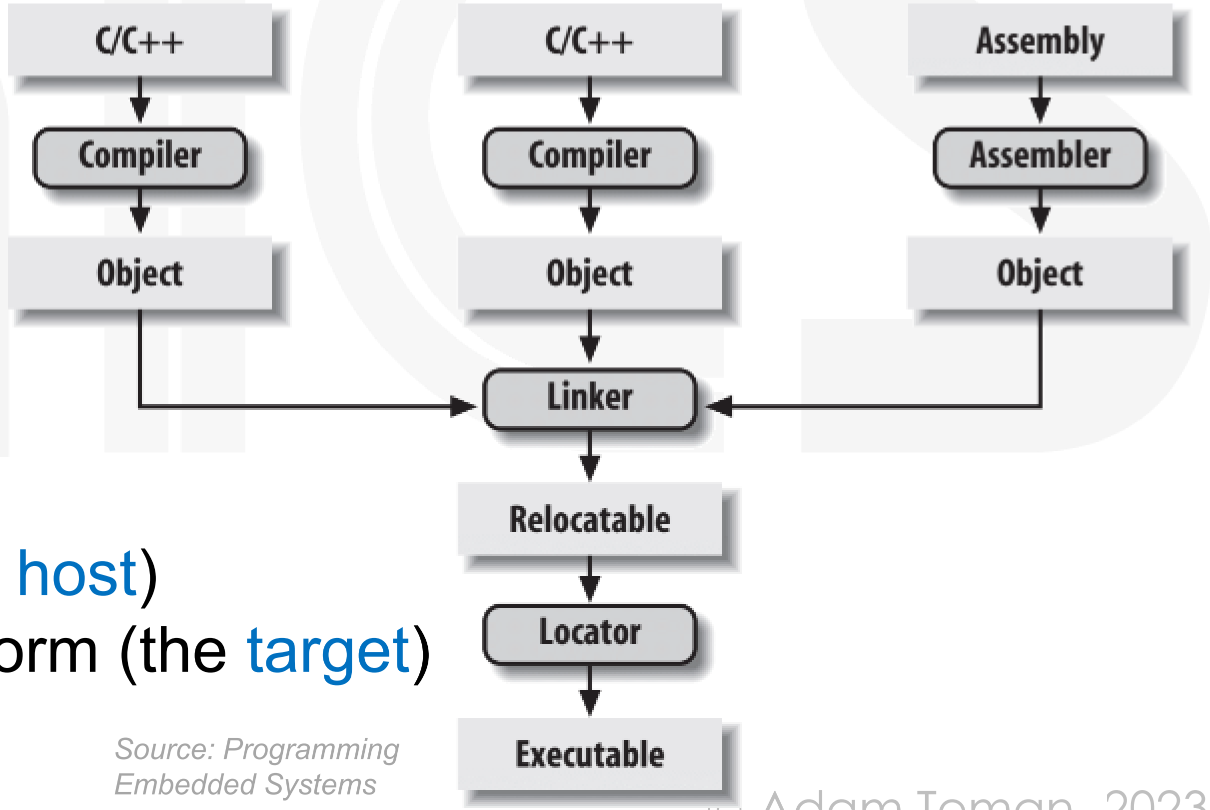
Running a Program - CALL

(Compiling, Assembling, Linking, and Loading)

EnICS Emerging Nanoscaled Integrated Circuits and Systems Labs
The Alexander Kofkin Faculty of Engineering
Bar-Ilan University

General Compilation Process

- **Converting source code into an executable binary image involves three steps:**
  1. **Compiling** (and **assembling**) each source file into an object file.
  2. **Linking** together all object files into a single object file
  3. **Relocating** relative addresses in the object file into absolute addresses
- **The result is a binary image, ready to run on the hardware**



Source: Programming Embedded Systems

- ***Cross Compilation*:**
  - Compiling software on one system (the host) intended for running on a different platform (the target)

