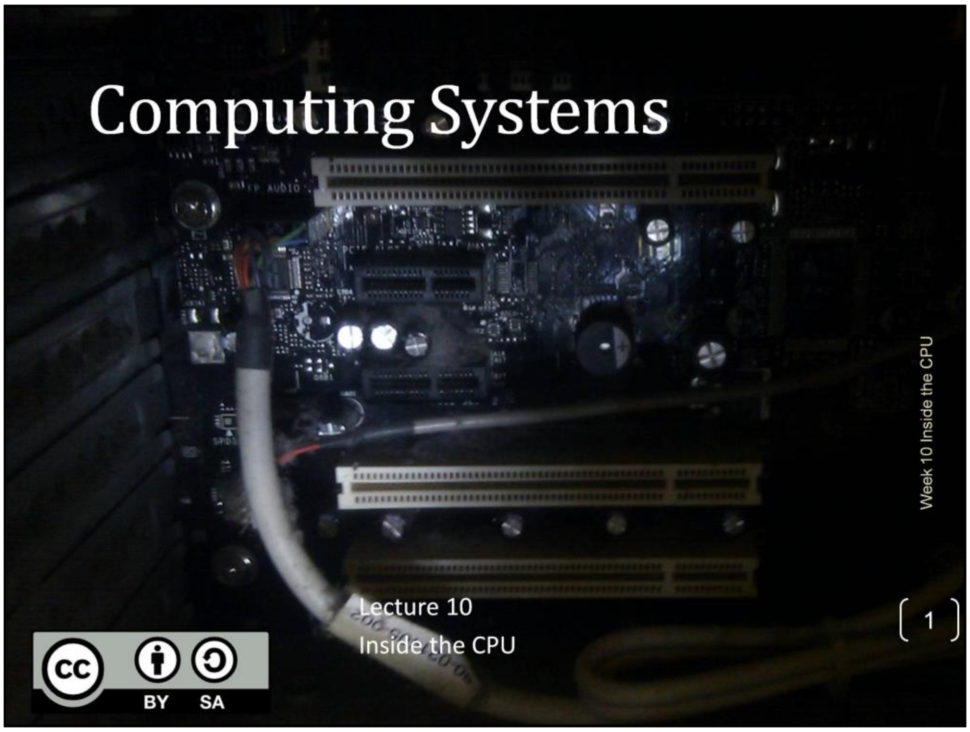
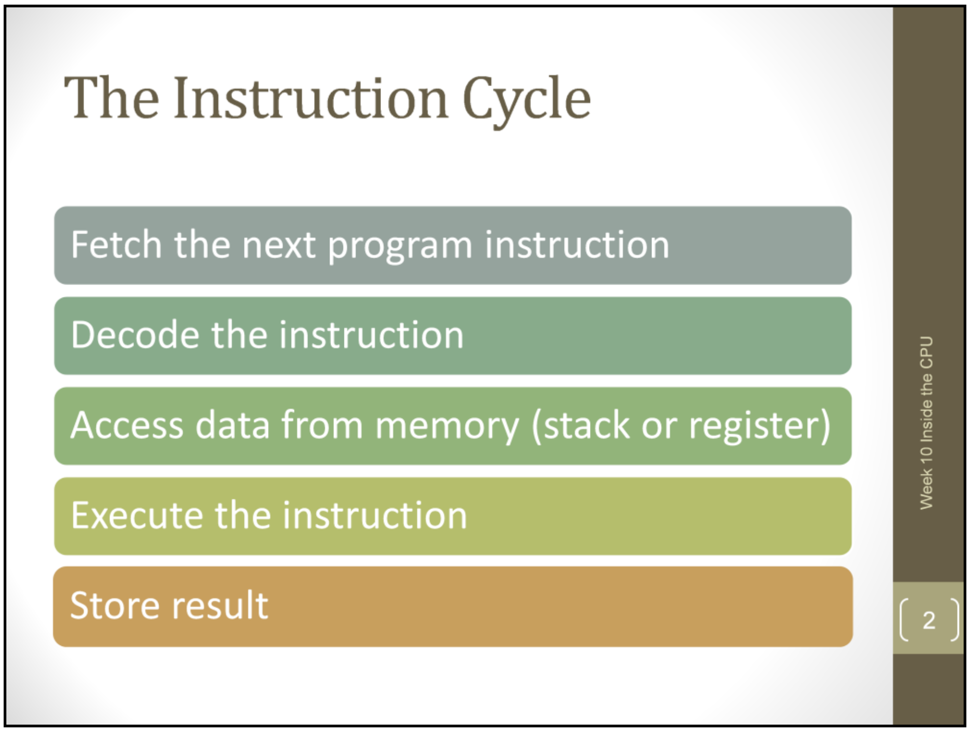


Computing Systems

Lecture 10
Inside the CPU



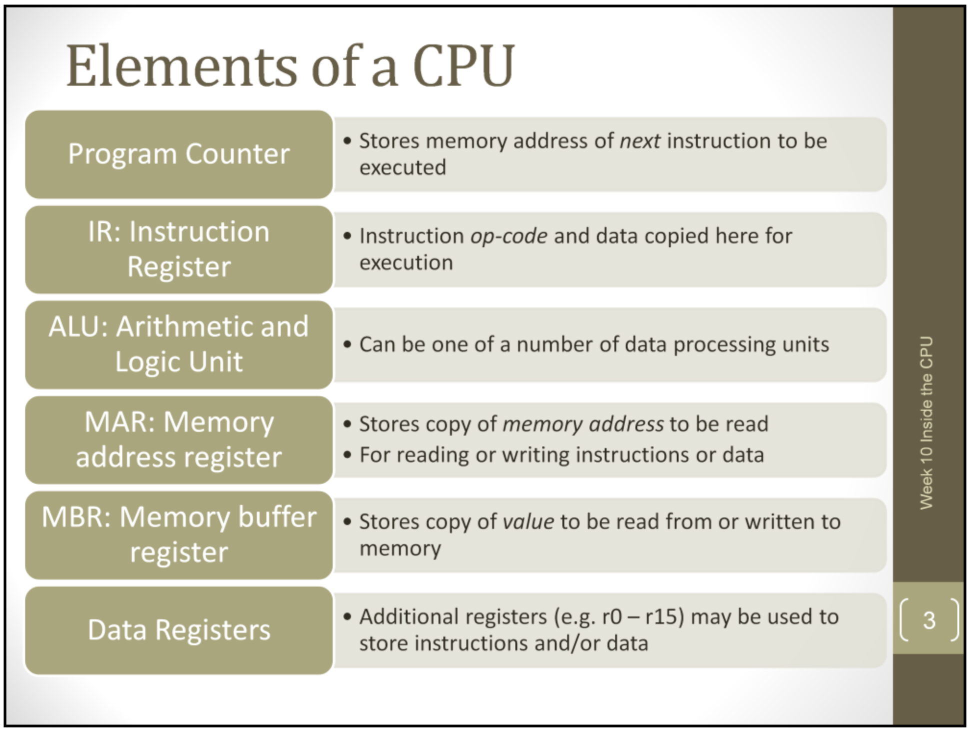
The Instruction Cycle

- Fetch the next program instruction
- Decode the instruction
- Access data from memory (stack or register)
- Execute the instruction
- Store result

Week 10 Inside the CPU

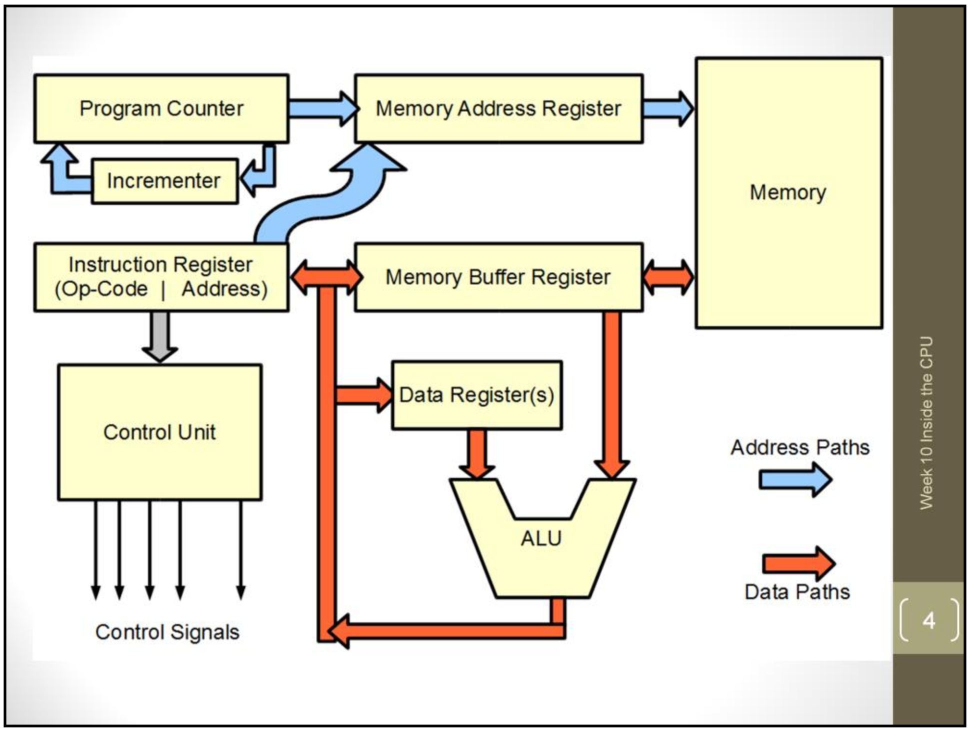
2

The time period during which one instruction is fetched from memory and executed when a computer is given an instruction in machine language. There are typically FIVE stages of an instruction cycle that the CPU carries out: Fetch; Decode; Access Data from Memory; Execute and Store result.

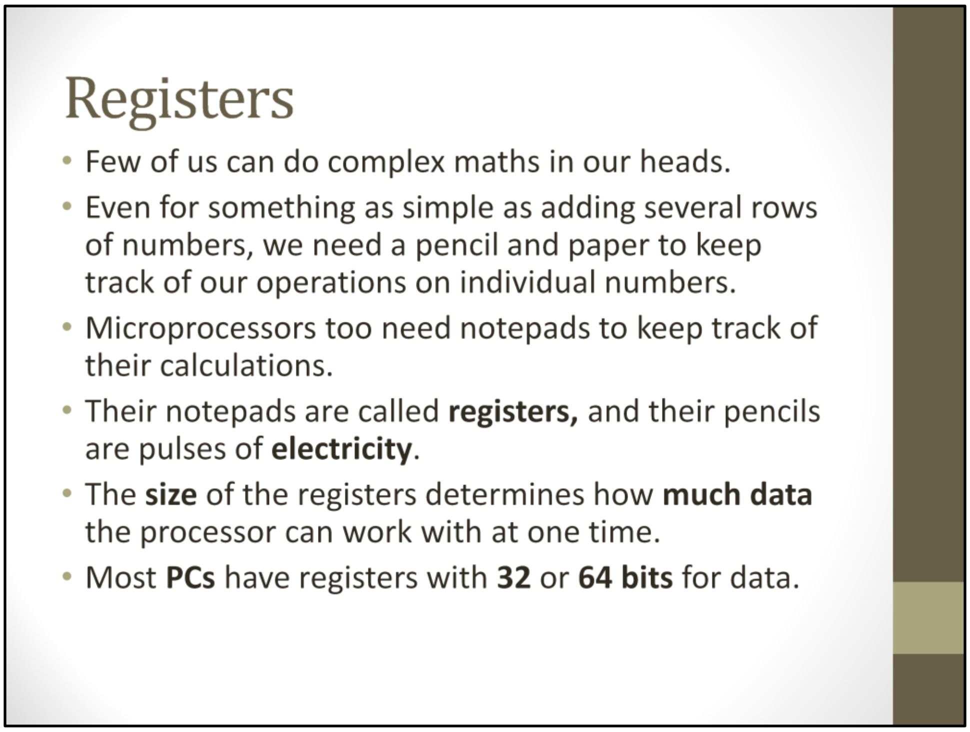


Elements of a CPU

Program Counter	• Stores memory address of *next* instruction to be executed
IR: Instruction Register	• Instruction *op-code* and data copied here for execution
ALU: Arithmetic and Logic Unit	• Can be one of a number of data processing units
MAR: Memory address register	• Stores copy of *memory address* to be read • For reading or writing instructions or data
MBR: Memory buffer register	• Stores copy of *value* to be read from or written to memory
Data Registers	• Additional registers (e.g. r0 – r15) may be used to store instructions and/or data

