

# COMP2611: Computer Organization

HONG, LANXUAN

Spring 2025

This is the lecture note for COMP 2611: Computer Organization.

- Digital Design and Computer Architecture: [website](#), [playlist](#)
- CS:APP: [website](#)
- ELEC 3310: [lecture note](#)
- COMP 2611: [website](#)

Last updated on Tuesday 16<sup>th</sup> September, 2025.

# Contents

<b>1 Overview</b>	<b>3</b>
1.1 A Computer Architecture's View . . . . .	3
1.2 A Programmer's Perspective . . . . .	3
<b>2 Digital Design</b>	<b>4</b>
2.1 Transistors . . . . .	4
2.2 Logic Circuit . . . . .	5
2.3 Combinational Logic Circuit . . . . .	5
2.4 Sequential Logic Circuit . . . . .	6
<b>3 Representing and Manipulating Information</b>	<b>8</b>
3.1 Information Storage . . . . .	8
3.2 Integer Representation . . . . .	8
3.3 Floating Point Representation . . . . .	9
3.4 Characters . . . . .	10
<b>4 ISA and Assembly</b>	<b>11</b>
4.1 The Von Neumann Model . . . . .	11
4.2 Data Processing with MIPS . . . . .	12
4.3 Control Flow with MIPS . . . . .	13
4.4 MIPS Registers and memory space . . . . .	13
4.5 MIPS Instructions . . . . .	13
4.6 Instruction Implementation . . . . .	15
4.7 Procedure Call . . . . .	16
4.8 Memory Addressing Mode . . . . .	16
<b>5 Computer Arithmetic</b>	<b>16</b>

# 1 Overview

## 1.1 A Computer Architecture's View

The current direction of computer architecture is: to build **secure, reliable, and safe** architectures; build **energy efficient** architectures for on specific use such as memory-centric, data-centric, and compute-centric; **low-latency** problem and **predictable** problem of architectures; computer architecture for rising areas such as AI/ML, genomics, medicine.

To understand how computers work from the ground up, the **transformation hierarchy** is introduced. The real world **problem** can be solved by procedures for termination, i.e. **algorithm**, which is programmed into **languages** such as Java or C++. These programs will be

The general goal is to build computing system for high-performance, energy efficiency (**the state of the art is the build computing architecture with minimal data movement, e.g. function-in-memory, processor near data**), sustainability, reliability and security (**fundamentally secured not achieved yet**), low-cost or cost efficient, and work for specific goals(**accelerators, AI chips**) which requires algorithm-hardware co-designs.

## 1.2 A Programmer's Perspective

A computer system consists of hardware and system software that work together to run application programs. A source file `hello.c` is represented by **bytes**(8-bit chunks), each bit with a value 0 or 1. A **compilation system** reads source file and translates it into an executable object file `hello`.

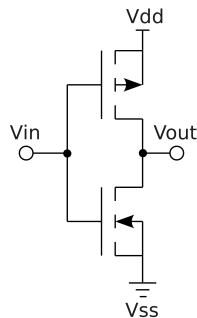
## 2 Digital Design

There are 3 key components in a computer: **computation** component, **communication** component, and **storage/memory** component. These components are built on complex combination of simple logic operations on logic gates. Systems (Microprocessors and FPGAs) for general purpose such as CPU and GPUs, are easier to program (C/C++/Java) but with lower efficiency. Systems for special purpose, such as FPGAs and ASICs are efficient and of high performance, but with limited set of programs (Verilog/VHDL).

### 2.1 Transistors

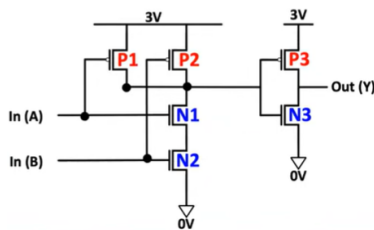
The building block of logic gates is **transistors**. By combining Conductors(Metal), Insulators(Oxide), and Semiconductors, we get a transistor(MOS). Transistors work logically by forming a closed circuit for electrons to move.

A CMOS transistor is constructed by a pMOS transistor(active **low**) and an nMOS transistor(active **high**). The source of pMOS connects to high voltage (to pass 1) and the drain of nMOS connects to low voltage (to pass 0). Exactly one network should be on while the other one is off so that the output is either 1 or 0, otherwise would lead to **short circuit** (both on) or **floating output** (both off).



