CSCI 6461 Computer Systems Architecture

Lecture 1

Course Intro, Administration, Basic Concepts

Purpose

- To understand some of the basic concepts and tradeoffs in the design of modern computer architectures.
- This course outline is as follows
  - ISA
  - Machine Language
  - Microarchitectures
  - CPU Caches
  - Virtual Memory Memory System Design
  - Multiprocessor and Multicore Architectures

Computer Architecture

As this is a Computer Science course, we will focus on understanding computer architecture from the perspective of the system software programmer (at an assembly language level).

We are interested in how a machine **executes instructions** .

From a performance perspective, we will explore what is the '**performance impact** ' of s/w and h/w designs; what makes programs run slowly, what h/w features can speed up the execution of programs.

Course Requirements

This course consists of take-home assignments, a class project, and a final exam.

- Take-home assignments = **20%**
- Quizzes (x2) = **10%**
- Mid Term Exam = **35%**
- Final Exam = 35**%**

UNIVERSITET ◦ UNIVERSITY
ADA

Textbooks

- An Introduction to Computer Architecture: RISC ARM32/64. J, Burns and S. Ziyatkhanov, CourseTextBook-2025.pdf (in progress)
- Digital Design and Computer Architecture: ARM ed., S. Harris, D. Harris.
- ARM Assembly Language: Fundamentals and Techniques, 2nd ed., W. Hohl, C. Hinds.

UNIVERSITET ◦ UNIVERSITY ADA

Class Notes

- Lectures will be posted to the Blackboard.
- The final exam will be drawn from the lectures, class notes and assignments.
- The textbook is supplemental to the lectures.
- You should read for absorption and understanding, not for regurgitation.
- Additional material may be provided during the semester.

UNIVERSITET • UNIVERSITY
ADA

Introduction

"Students of computer architecture all too frequently see the microprocessor in the light of the latest high-performance personal computer. For many of them, there is no past—the computer suddenly burst onto the scene as if it had fallen through a time warp. In reality, the computer has a rich and complex history."

A. Clements

What We will Learn?

- how computer systems work, not just the CPU and memory
- the structure of a computer system
- how to analyze their performance
- issues affecting computer systems (caches, pipelines, I/O systems, storage systems)

UNIVERSITET ◦ UNIVERSITY ADA

Why Learn This Stuff?

- You want to become a computer expert
- You want to build high-performance systems: both hardware and software are needed
- You need to make a purchasing decision or offer “expert” advice
- You need to develop new ways of thinking about computation and the optimization of machines and techniques that optimize computation for a given purpose.
- You need to recognize how advances in technology of computation and information communications affect problem solutions and impact society.

UNIVERSITET ◦ UNIVERSITY ADA

Why Learn This Stuff?

- Both Hardware and Software affect system performance!!
- Algorithm determines number of source-level statements
- Language, Compiler, and Architecture determine machine instructions
- Processor and Memory determine how fast instructions are executed
- I/O and network systems determine how fast you can feed data to the computer and get data from it

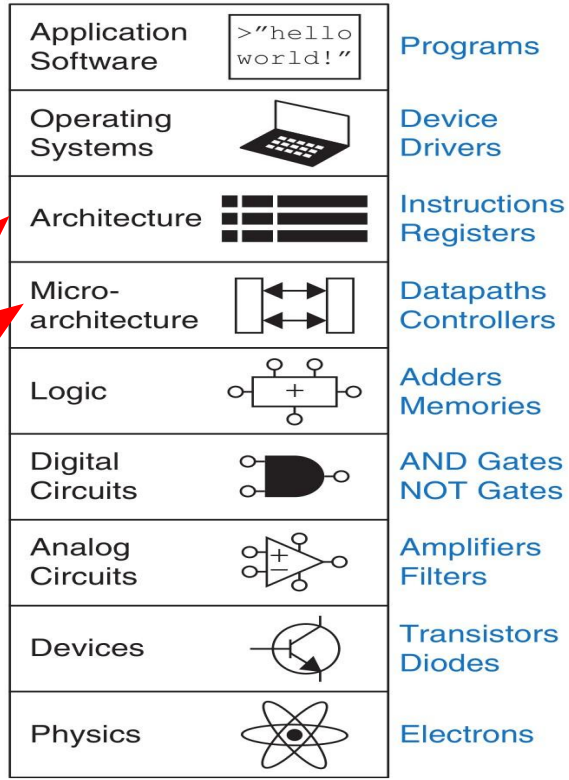
Levels of Abstraction

Levels of Abstraction

We will focus on understanding computer architecture from the perspective of the system software programmer (at an assembly language level).