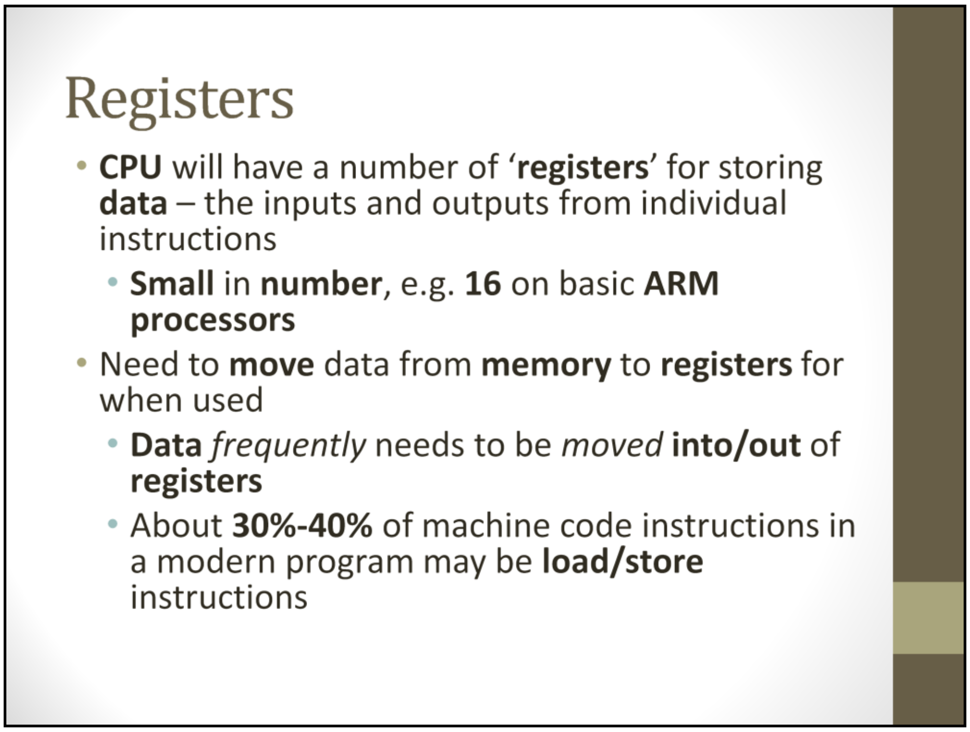


This diagram is based on Alan Clement’s schematics from http://www.alanclements.co.uk/Resources/CPU_FE/FetchExecute.htm (now a dead link)



Registers

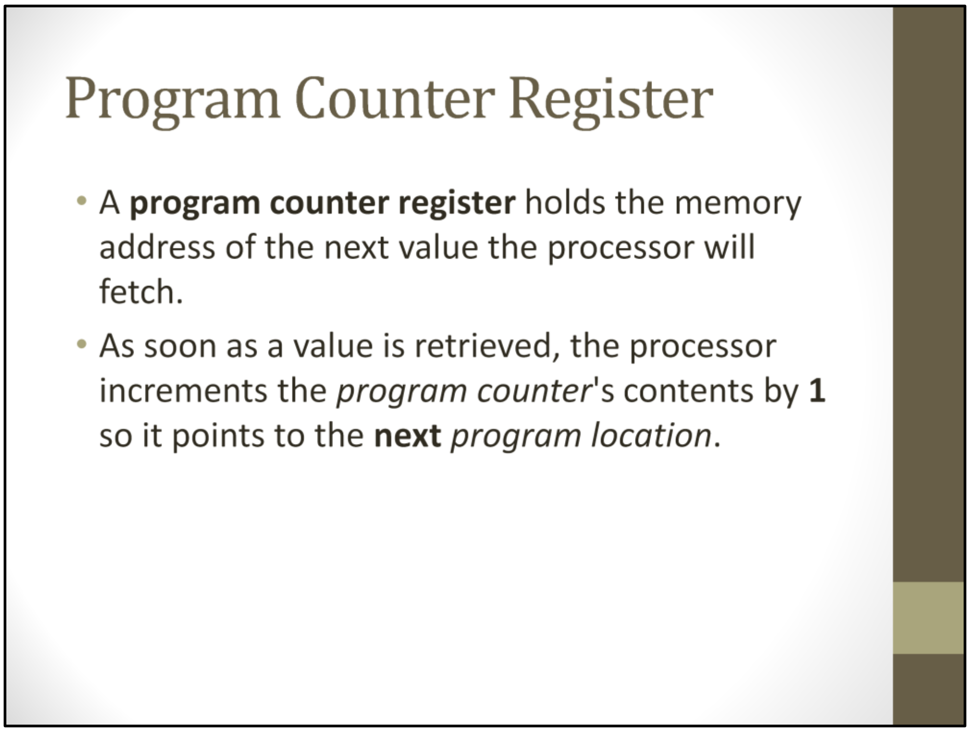
- Few of us can do complex maths in our heads.
- Even for something as simple as adding several rows of numbers, we need a pencil and paper to keep track of our operations on individual numbers.
- Microprocessors too need notepads to keep track of their calculations.
- Their notepads are called **registers,** and their pencils are pulses of **electricity**.
- The **size** of the registers determines how **much data** the processor can work with at one time.
- Most **PCs** have registers with **32** or **64 bits** for data.



Registers

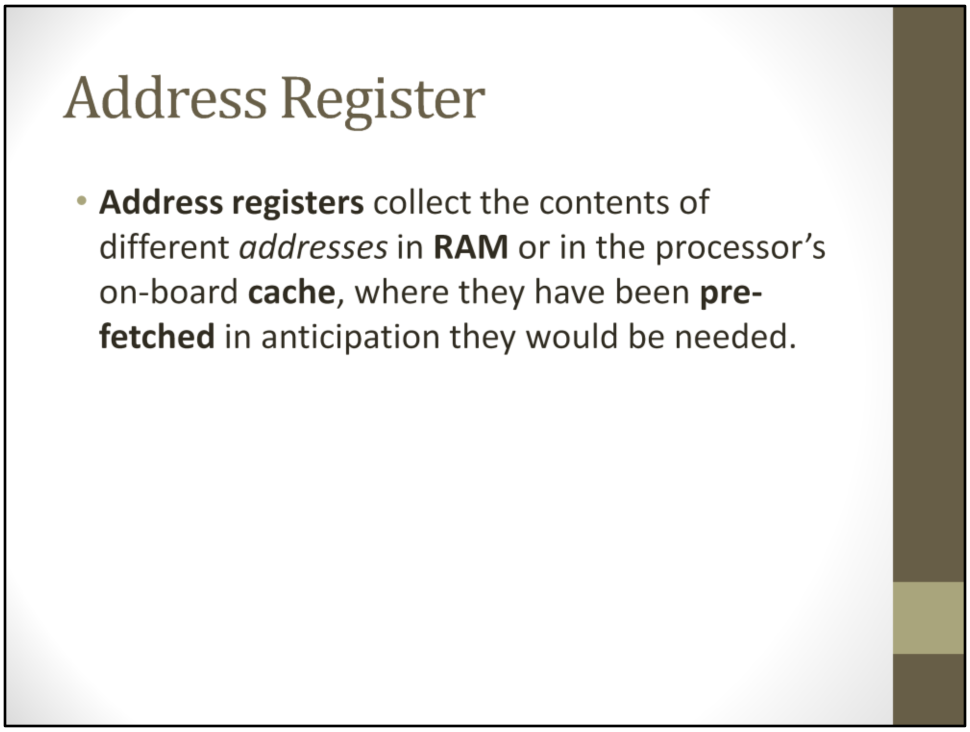
- **CPU** will have a number of ‘**registers**’ for storing **data** – the inputs and outputs from individual instructions
  - **Small** in **number**, e.g. **16** on basic **ARM processors**
- Need to **move** data from **memory** to **registers** for when used
  - **Data** *frequently* needs to be *moved* **into/out** of **registers**
  - About **30%-40%** of machine code instructions in a modern program may be **load/store** instructions

1970’s analysis was that about 45% of instructions were load/store – see PCH p328-329



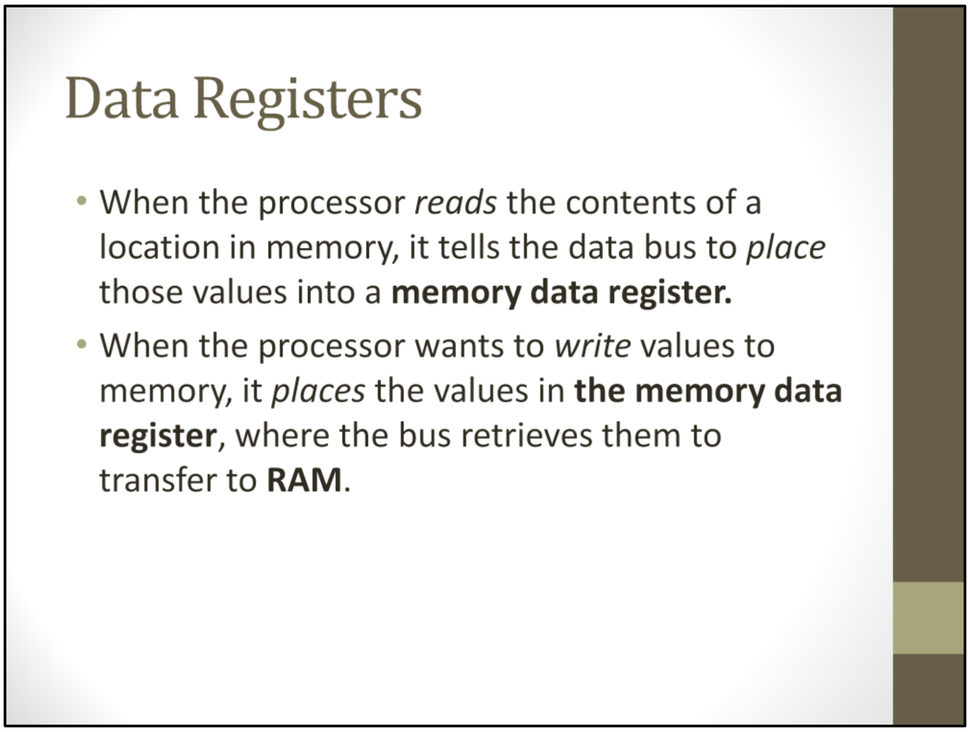
Program Counter Register

- A **program counter register** holds the memory address of the next value the processor will fetch.
- As soon as a value is retrieved, the processor increments the *program counter*'s contents by **1** so it points to the **next** *program location*.



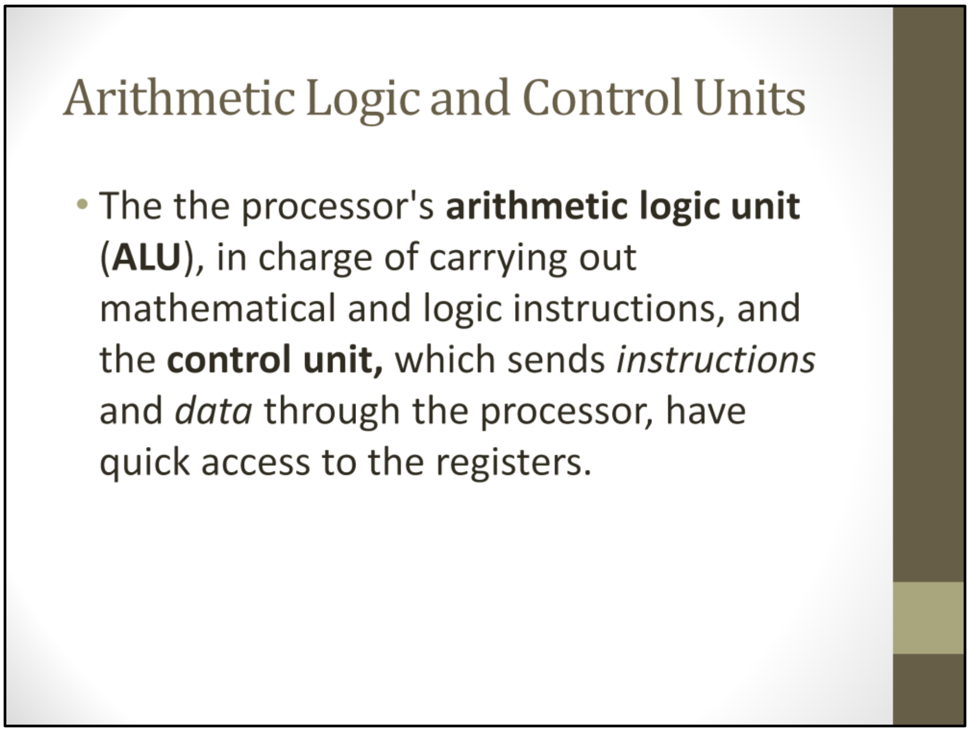
Address Register

- **Address registers** collect the contents of different *addresses* in **RAM** or in the processor's on-board **cache**, where they have been **pre-fetched** in anticipation they would be needed.