#### Example 2.1

An inverter (NOT Gate) from CMOS circuit, where pMOS connecting to 3V pulls the output out, while nMOS connecting to 0V pulls the output down.



#### Example 2.2

A more complex CMOS design for AND Gate from NAND Gate and NOT Gate. P1 and P2 are connected in **series** while N1 and N2 are connected in **parallel**. In this case 0 is achieved when both inputs are 1 (active both Ns).

## 2.2 Logic Circuit

The logic circuit abstraction consists of four components: **input**, **output**, **functional specific**, and **timing specific**. Analog signals are converted to digital bits for inputs and outputs in digital circuits. Binary representation supports boolean arithmetic, implemented by logic gates. Complex functions are realized through combinations of logic gates in **combinational logic circuits**. Additionally, **memory** for **sequential logic circuits** can also be constructed using logic gates, representing current output based on current and previous inputs.

## 2.3 Combinational Logic Circuit

Functional specification creates a unique mapping from current input values to current output values. A truth table shows the logic function from input-output mapping. **SOP** (sum of products) and **POS** (product of sums) can be applied on minterms and maxterms of truth table for standard function representation.

- **adder** is a simple device implementing boolean addition.
- **multiplexer/MUX** is a function selector which select one of the  $n$  inputs to connect its value to the output. It takes  $2^n$  data inputs,  $n$  select inputs, and gives a single output. The select inputs forms a look-up-table(LUT) for pattern selection, which is implemented widely in programmable devices such as PLA and FPGA.

An example of MUX application is Arithmetic Logic Unit(ALU) which combines various functions into a single unit.

- **decoder** is an input pattern detector and a minterm generator. It takes  $n$  inputs have  $2^n$  outputs where the only output of value 1 indicates input pattern. This is often applied to access encoded memory storage or decide action.
- **Programmable Logic Array/PLA** is an array of AND gates followed by an array of OR gates with inverters. **(all logic function can be expressed in SOP or POS, which is 2-level-function)** We can program the connection from AND gate outputs to OR gate inputs from the look-up-table.
- **tri-state buffer** is an enable gate of different signals onto a wire. A 0 in enable input results in floating signal  $z$  in output. This can be applied to connect

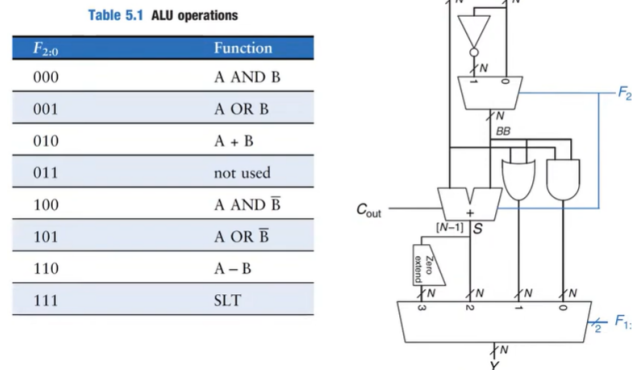


Figure 1: An ALU Example

CPU and memory on a shared output bus.

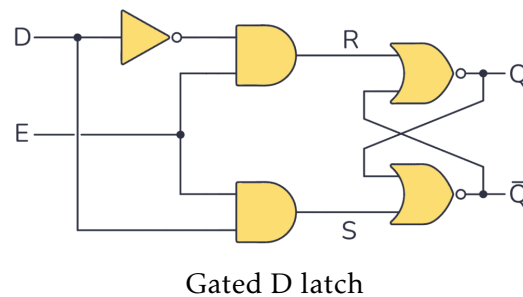
Logic simplification is important for increasing efficiency and saving energy by minimizing the number of logic gates. The primary focus is to eliminate changing variables and identify the deciding variable. The **K-map** method (**Gray code**) is a valuable tool for achieving this simplification.

## 2.4 Sequential Logic Circuit

Sequential logic circuits are circuits that can store information and produce output depending on current and past input values. There are different **storage** elements and devices with different cost of transistors.

- **S-R(Set-Reset) latch** can be implemented by cross-coupled NAND gates or cross-coupled NOR gates. Data is stored at Q and Q' and control inputs S and R provides a control mechanism. The following truth table of cross-coupled NOR gate implementation shows a forbidden state, which violates  $Q \neq Q'$ . This would also cause **race condition** which lead to undetermined state.