We are interested in how a machine executes instructions.

We are working here



What is Computer Architecture?

"The attributes of a [computing] system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation."

Amdahl, Blaauw, and Brooks, 1964

Architecture is defined by

Architecture is defined by:

- the instruction set (language)
- operand locations (registers and memory)

Instructions – the words in a computer's language

Instruction set – the computer's vocabulary

- All programs running on a computer use the same instruction set.
- Even complex software applications are eventually compiled into a series of simple instructions such as **add**, **subtract** , and **branch** .

Many different Architectures

MIPS

x86

ARM

AVR

Also, SPARC, PowerPC, ...

UNIVERSITET ° UNIVERSITY ADA

Mobile Architectures

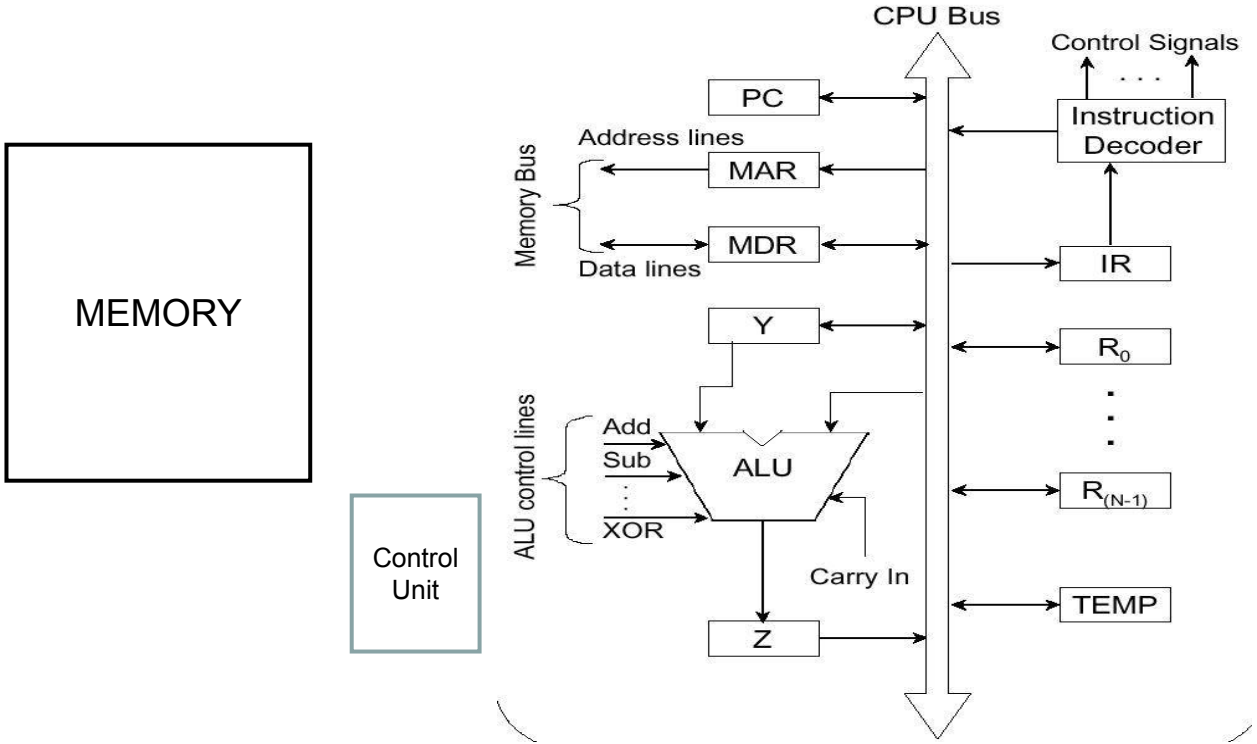
- More than 75% of humans on the planet use products with ARM processors.
- Nearly every cell phone and tablet sold contains one or more ARM processors.
- Of course, alongisde this, the worlds most popular OS kernel is …?