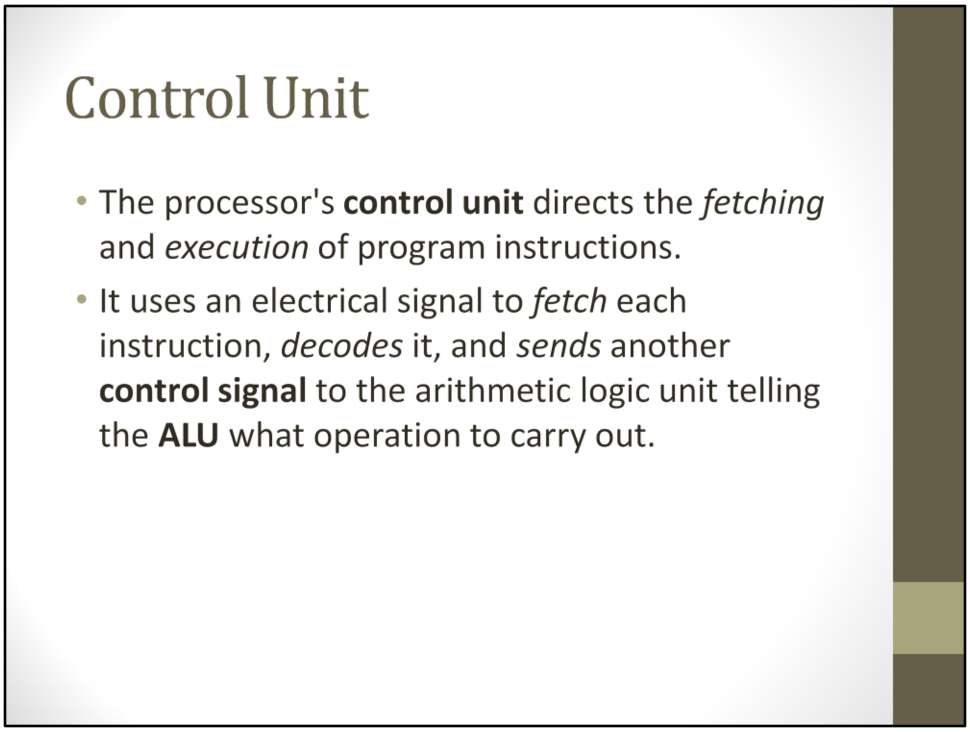
Data Registers

- When the processor *reads* the contents of a location in memory, it tells the data bus to *place* those values into a **memory data register.**
- When the processor wants to *write* values to memory, it *places* the values in **the memory data register**, where the bus retrieves them to transfer to **RAM**.



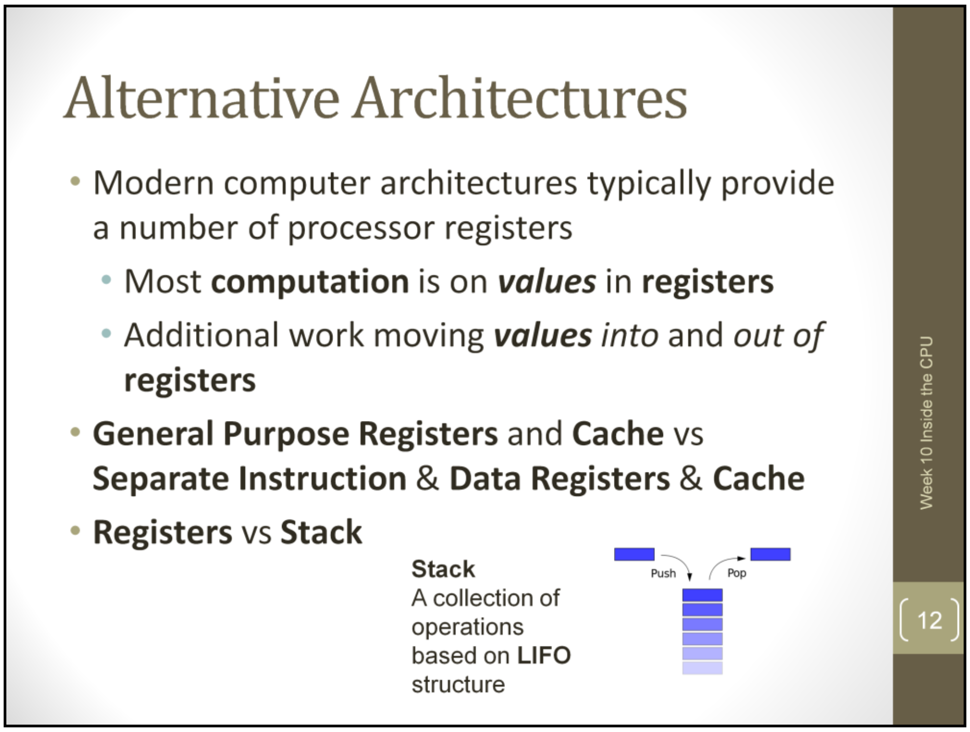
Arithmetic Logic and Control Units

- The the processor's **arithmetic logic unit** (**ALU**), in charge of carrying out mathematical and logic instructions, and the **control unit,** which sends *instructions* and *data* through the processor, have quick access to the registers.



Control Unit

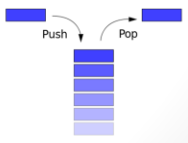
- The processor's **control unit** directs the *fetching* and *execution* of program instructions.
- It uses an electrical signal to *fetch* each instruction, *decodes* it, and *sends* another **control signal** to the arithmetic logic unit telling the **ALU** what operation to carry out.



Alternative Architectures

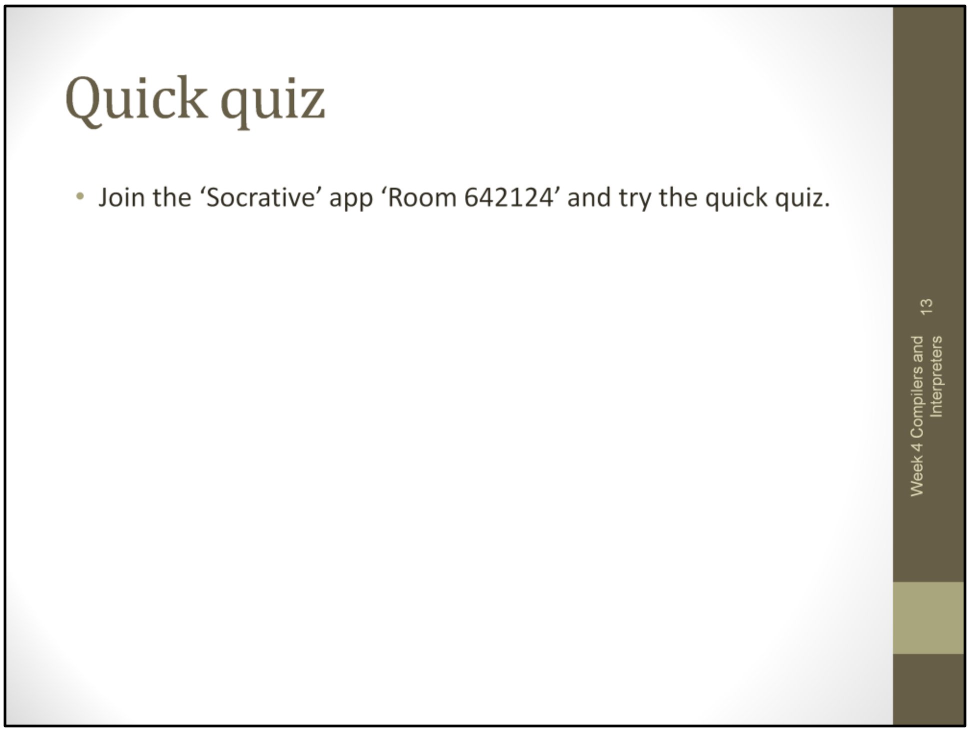
- Modern computer architectures typically provide a number of processor registers
  - Most **computation** is on ***values*** in **registers**
  - Additional work moving ***values*** *into* and *out of* **registers**
- **General Purpose Registers** and **Cache** vs **Separate Instruction** & **Data Registers** & **Cache**
- **Registers** vs **Stack**

Stack
A collection of operations based on **LIFO** structure



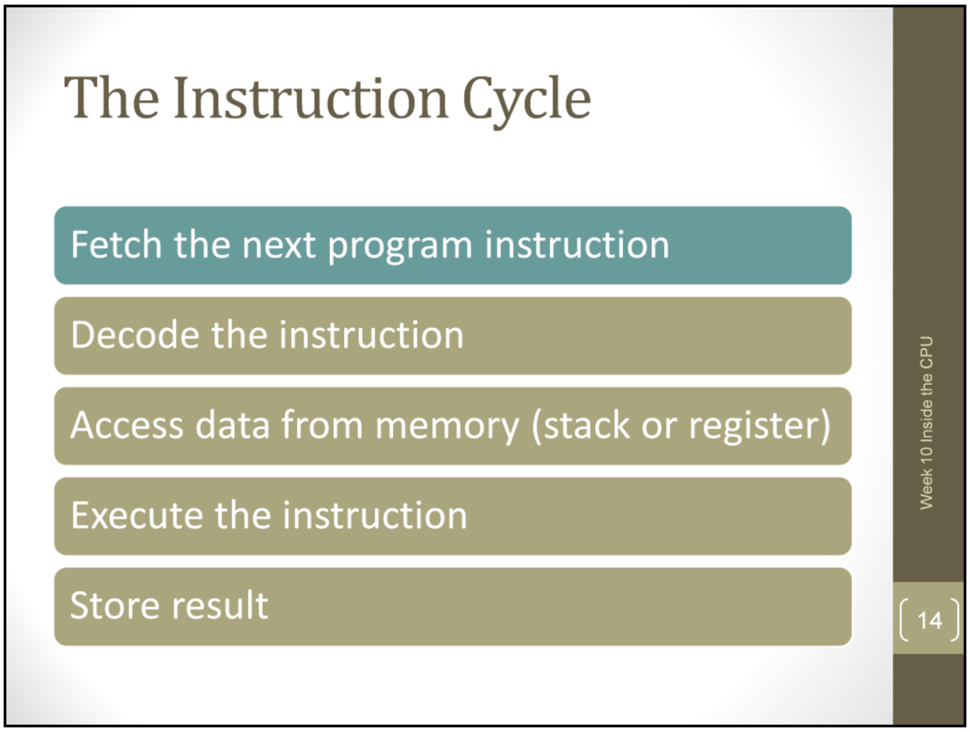
In **computer science**, a **stack** is a particular kind of abstract **data typ**e or **collection** in which the principal (or only) operations on the collection are the addition of an entity to the collection, known as push and removal of an entity, known as pop. The relation between the push and pop operations is such that the stack is a Last-In-First-Out (LIFO) data structure. In a LIFO data structure, the last element added to the structure must be the first one to be removed. This is equivalent to the requirement that, considered as a linear data structure, or more abstractly a sequential collection, the push and pop operations occur only

at one end of the structure, referred to as
the top of the stack.



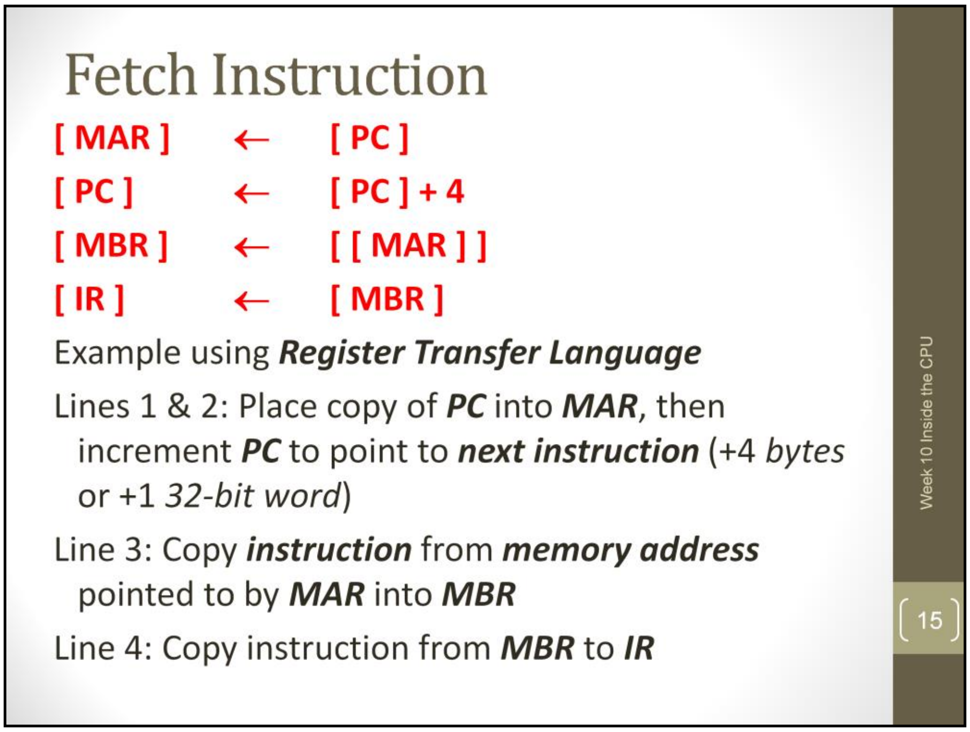
Quick quiz

- Join the ‘Socrative’ app ‘Room 642124’ and try the quick quiz.