S-R Latch				
S	R	Q	Q'	State
0	0	Q	Q'	Hold
0	1	0	1	Reset
1	0	1	0	Set
1	1	0	0	Forbidden!



- **gated d-latch** guarantee correct operation of an S-R latch by controlling only 1 of the SR input be 1. A write enable gate(WE) allows hold state, which is often controlled by a clock signal making latch level-sensitive.
- **register** is a connected and integrated series of sequential devices(latches or flip-flops) and stores **memory**.

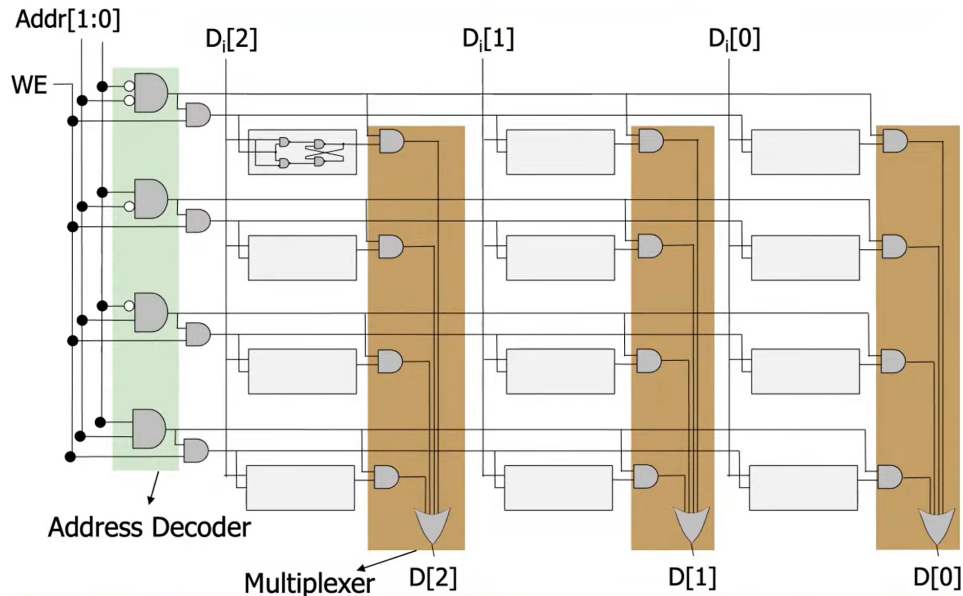


Figure 2: Memory Array Construction

**Note** ⚠ Level triggered latches are hard to synchronize and sensitive to glitches. To solve the transparency of latches and build state register, master-slave flip-flops are structured using 2 latches (**a protector and a activator**) connected in series which makes it edge triggered.

- **d-flip-flop** samples  $D$  on the rising/falling edge of CLK. Register based on flip-flops are synchronized but more expensive.
- **finite state machine(FSM)** tells the state register(current state of the system), next state logic(possible next state), and output logic(interface with the out-side world) of a digital design. 2 types of FSM include **mealy machine** and **moore machine**.

## 3 Representing and Manipulating Information

### 3.1 Information Storage

A machine level program views memory as a very large array of bytes, referred to as virtual memory. Every byte of memory is identified by its address, consisting of the virtual address space. Compiler and run-time system will partition this memory space into more manageable units to store the different **program objects**(i.e. program data, instructions, and control information).

For program objects that span multiple bytes, 2 conventions must be established: what the address of the object will be, and how we will order the bytes in memory. In a virtually all machines, a multi-byte object is stored as a contiguous sequence of bytes, with the address of the object given by the smallest address of the bytes used.

### 3.2 Integer Representation

**Bits** stored in flip-flops work together as **bit sequences**, e.g. 32-bit, 64-bit. We can represent **integers** using bit sequences.