Architecture

Basic Diagrams

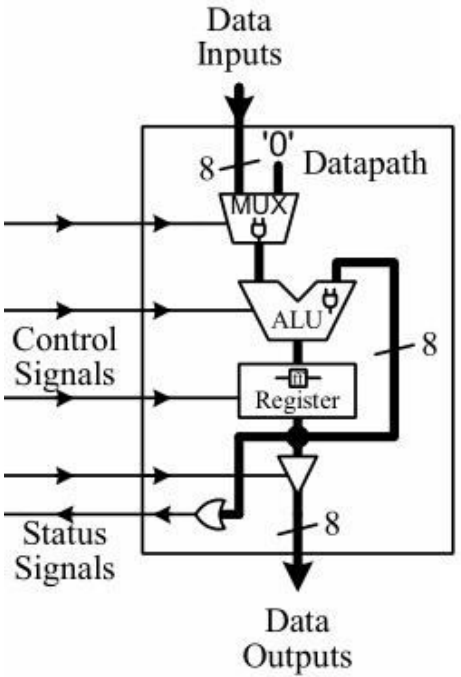
Basic Computer Architecture

VON NEUMANN ARCHITECTURE



Datapath

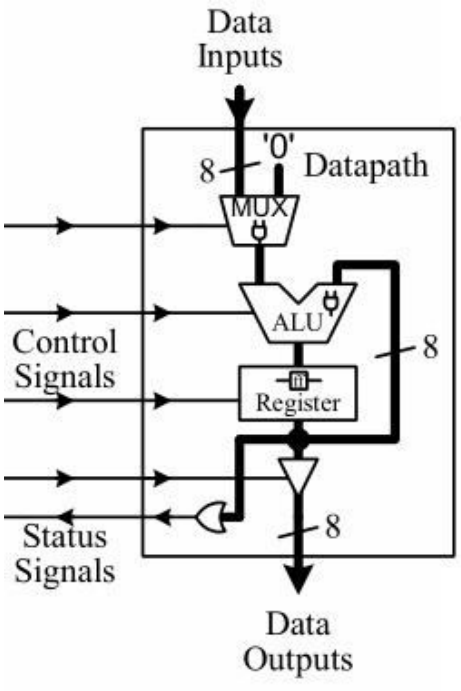
- responsible for the execution of **data** operations performed by the microprocessor, such as
  - the addition of two numbers inside the arithmetic logic unit (ALU).
- includes registers for the temporary storage of data
- Several data signal lines are grouped together to form a bus.
- The width of the bus (the number of data signal lines in the group) is annotated next to the bus line.



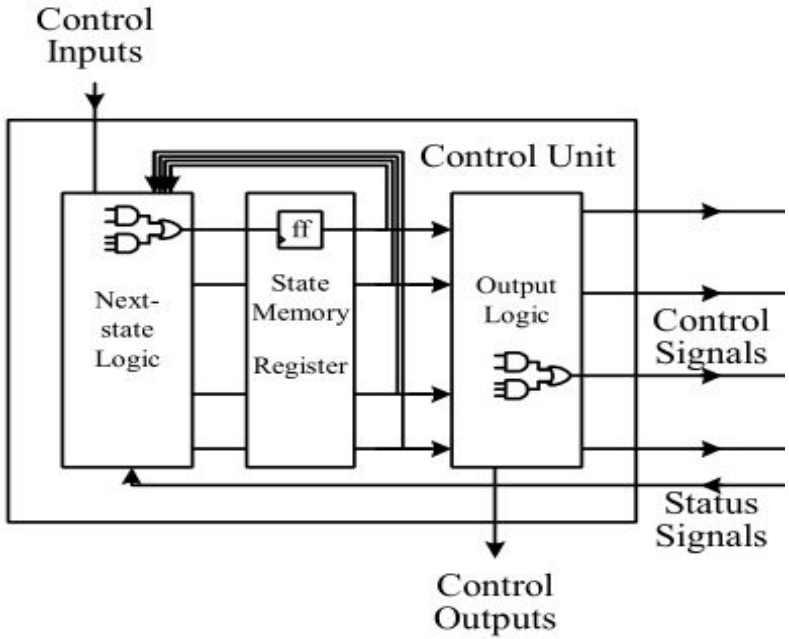
Datapath

Multiplexers (MUXes) are for selecting data from two or more sources to go to one destination.