The Instruction Cycle

- Fetch the next program instruction
- Decode the instruction
- Access data from memory (stack or register)
- Execute the instruction
- Store result



Fetch Instruction

[MAR]	←	[PC]
[PC]	←	[PC] + 4
[MBR]	←	[[MAR]]
[IR]	←	[MBR]

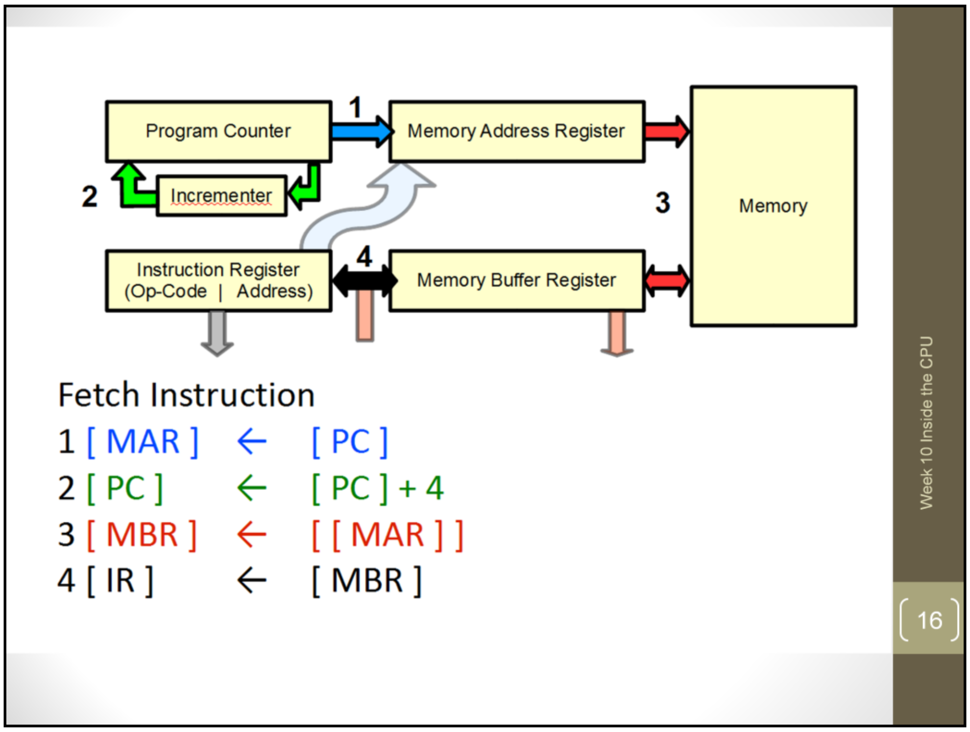
Example using ***Register Transfer Language***

Lines 1 & 2: Place copy of ***PC*** into ***MAR***, then increment ***PC*** to point to ***next instruction*** (+4 *bytes* or +1 *32-bit word*)

Line 3: Copy ***instruction*** from ***memory address*** pointed to by ***MAR*** into ***MBR***

Line 4: Copy instruction from ***MBR*** to ***IR***

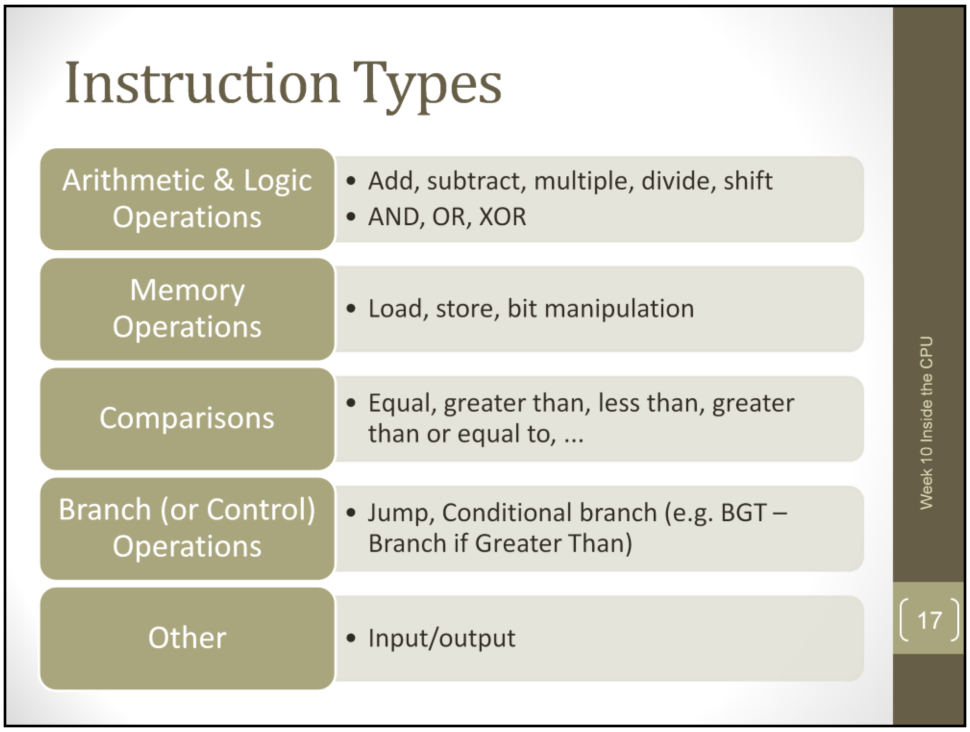
This example is given using Register Transfer Language (RTL) – see http://www.alanclements.co.uk/Resources/CPU_FE/FetchExecute.htm , Principles of Computer Hardware Chapter 6 (4th Edition)



Fetch Instruction

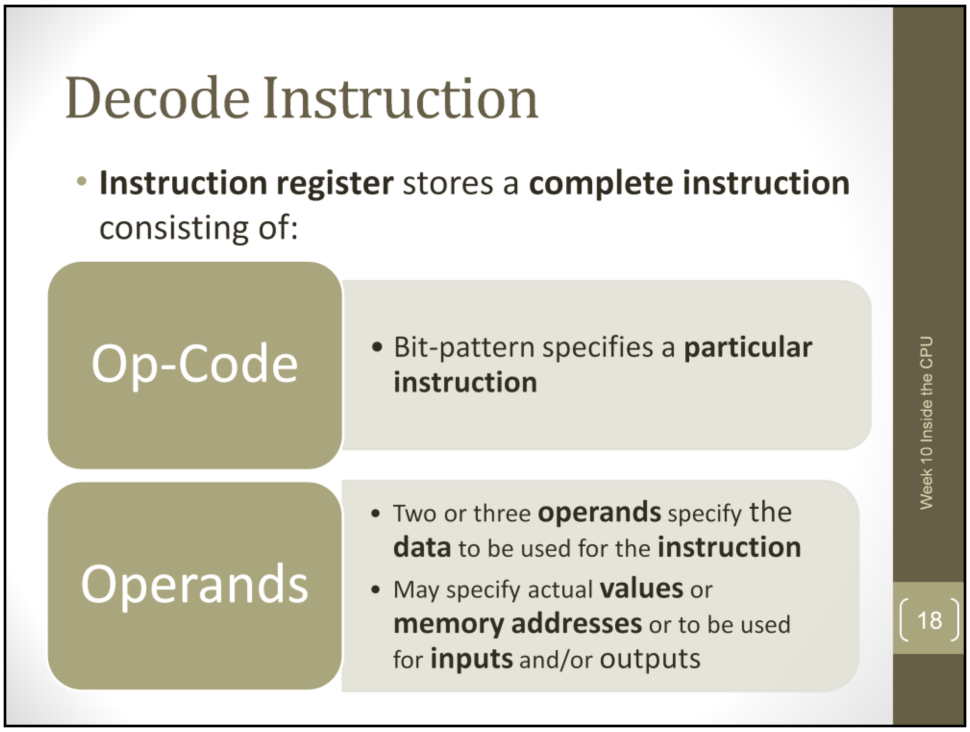
1 [MAR] ← [PC]
2 [PC] ← [PC] + 4
3 [MBR] ← [[MAR]]
4 [IR] ← [MBR]

This diagram is based on Alan Clement's schematics from http://www.alanclements.co.uk/Resources/CPU_FE/FetchExecute.htm (now a dead link)



Instruction Types

Arithmetic & Logic Operations	• Add, subtract, multiple, divide, shift • AND, OR, XOR
Memory Operations	• Load, store, bit manipulation
Comparisons	• Equal, greater than, less than, greater than or equal to, ...
Branch (or Control) Operations	• Jump, Conditional branch (e.g. BGT – Branch if Greater Than)
Other	• Input/output



Decode Instruction

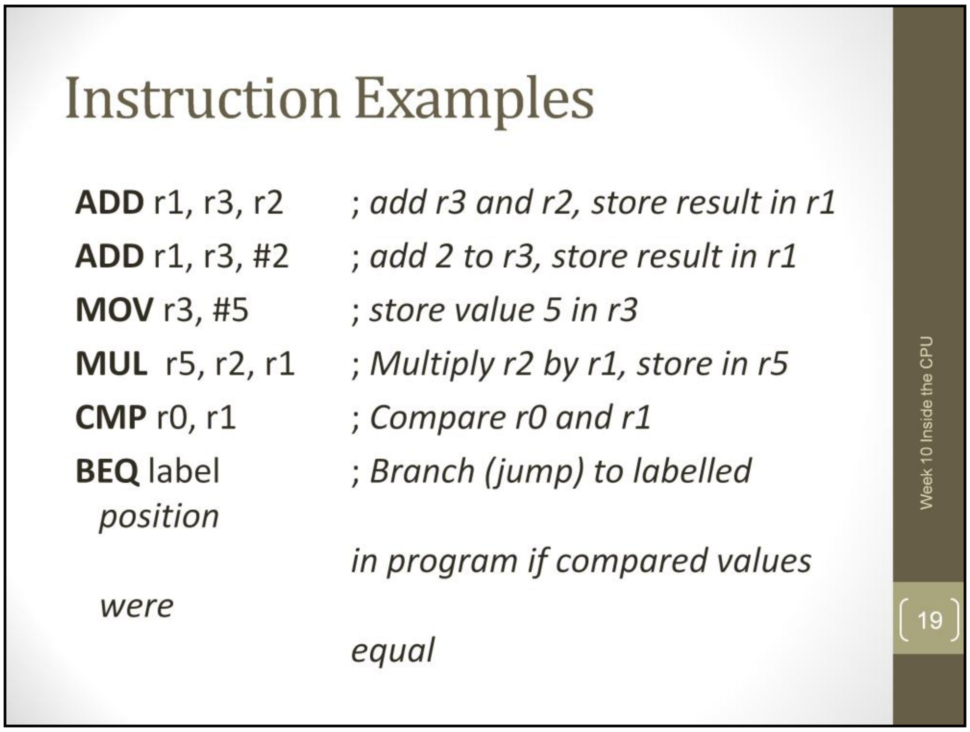
- **Instruction register** stores a **complete instruction** consisting of:

Op-Code

- Bit-pattern specifies a **particular instruction**

Operands

- Two or three **operands** specify the **data** to be used for the **instruction**
- May specify actual **values** or **memory addresses** or to be used for **inputs** and/or outputs



Instruction Examples

ADD r1, r3, r2 ; *add r3 and r2, store result in r1*

ADD r1, r3, #2 ; *add 2 to r3, store result in r1*

MOV r3, #5 ; *store value 5 in r3*

MUL r5, r2, r1 ; *Multiply r2 by r1, store in r5*

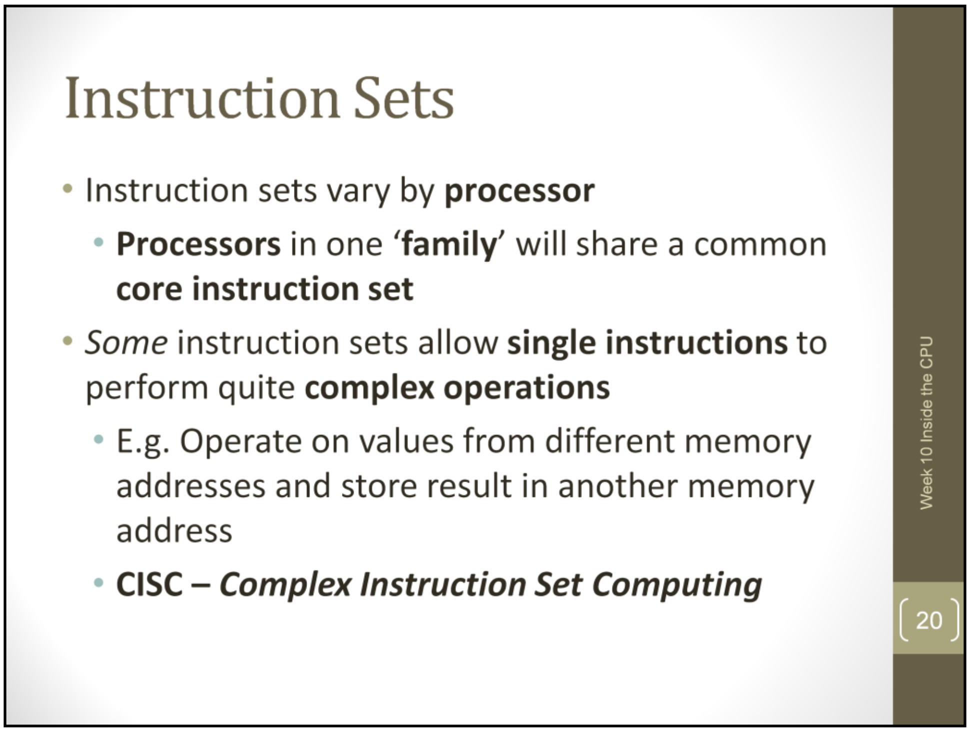
CMP r0, r1 ; *Compare r0 and r1*

BEQ label ; *Branch (jump) to labelled position in program if compared values were equal*