Steps in Compiling and Running a C Program

```
gcc -O2 -S -c foo.c
```

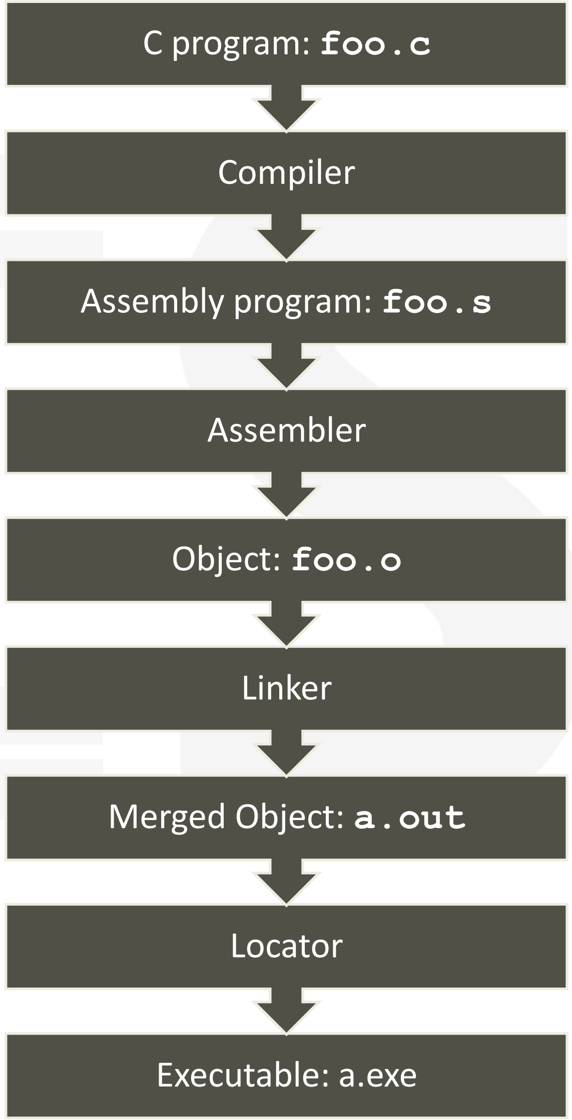
`foo.c`

- **Compiler**
  - Input: High-Level Language Code (`foo.c`)
  - Output: Assembly Language Code (`foo.s`)
  - (Note: Output *may* contain pseudo-instructions)