The tri-state buffer is used to control or isolate the output of the data from the register if required using a high impedance output.



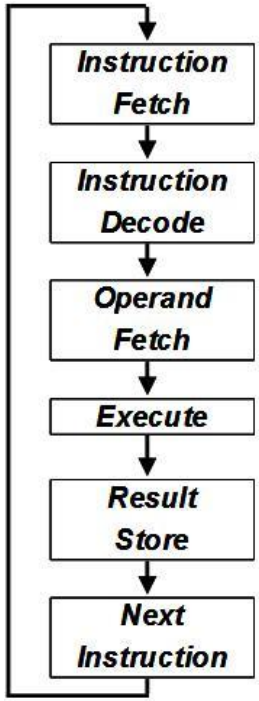
Control Unit



- controls the operations of the datapath, and therefore, the operations of the entire microprocessor.
  - EG, ALU, memory, I/O
- The control unit is a
- finite state machine (**FSM**)
- The output logic of this FSM generates the control signals for controlling the datapath

Basic Computer Architecture

For the basic machine, to execute instructions we will need a few functional units.
The basic instruction execution cycle looks something like this:



Obtain instruction from program storage

Determine required actions and instruction size

Locate and obtain operand data

Compute result value or status

Deposit results in storage for later use

Determine successor instruction

Illustrative Example

Fetch, Decode, Fetch, Execute, Store

Processing an instruction

- What happens when our code executes on some architecture?
- Let us begin by thinking of the typical five stages of instruction processing
  - Fetch, Decode, Fetch, Execute, Store
- We will return to these topics in much more detail in another lecture
- But let's take a look … and think a bit about instruction **costs**

#1 Instruction Fetch: MAR ← PC

- The Program Counter (PC) contains the address of the next instruction to be executed.
- This address should be transferred to the Memory Address Register (MAR).
- This takes **1 cycle** .

MAR ← PC

#1 Instruction Fetch: MBR ← MEM[MAR]

- On the **next cycle** , the Memory Control Unit (MCU) uses the address in the MAR to fetch a word from memory.
- This absolutely fastest fetch occurs in **1 cycle**
  - But it could take 200-300 cycles!.
- The word fetched from memory is placed in the Memory Buffer Register (MBR).

MBR ← MEM [MAR]

#1 Instruction Fetch: IR ← MBR

The contents of the Memory Buffer Register (MBR) are moved to the Instruction Register (IR).

This takes 1 cycle.

Best case: 3 cycles to get a word from memory.

IR ← MBR

#2 Instruction Decode

In 1 cycle process the instruction and use it to set several flags:

- extract the opcode from the IR
- determine the class of opcode:
  - determines the functional unit that will be used to execute the instruction
- set internal flags based on opcode

The above are done in parallel in the decoding logic of the processor.

For example, if the instruction (opcode) is an LDR:

Move the target register from the IR to the Register Select 1

#3 Operand Fetch

Fetch the operand:

- if it is located in memory,
- or extract it from the instruction, if it is immediate,
- or fetch from the stack

- In one cycle, move the first operand address from the IR to the Instruction Address Register **IAR**
  - **IAR** ← **IR**(address field)

#3 Operand Fetch

- If the operand is indexed, in 1 cycle add the contents of the specified index register to the **IAR**
  - **IAR ← IAR + X(index field)**
- In 1 cycle, move the contents of the **IAR** to the **MAR**
  - **MAR ← IAR**
- Again, best case, one cycle fetch the contents of the word in memory
  - specified by the MAR into the MBR.
- **Worst case? - 200-300 cycles**

#4 Execute

Depending on the operation code, execute the operation.

Some examples:

LDR: In one cycle, move the data from the MBR to an Internal Result Register (**IRR**)

STR: Move the contents of the specified register using Register Select 1 to the IRR.

ADDI: In one cycle, using the Register Select 1,

add the contents of the Immediate portion of the IR to the contents of the specified register using an internal adder.

#5 Store

In 1 cycle, move the contents of the IRR to:

- If target = register, use Register Select 1 to store **IRR** contents into the specified register
- If target = memory, such as a **STR**, move contents of **IRR** to **MBR**.
  - On the next cycle, move contents of **MBR** to memory using address in **MAR**.

Q? How did we know where to move the word into memory?

Q? What is an effective address?