Week 10 Inside the CPU

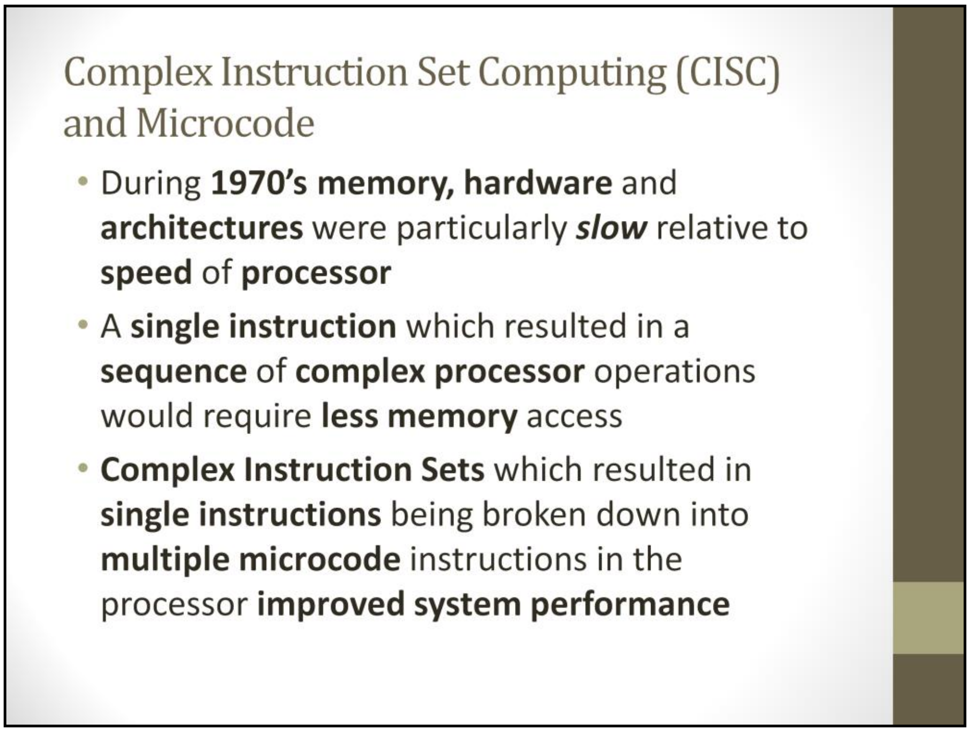
19

r1, r2, etc are the registers



Instruction Sets

- Instruction sets vary by **processor**
  - **Processors** in one '**family**' will share a common **core instruction set**
- *Some* instruction sets allow **single instructions** to perform quite **complex operations**
  - E.g. Operate on values from different memory addresses and store result in another memory address
  - **CISC –** ***Complex Instruction Set Computing***

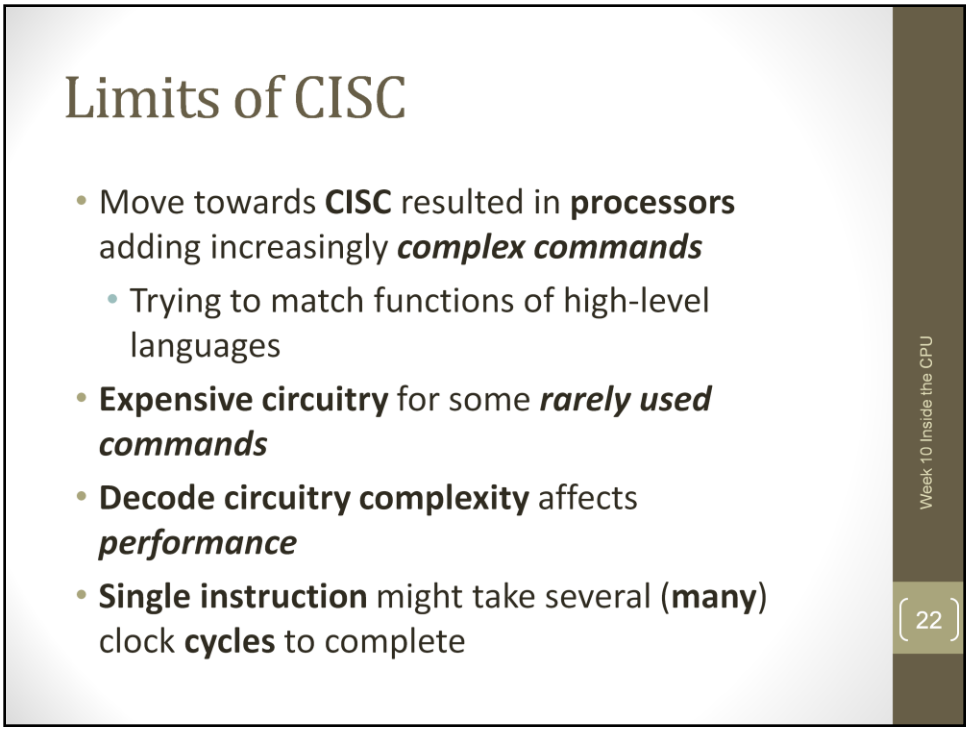


Complex Instruction Set Computing (CISC) and Microcode

- During **1970's memory, hardware** and **architectures** were particularly ***slow*** relative to **speed** of **processor**
- A **single instruction** which resulted in a **sequence** of **complex processor** operations would require **less memory** access
- **Complex Instruction Sets** which resulted in **single instructions** being broken down into **multiple microcode** instructions in the processor **improved system performance**

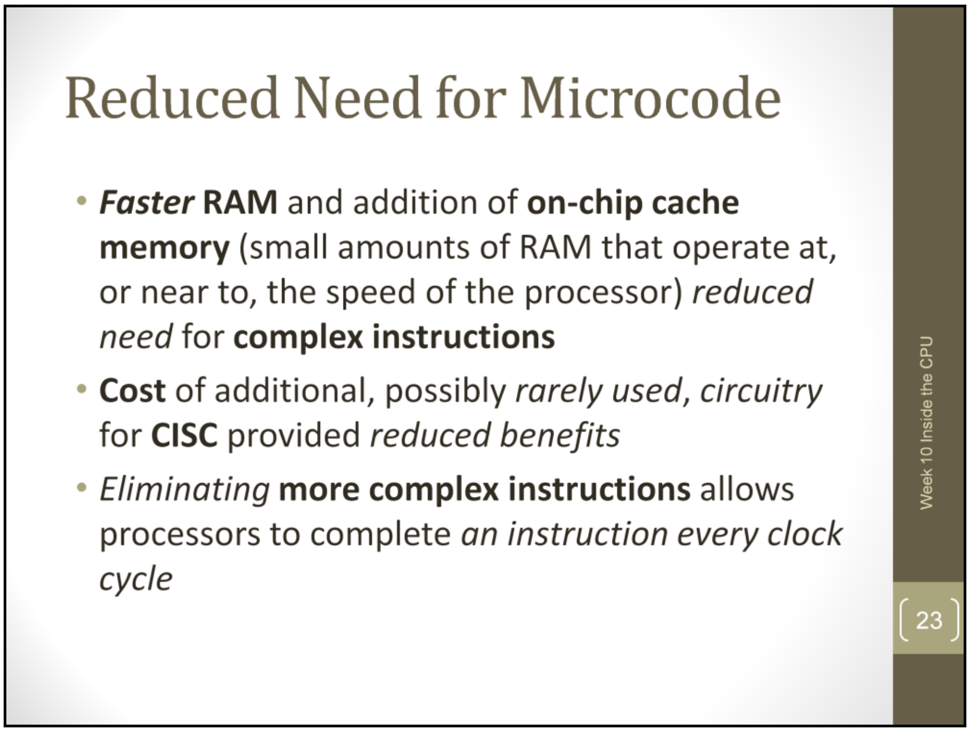
The key point of this is simply to reinforce the notion that a single assembly language/machine code instruction can translate into many different operations that need to be carried out on the processor. A detailed discussion of microcode is *not* required!

The improved system performance referred to in the last point is an overall improvement due to a decreased need to fetch instructions from main memory.



Limits of CISC

- Move towards **CISC** resulted in **processors** adding increasingly ***complex commands***
  - Trying to match functions of high-level languages
- **Expensive circuitry** for some ***rarely used commands***
- **Decode circuitry complexity** affects ***performance***
- **Single instruction** might take several (**many**) clock **cycles** to complete



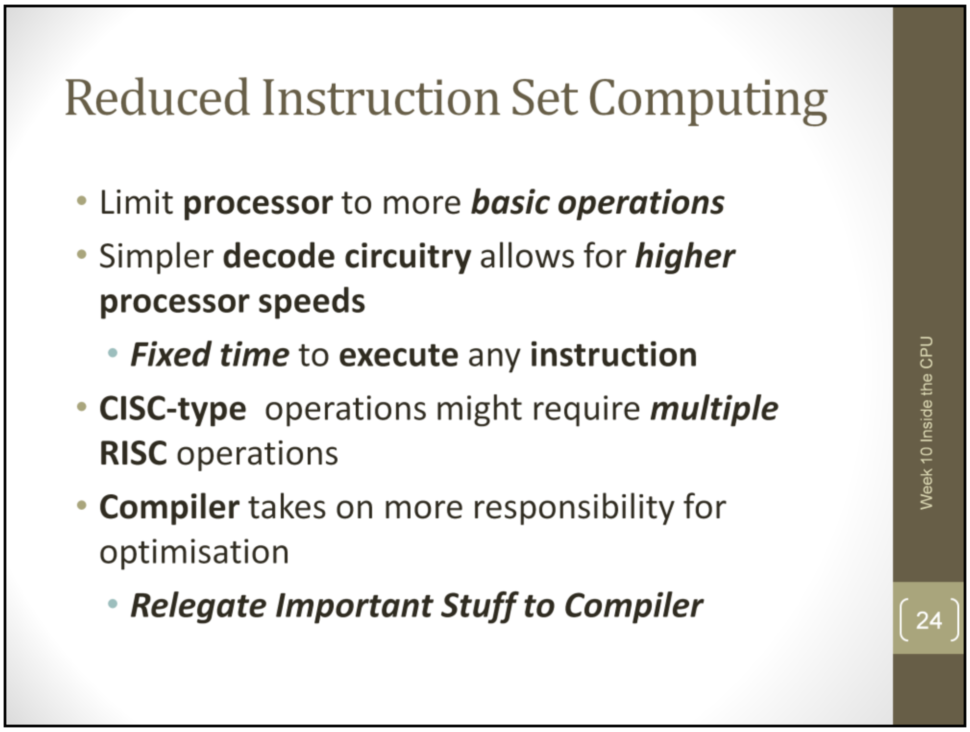
Reduced Need for Microcode

- ***Faster* RAM** and addition of **on-chip cache memory** (small amounts of RAM that operate at, or near to, the speed of the processor) *reduced need* for **complex instructions**
- **Cost** of additional, possibly *rarely used*, *circuitry* for **CISC** provided *reduced benefits*
- *Eliminating* **more complex instructions** allows processors to complete *an instruction every clock cycle*