`foo.s`

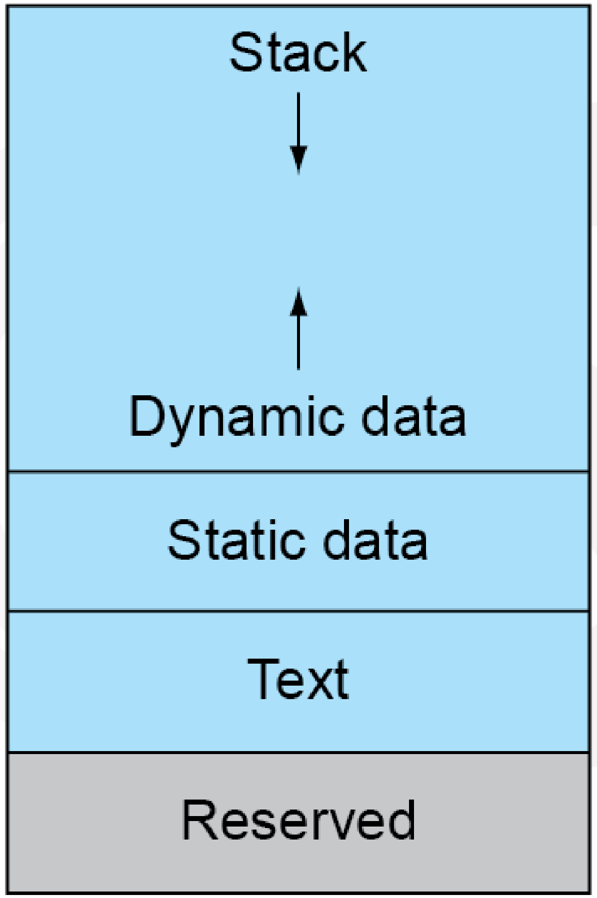
`foo.o`

- **Assembler**
  - Input: Assembly Language Code (`foo.s`)
  - Output: Object Code, information tables (`foo.o`)
  - Reads and Uses Directives
  - Replace Pseudo-instructions
  - Produce Machine Language
  - Creates Object File (ELF or COFF format)



Processor Memory Map

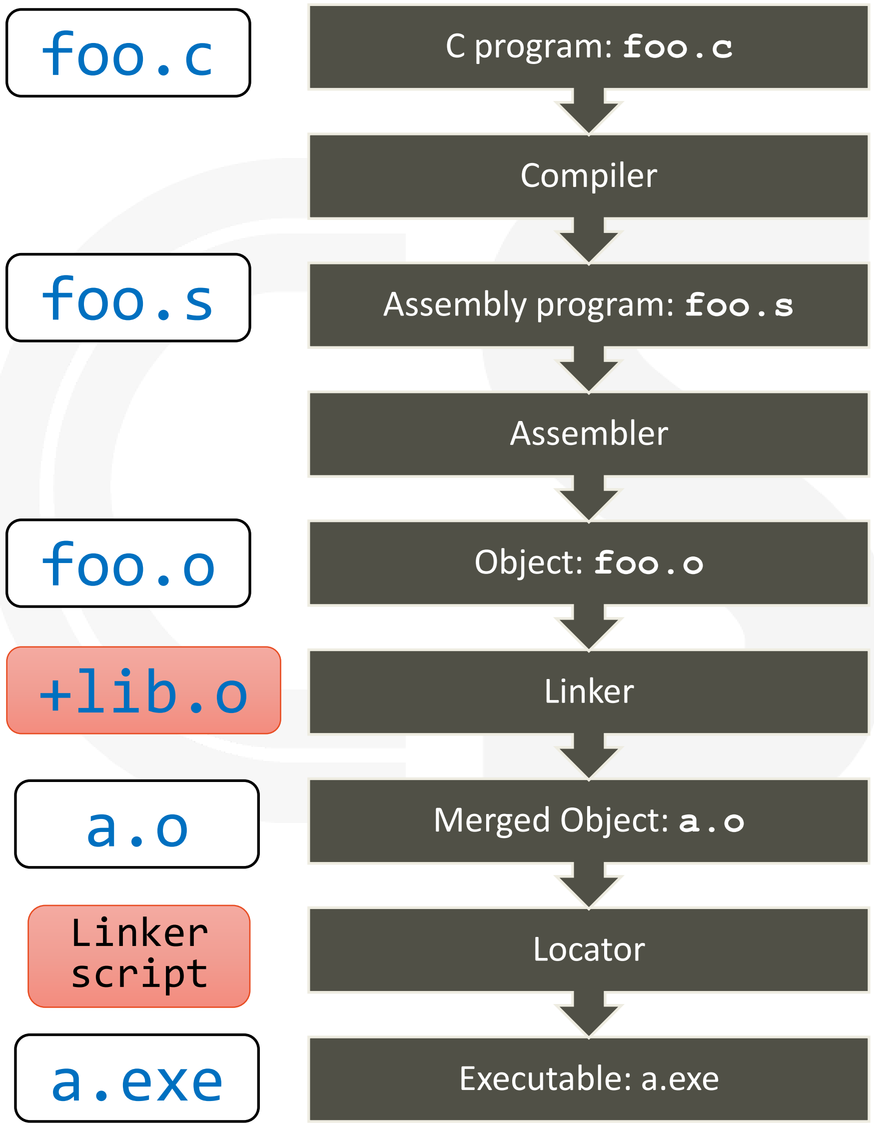
- **Text:**
  - Program code
- **Static data:**
  - Global variables, e.g., static variables in C, constant arrays and strings
  - A global pointer (`GP`) is used initialized to address allowing ±offsets into this segment
- **Dynamic data:**
  - a.k.a., "**heap**"
  - e.g., `malloc` in C, `new` in Java
- **Stack:**
  - Automatic storage for managing procedure called