Next Instruction

- In most cases, we execute instructions sequentially, so we just increment the PC by the number of bytes/words based on machine addressing scheme
- Q? What is the PC increment given the brief description of the machine?
- But, if it is a branch instruction, then we have to replace the contents of the PC by a different address.

Machine Language

1110 0001 1010 0000 0011 0000 0000 1001

ADA UNIVERSITET ◦ UNIVERSITY

Binary code

- Computer hardware understands only 1's and 0's, so instructions are encoded as binary numbers in a format called machine language.
- The ARM architecture represents each instruction as a **32-bit** or **64-bit** word depending on the ISA.
- Reading machine language is tedious for humans, so we represent the instructions in a symbolic format called assembly language.

Assembly Code

The assembly code is converted into executable machine code by a utility program referred to as an assembler

For example

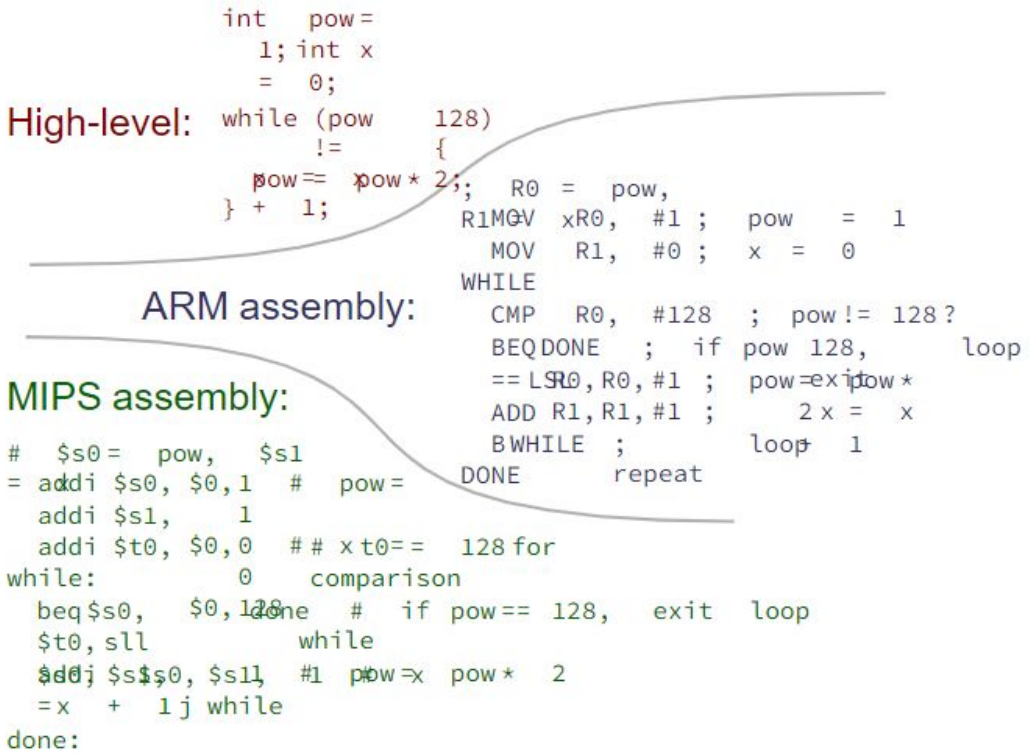
copy the value from “register 9” into “register 3” is written as

MOV R3, R9, and *assembled* into

```
1110 0001 1010 0000 0011 0000 0000 1001
```

C code to Assembly

For engineering tasks where deadlines are crucial, understanding the relationship between high level code and assembly is critical



1:1 Mapping from Assembly

- In general, it is true to say that your assembly language code maps 1:1 with the generated machine code
- This is one reason why developers choose assembly
- This also a reason why C is the choice of performance conscious developers (eg, OS developers)
- By contrast, in Java/C#/Python, we don't really know what kind of binary code is being generated

ARM emulators

For Windows, Linux and Mac OS

ARM Simulators/Emulators

https://cpulator.01xz.net/?sys=arm

- This is a **simulator** and easy to use
  - it runs inside your browser
  - does not require any software install
  - Let's take a look at **cpulator** now
- Qemu - an excellent fully capable **emulator** for many archictecures including ARM/Sparc/x86 etc
- You may wish to develop on an actual ARM platform
  - Raspberry PI
  - AWS/GCP ARM VMs