Week 10 Inside the CPU

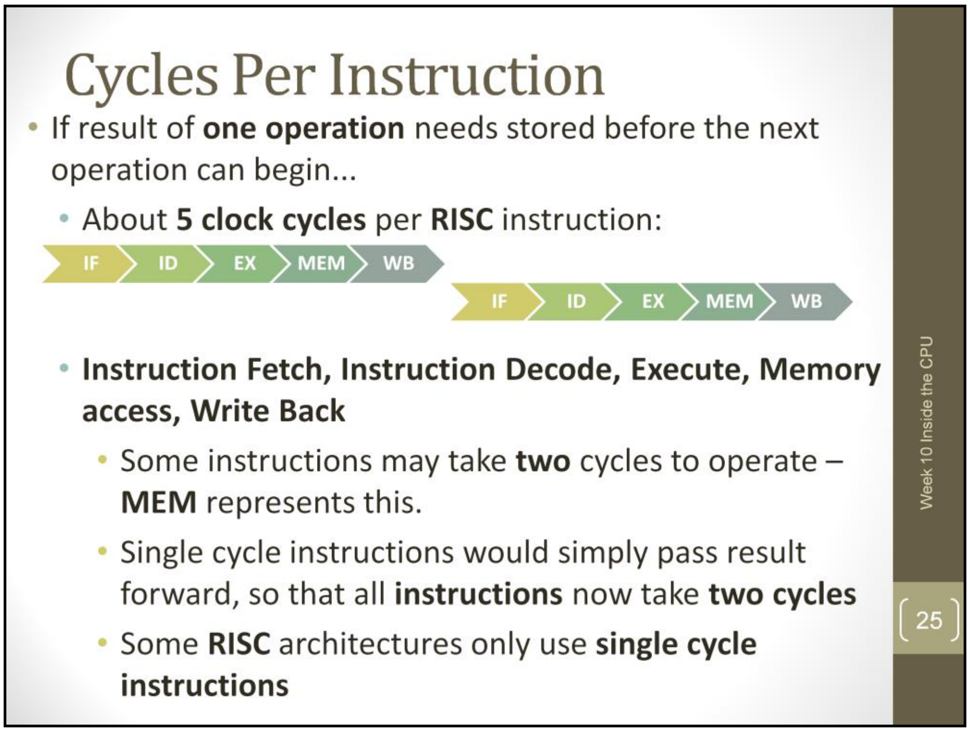
23

Essential point here is that micro-code no longer provides the scale of benefit it once did: most instruction fetches are from L1 or L2 cache, which is not as slow as a fetch from main memory. Benefit of micro-code now offset by the additional complexity of such systems and by the extra processor cycles required to complete a single complex instruction



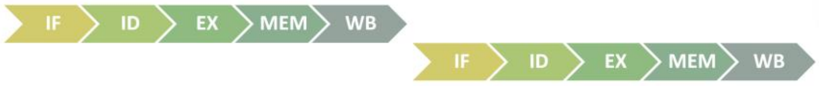
Reduced Instruction Set Computing

- Limit **processor** to more ***basic operations***
- Simpler **decode circuitry** allows for ***higher* processor speeds**
  - ***Fixed time*** to **execute** any **instruction**
- **CISC-type** operations might require ***multiple* RISC** operations
- **Compiler** takes on more responsibility for optimisation
  - ***Relegate Important Stuff to Compiler***

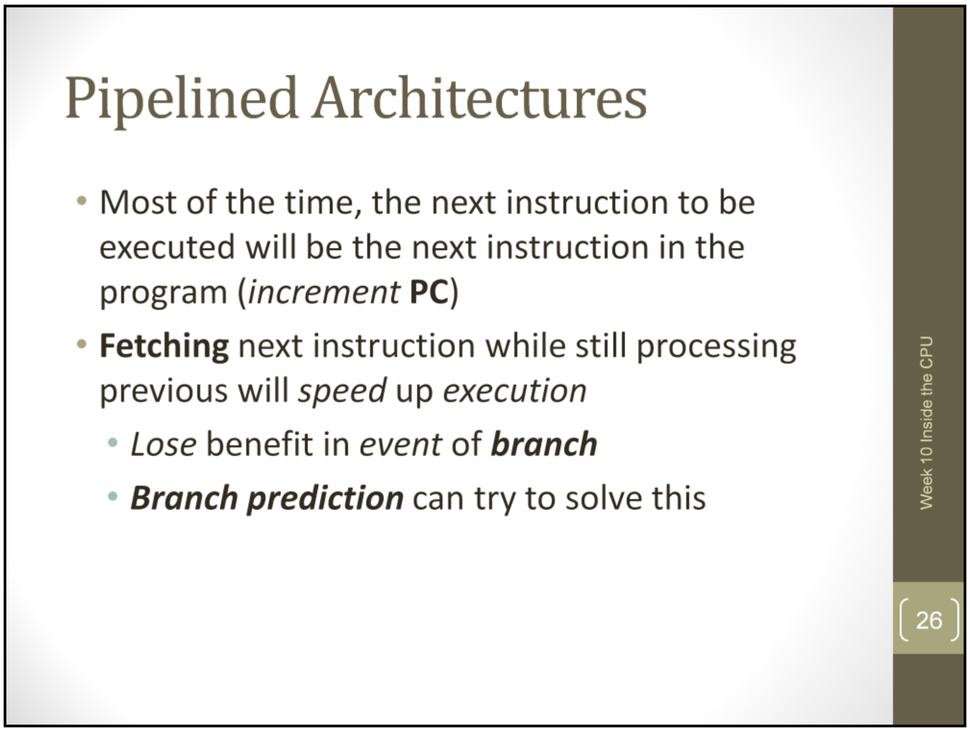


Cycles Per Instruction

- If result of **one operation** needs stored before the next operation can begin...
  - About **5 clock cycles** per **RISC** instruction:

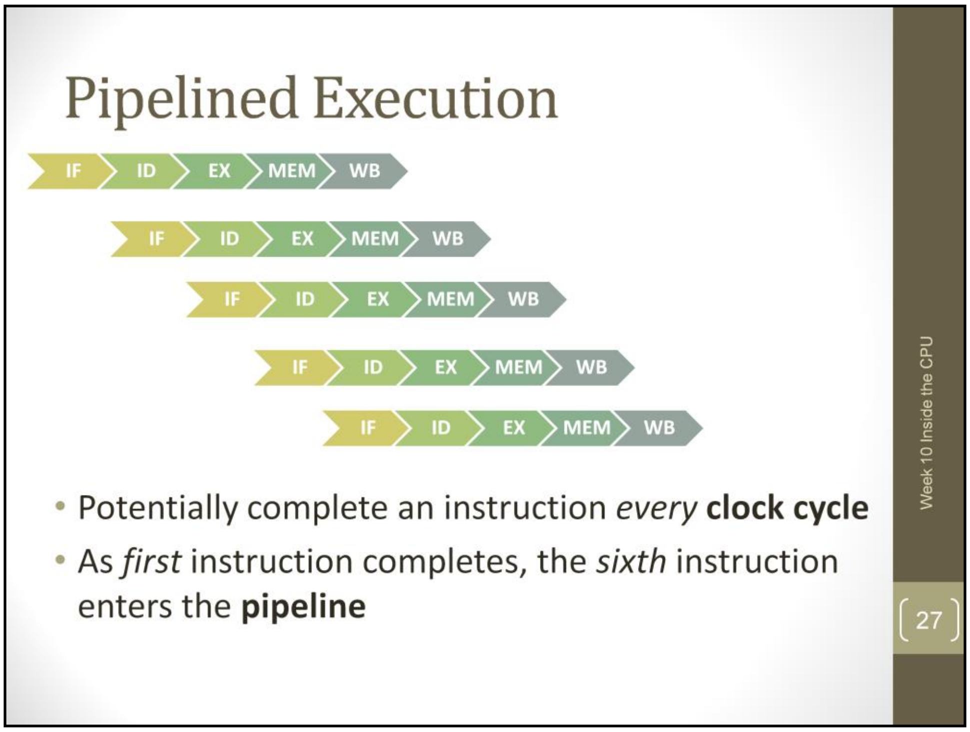


  - **Instruction Fetch, Instruction Decode, Execute, Memory access, Write Back**
    - Some instructions may take **two** cycles to operate – **MEM** represents this.
    - Single cycle instructions would simply pass result forward, so that all **instructions** now take **two cycles**
    - Some **RISC** architectures only use **single cycle instructions**

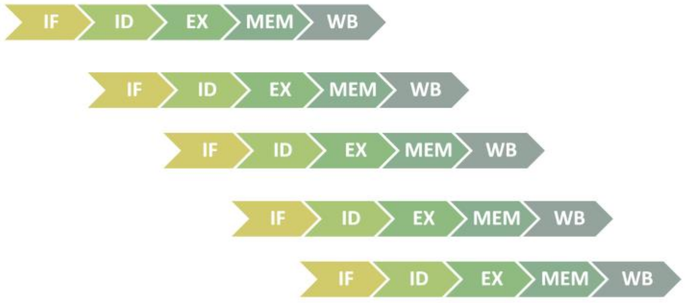


Pipelined Architectures

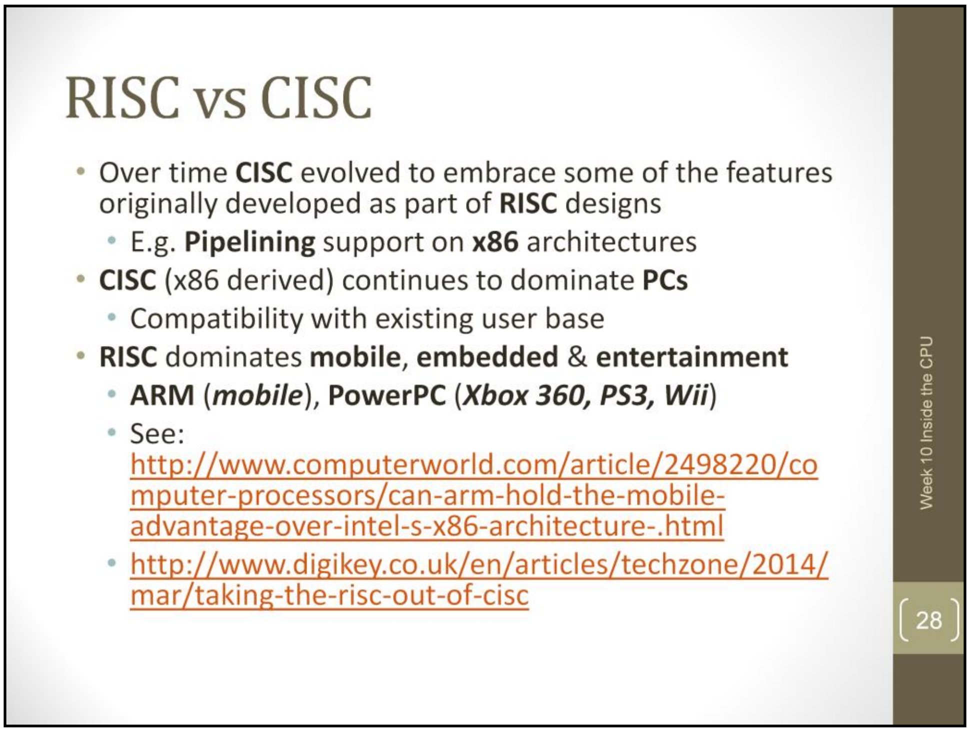
- Most of the time, the next instruction to be executed will be the next instruction in the program (*increment* **PC**)
- **Fetching** next instruction while still processing previous will *speed* up *execution*
  - *Lose* benefit in *event* of ***branch***
  - ***Branch prediction*** can try to solve this



Pipelined Execution

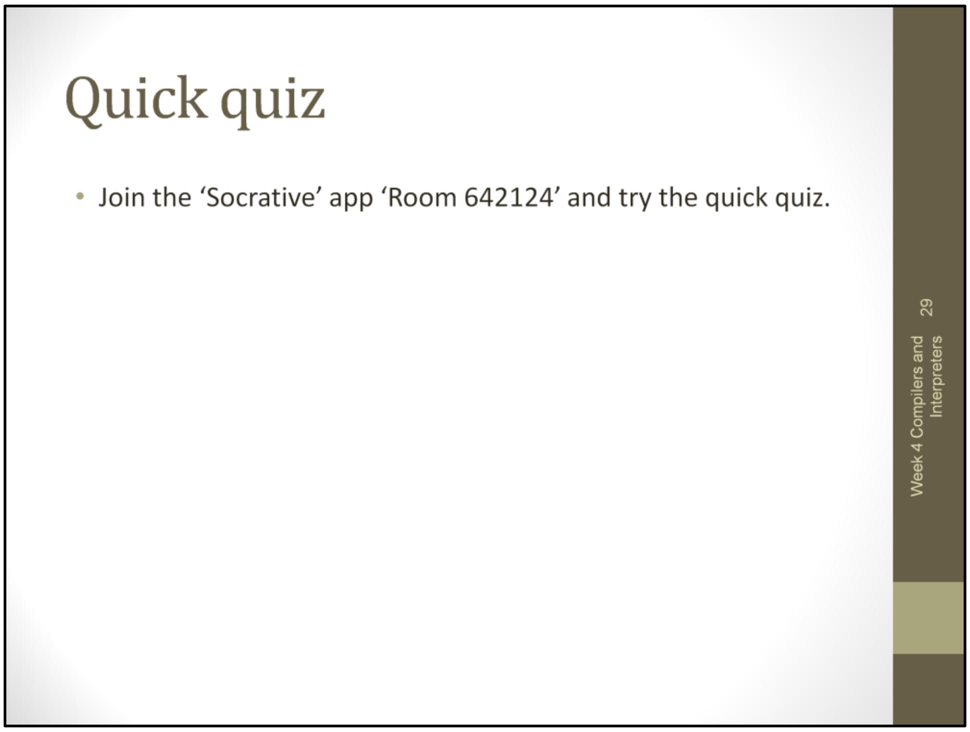


- Potentially complete an instruction *every* **clock cycle**
- As *first* instruction completes, the *sixth* instruction enters the **pipeline**