Source: P&H, Ch. 2

Steps in Compiling and Running a C Program

- **Linker**
  - Input: Several Object code files (e.g., `foo.o`, `lib.o`)
  - Output: Merged object file (`a.o`)
  - Combines several `.o` files into a single object file
  - Enables separate compilation of files
    - Changes to one file do not require recompilation of the whole program (e.g., Linux source > 20 M lines of code!)
- **Locator**
  - Input: Linked object file (`a.o`), linker script
  - Output: Executable (`a.exe`)
  - Replace relative addresses with actual addresses
  - Linker script tells locator how to assign memory



Static vs. Dynamically Linked Libraries

- **What we've described is the traditional way: statically-linked approach**
  - Library is now part of the executable, so if the library updates, we don't get the fix (have to recompile if we have source)
  - Includes the entire library even if not all of it will be used
  - Executable is self-contained
- **Alternative is *dynamically linked libraries* (DLL), common on Windows & UNIX platforms**
- **The Loader (OS) has to dynamically link the functions at runtime:**
  - The OS starts the dynamic linker
  - The dynamic linker starts the program, copies first time calls into memory
  - The programs are changed to point to the correct function

Measuring Performance

How do we measure performance?

- **Response time**
  - How long it takes to do a task
- **Throughput**
  - Total work done per unit time
    - e.g., tasks/transactions/… per hour
- **We’ll focus on response time for now…**
  - Define: Performance = 1/Execution Time
- **Elapsed time**
  - Total response time, including all aspects
    - Processing, I/O, OS overhead, idle time
- **CPU time**
  - Time spent processing a given job
    - Discounts I/O time, other jobs’ shares

Increasing Performance

- **CPU Time:**
  - Reduce number of clock cycles
  - Increase clock rate
- **CPU Clock Cycles**
  - Reduce Instruction Count
  - Reduce CPI
- **Instruction Count**
  - Determined by program, ISA and compiler
- **Average cycles per instruction**
  - Determined by CPU hardware
  - Average CPI affected by instruction mix