RISC vs CISC

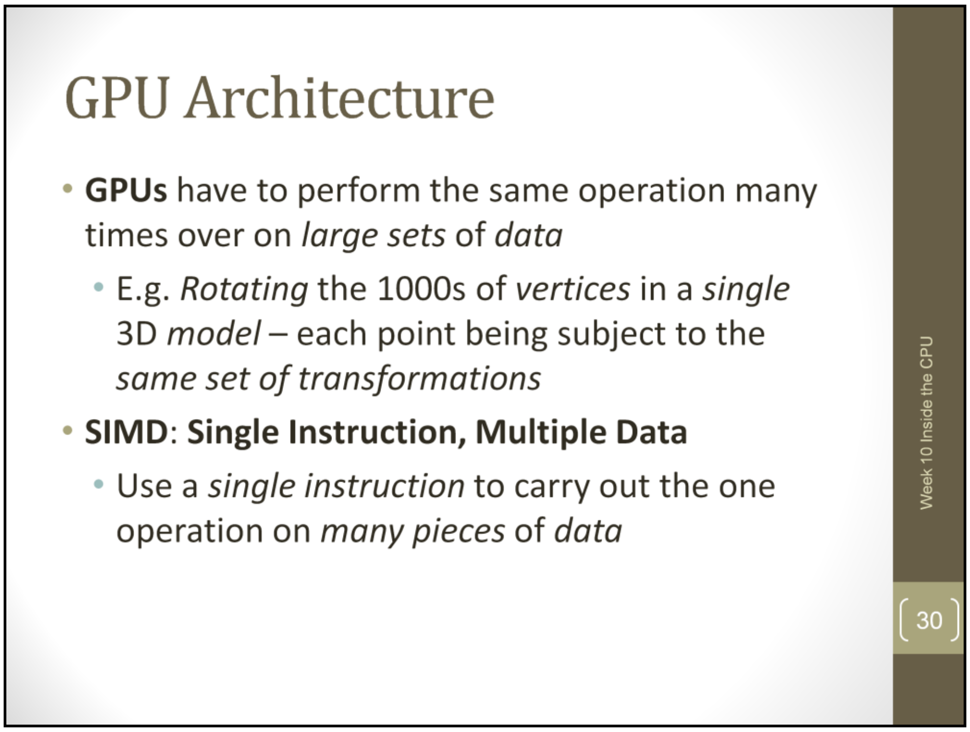
- Over time **CISC** evolved to embrace some of the features originally developed as part of **RISC** designs
  - E.g. **Pipelining** support on **x86** architectures
- **CISC** (x86 derived) continues to dominate **PCs**
  - Compatibility with existing user base
- **RISC** dominates **mobile**, **embedded** & **entertainment**
  - **ARM** (***mobile***), **PowerPC** (***Xbox 360, PS3, Wii***)
  - See: http://www.computerworld.com/article/2498220/computer-processors/can-arm-hold-the-mobile-advantage-over-intel-s-x86-architecture-.html
  - http://www.digikey.co.uk/en/articles/techzone/2014/mar/taking-the-risc-out-of-cisc



Quick quiz

- Join the ‘Socrative’ app ‘Room 642124’ and try the quick quiz.

Week 4 Compilers and Interpreters 29



GPU Architecture

- **GPUs** have to perform the same operation many times over on *large sets* of *data*
  - E.g. *Rotating* the 1000s of *vertices* in a *single* 3D *model* – each point being subject to the *same set of transformations*
- **SIMD**: **Single Instruction, Multiple Data**
  - Use a *single instruction* to carry out the one operation on *many pieces* of *data*

Week 10 Inside the CPU

30

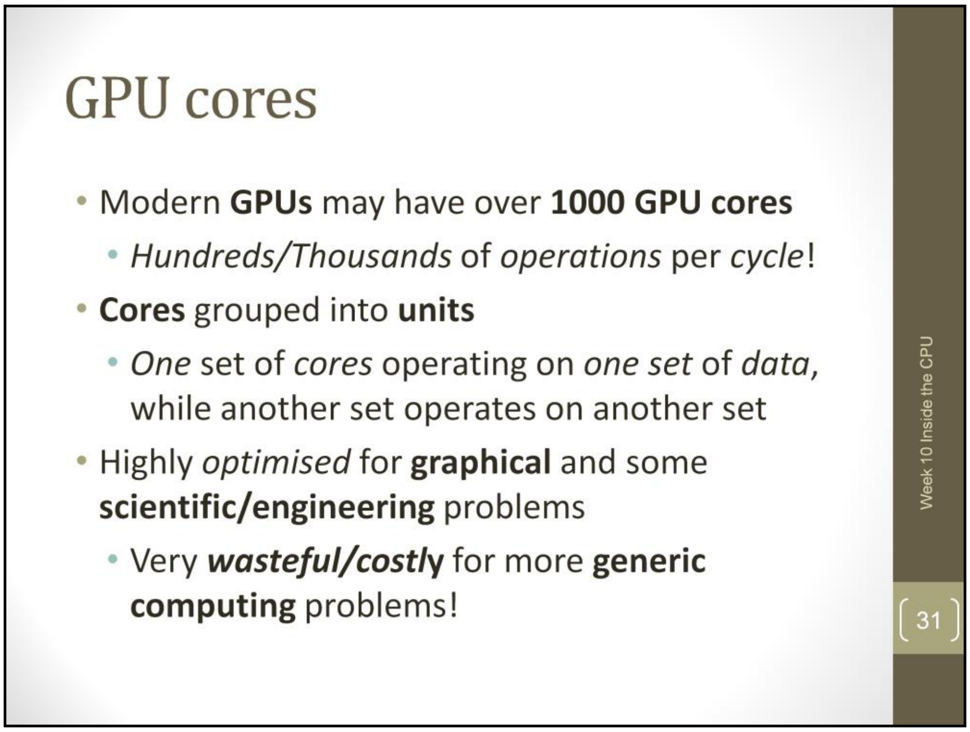
Single instruction, multiple data (SIMD), is a class of parallel computers in Flynn's taxonomy.

It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously.

Thus, such machines exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment.

SIMD is particularly applicable to common tasks like adjusting the contrast in a digital image or adjusting the volume of digital audio.

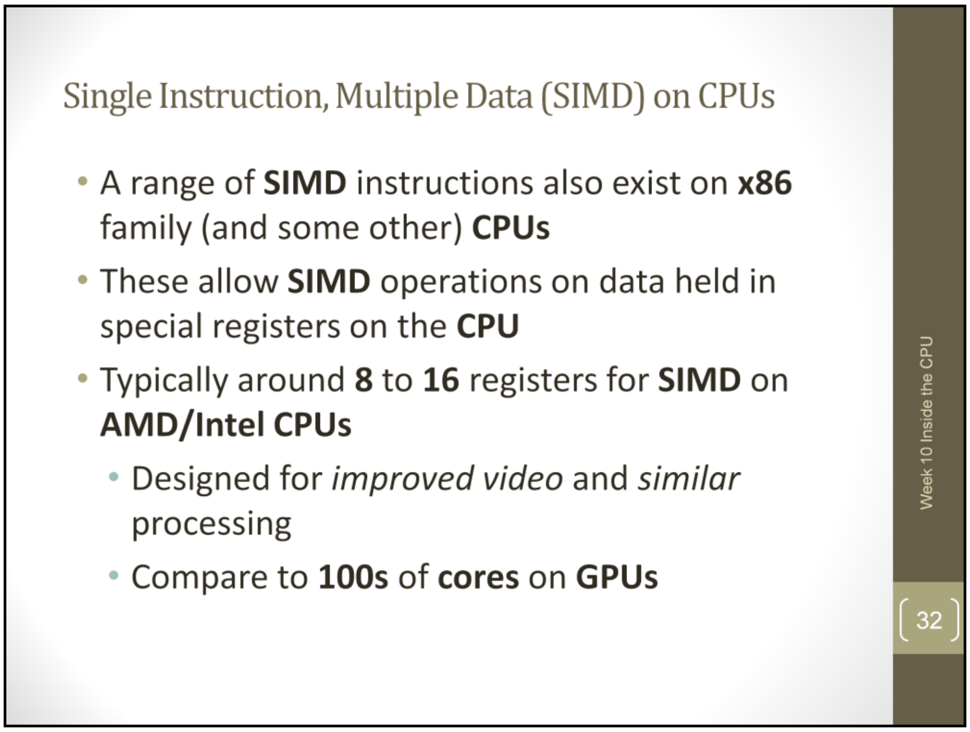
Most modern CPU designs include SIMD instructions in order to improve the performance of multimedia use.



GPU cores

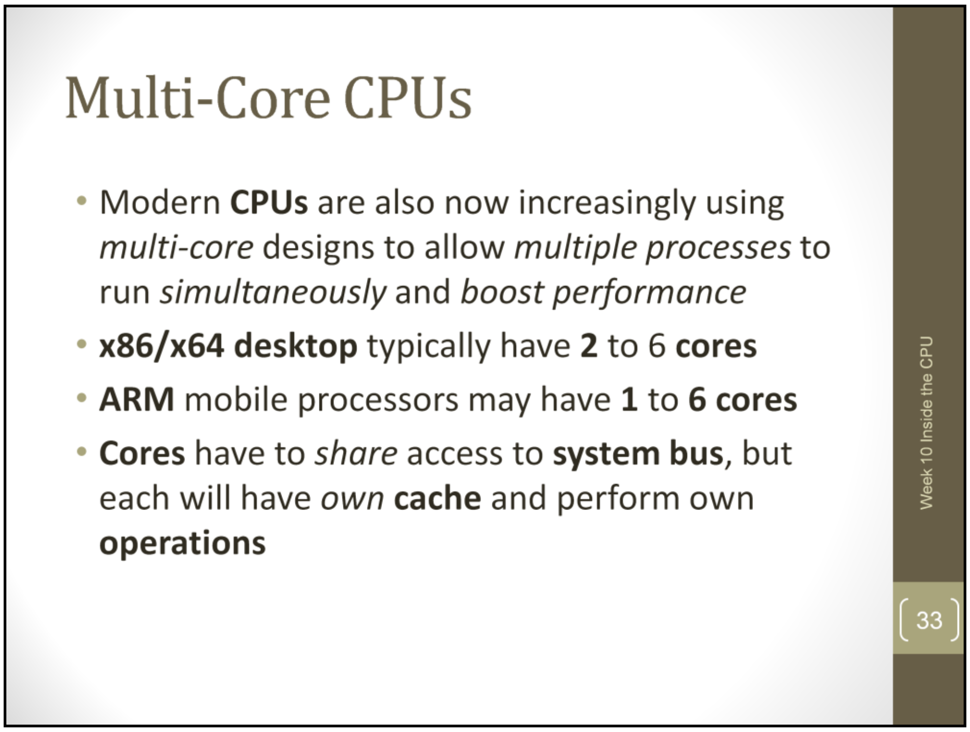
- Modern **GPUs** may have over **1000 GPU cores**
  - *Hundreds/Thousands* of *operations* per *cycle*!
- **Cores** grouped into **units**
  - *One* set of *cores* operating on *one set* of *data*, while another set operates on another set
- Highly *optimised* for **graphical** and some **scientific/engineering** problems
  - Very ***wasteful/costly*** for more **generic computing** problems!

Avoiding going into the jargon here – there are two main GPU product lines from AMD and Nvidia, each with their own very specific sets of jargon which makes it challenging to pull out the more general issues and workings



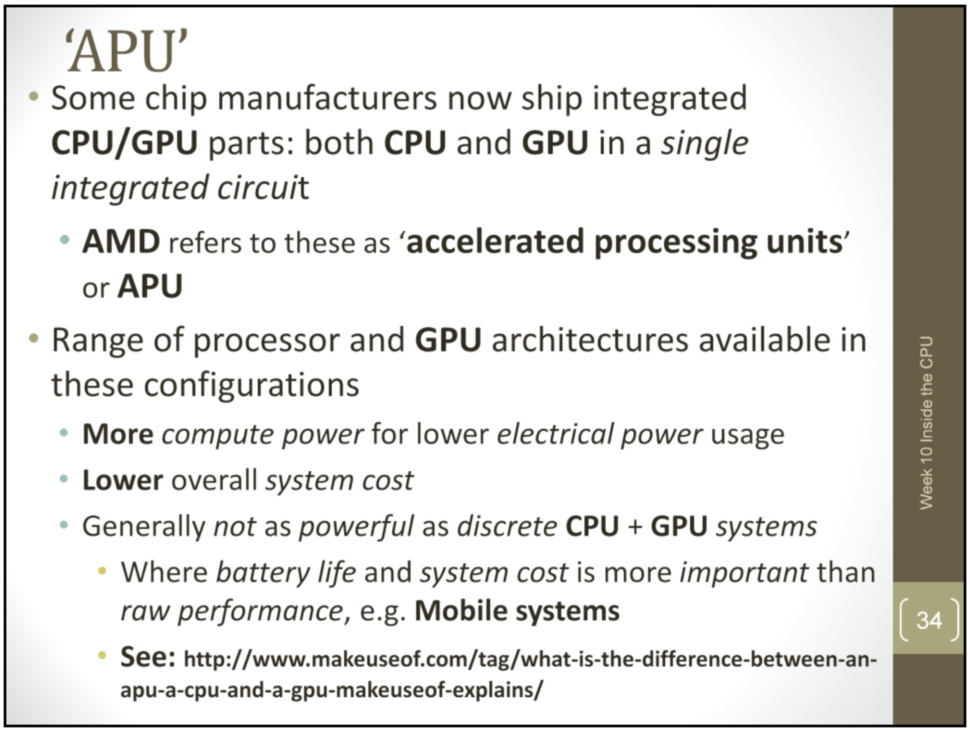
Single Instruction, Multiple Data (SIMD) on CPUs

- A range of **SIMD** instructions also exist on **x86** family (and some other) **CPUs**
- These allow **SIMD** operations on data held in special registers on the **CPU**
- Typically around **8** to **16** registers for **SIMD** on **AMD/Intel CPUs**
  - Designed for *improved video* and *similar* processing
  - Compare to **100s** of **cores** on **GPUs**



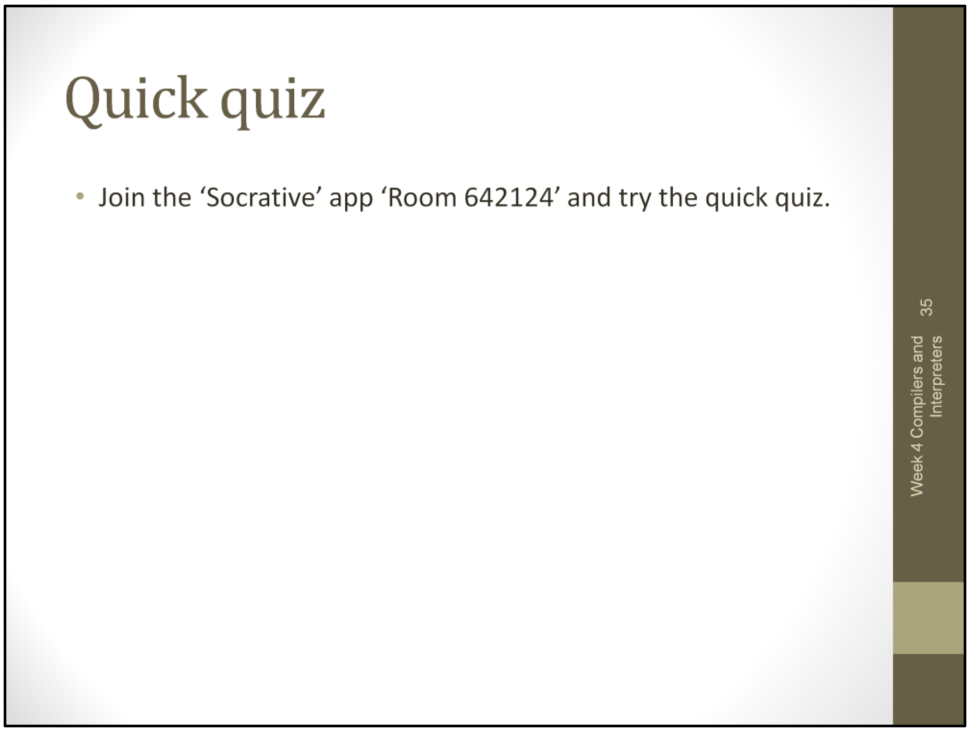
Multi-Core CPUs

- Modern **CPUs** are also now increasingly using *multi-core* designs to allow *multiple processes* to run *simultaneously* and *boost performance*
- **x86/x64 desktop** typically have **2** to 6 **cores**
- **ARM** mobile processors may have **1** to **6 cores**
- **Cores** have to *share* access to **system bus**, but each will have *own* **cache** and perform own **operations**



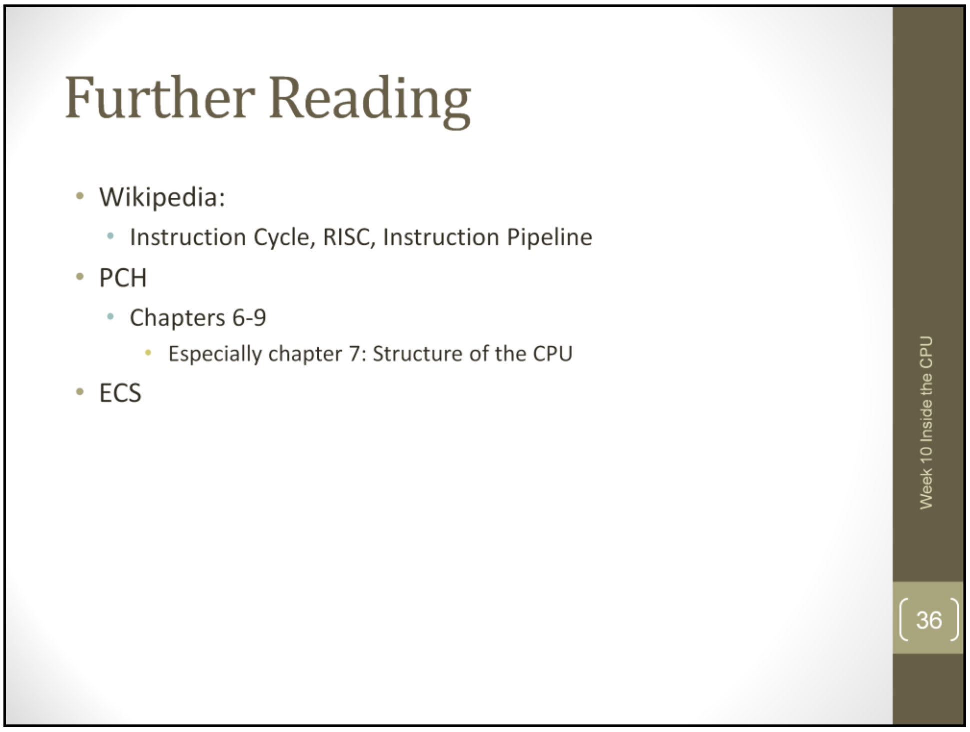
'APU'

- Some chip manufacturers now ship integrated **CPU/GPU** parts: both **CPU** and **GPU** in a *single integrated circui*t
  - **AMD** refers to these as '**accelerated processing units**' or **APU**
- Range of processor and **GPU** architectures available in these configurations
  - **More** *compute power* for lower *electrical power* usage
  - **Lower** overall *system cost*
  - Generally *not* as *powerful* as *discrete* **CPU** + **GPU** *systems*
    - Where *battery life* and *system cost* is more *important* than *raw performance*, e.g. **Mobile systems**
    - **See: http://www.makeuseof.com/tag/what-is-the-difference-between-an-apu-a-cpu-and-a-gpu-makeuseof-explains/**



Quick quiz

- Join the ‘Socrative’ app ‘Room 642124’ and try the quick quiz.

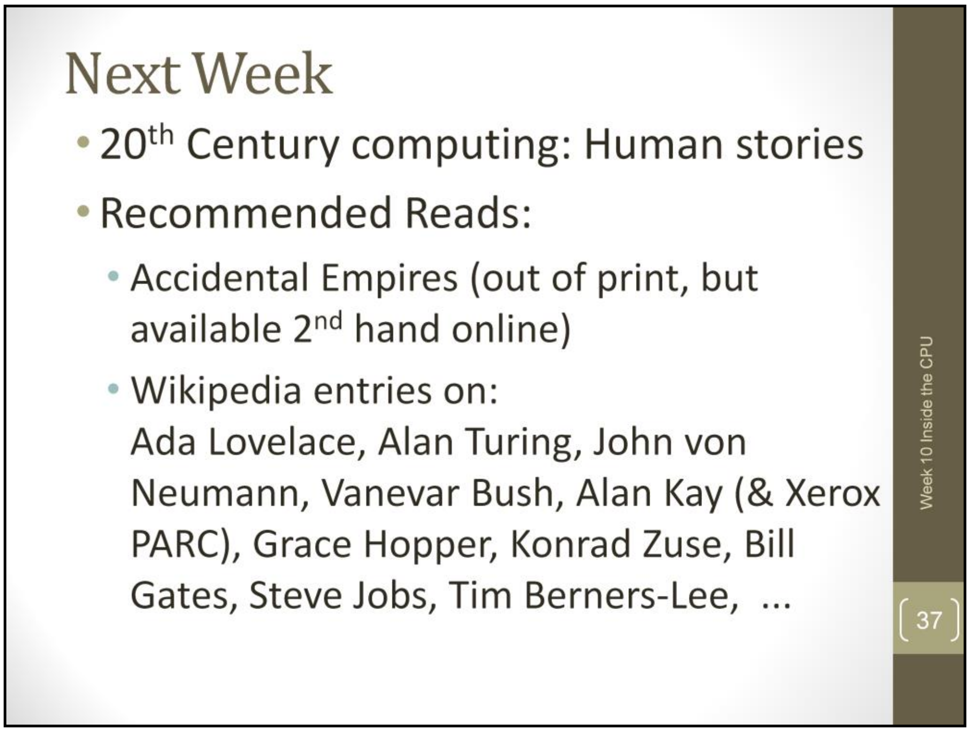


Further Reading

- Wikipedia:
  - Instruction Cycle, RISC, Instruction Pipeline
- PCH
  - Chapters 6-9
    - Especially chapter 7: Structure of the CPU
- ECS

Week 10 Inside the CPU

36


Next Week

- 20th Century computing: Human stories
- Recommended Reads:
  - Accidental Empires (out of print, but available 2nd hand online)
  - Wikipedia entries on:
    Ada Lovelace, Alan Turing, John von Neumann, Vanevar Bush, Alan Kay (& Xerox PARC), Grace Hopper, Konrad Zuse, Bill Gates, Steve Jobs, Tim Berners-Lee, ...

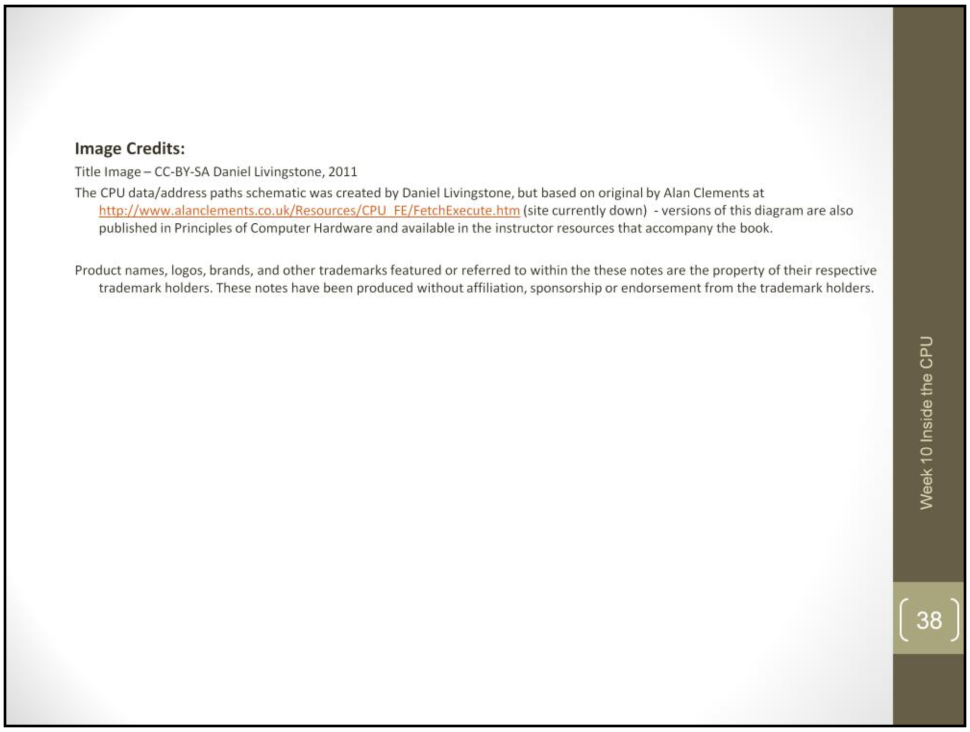


Image Credits:

Title Image – CC-BY-SA Daniel Livingstone, 2011

The CPU data/address paths schematic was created by Daniel Livingstone, but based on original by Alan Clements at http://www.alanclements.co.uk/Resources/CPU_FE/FetchExecute.htm (site currently down) - versions of this diagram are also published in Principles of Computer Hardware and available in the instructor resources that accompany the book.