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

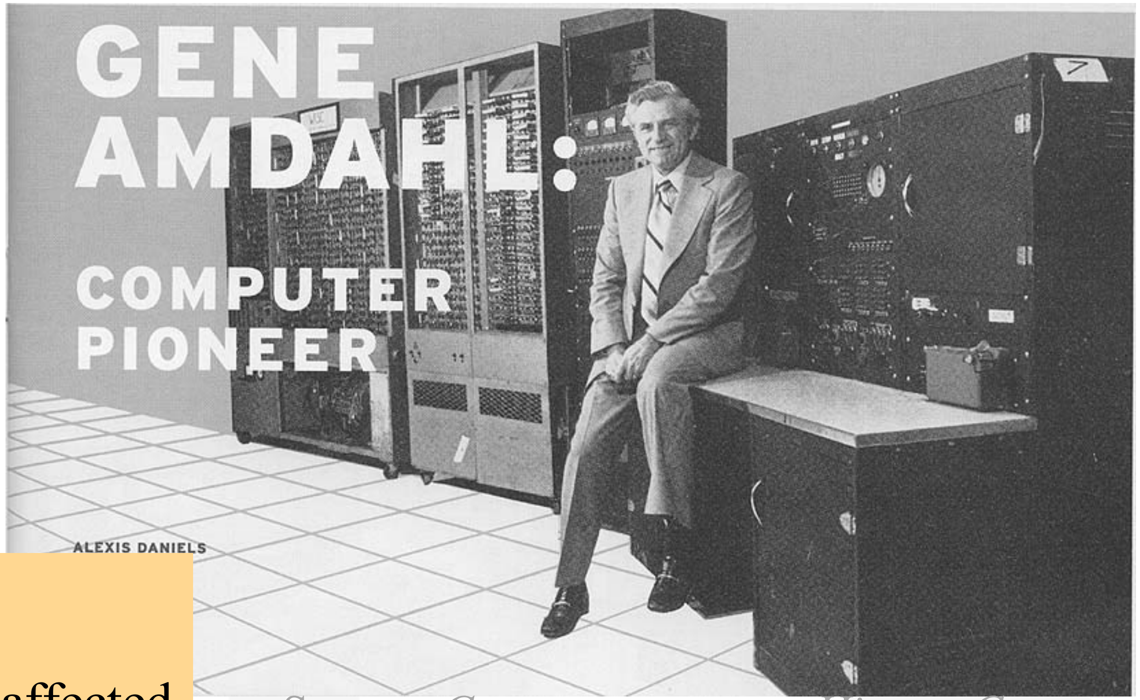
The Iron Law

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- **Performance depends on**
  - **Algorithm**: affects IC, possibly CPI
  - **Programming language**: affects IC, CPI
  - **Compiler**: affects IC, CPI
  - **Instruction set architecture**: affects IC, CPI, T_c

Amdahl's Law

- **Improving an aspect of a computer and expecting a proportional improvement in overall performance**



Source: *Computer museum History Center*

$$T_{improved} = \frac{T_{affected}}{improvement\ factor} + T_{unaffected}$$

- **Corollary: make the common case fast**
  - *Common* == "most time consuming", not necessarily "most frequent"
  - The *uncommon case* doesn't make much difference
  - But the common case changes.
    With optimization, common becomes uncommon and vice versa.

Amdahl's Law Example

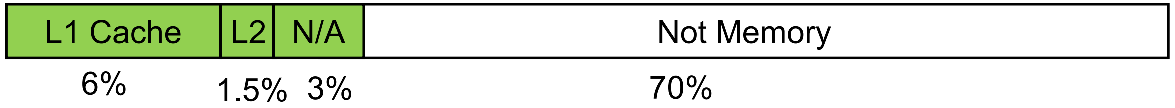
- **How does applying cache to a processor help?**
  - Memory operations currently take 30% of execution time.
  - By adding a cache, 80% of the memory operations gets a 4X speedup
  - What is the overall speedup?

$$0.7+\frac{(0.3\cdot 0.8)}{4}+(0.3\cdot 0.2)=0.82\mapsto 1.22\times$$

- **Now we add an L2 cache**
  - Speed up half of the remaining 20% by a factor of 2.

$$0.7+\frac{(0.3\cdot 0.8)}{4}+\frac{(0.\,[illegible]\cdot 0.2)}{[illegible]}=0.775\mapsto 1.29\times$$

- **This is wrong!**
  - We forgot the part of that the L2 cache didn't get optimized!



$$0.7+\frac{0.24}{4}+\frac{(0.5\cdot 0.06)}{2}+(0.5\cdot 0.06)=0.805\mapsto 1.24\times$$

MIPS as a Performance Metric

- **MIPS: Millions of Instructions Per Second**

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

$$= \frac{\text{Instruction count}}{\frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

- **Doesn't account for**
  - Differences in ISAs between computers
  - Differences in complexity between instructions
- **CPI varies between programs on a given CPU**
  - No single MIPS for a given computer

Main References

- Patterson, Hennessy "Computer Organization and Design – The RISC-V Edition"
- Berkeley CS-61C, "Great Ideas in Computer Architecture"
- Patterson, Waterman "The RISC-V Reader"
- Wolf, "Computer as Components - Principles of Embedded Computing System Design," Elsevier 2012
- Barr, Massa "Programming Embedded Systems with C and GNU Development Tools", O'Reilly 2005