#### 1. binary encoding

$$B2U_{\omega}(\vec{x}) = \sum_{i=1}^{\omega-1} x_i \cdot 2^i$$

$$\text{e.g.: } B2U_4(0110) = 0 \times 2^3 + 1 \times 2^2 + 1 \times 2 + 0 \times 2^0 = 6$$

#### 2. 2's complement encoding

$$B2T_{\omega}(\vec{x}) = -x_{\omega-1} 2^{\omega-1} + \sum_{i=0}^{\omega-2} x_i \cdot 2^i$$

where the most significant bit  $x_{\omega-1}$  is considered as **sign bit** with weight  $2^{\omega-1}$ . The representation is negative if sign bit is 1, otherwise positive. e.g.  $B2T_4(1011) = -1 \times 2^3 + 3 = -5$ . For a 4-bit sequence, its representation range is  $[-8, 7]$ . To get the 2's complement representation from integer, first obtain the binary representation of the positive value, then flip all bits and add 1. e.g.  $T2B(-5) = -(0101) = 1010 + 1 = 1011$



One common operation is to convert between integers having different word sizes while retaining the same numeric value. Converting from a large data type to a smaller one sometimes causes **underflow** or **overflow**. However, it is always allowed to convert from a smaller to a larger data type. For unsigned integers, we apply **zero extension** by adding 0 to more significant bits. For 2's complement integers, we apply **sign extension**, adding copies of the sign bit to the representation.

Integer arithmetic see [ELEC 3310 lecture note page 5](#).

### 3.3 Floating Point Representation

Consider the normalized scientific notation  $a \times 10^a$  where  $a \in [1, 10)$ , which gives a unique representation to any decimal number. This can be applied to binary representation, where the first digit is always 1. **⚠ important, no need to memorize**

**Example 3.1** Normalized scientific notation of binary representation

$$-5.75 = -101.11 = -1.0111 \times 2^2$$

Note that a fractional number can be represented by negative binary powers.

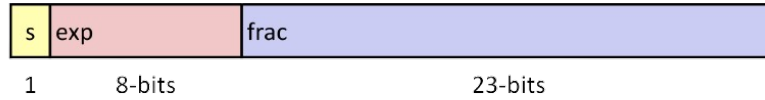
$$b = \sum_{i=n}^m 2^i \times b_i$$

To pack this representation into computer, we need to store the **signs**, **significand**  $M$ , and **exponent**  $E$  ( $E = e + Bias$ ). The **floating-point representation** encodes rational numbers of the form  $V = (-1)^s \times (1 + M) \times 2^{E-Bias}$  (IEEE standard format). **⚠ 1+M instead of M: since the first digit is always 1, there is no need to specify**

There are 2 precision standards: single precision and double precision (more precise and larger range). The general philosophy for constructing this representation is to maximize the usage of digits.

1. Explicit 1: save 1 digit
2. Bias: easier for comparison
3. **de-normalized case**: to represent 0
4. **special case**: for infinitely or non-numbers

Single precision: 32 bits



Double precision: 64 bits

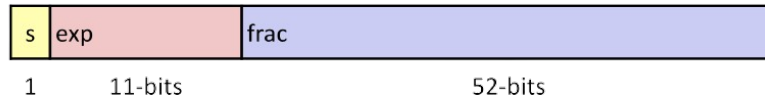


Figure 3: Floating Point Format

type	Exponent	Significand	Usage	Range
Normalized	1-254	Implicit 1	for normal numbers	$\pm 1.F \times 2^{E-127}$
Denormalized	all 0s	all 0 for 0	for small numbers	$\pm 0.F \times 2^{-126}$
Special Cases	all 1s(255)	all 0 for $\infty$	not all 0 for NaN	

Table 1: Single Precision Floating Point Representation

### 3.4 Characters

Characters are unsigned bytes(8-bit) following the [ASCII](#) standard(a look-up-table).

## 4 ISA and Assembly

Our program is a set of instructions, where each instruction specifies a well-defined piece of work for the computer to carry out. Each instruction, the smallest piece of specified work, can be written in **assembly language** and translate to **machine languages** by the assembler. The **instruction set architecture**(ISA) specifies all possible instructions that a computer is designed to be carry out.

**Note** ⚠ An ISA may have different ways of implementation according to the design of micro-architecture, but it defines a rather higher level idea of how hardware is allocated and processes, allowing different software written in such ISA being implemented on different machines.

### 4.1 The Von Neumann Model

The **von Neumann model** is an execution model for processing computer programs. It includes **memory**, **processing unit**, **input**, **output**, and **control unit**. The (Microprocessor without Interlocked Pipeline Stages)MIPS<sup>1</sup> ISA is an example of con Neumann model.

Memory stores programs and data, represented by bits. There are 2 common practice to order the address, namely big Endian(LSB is higher byte address) and little Endian(MSB is higher byte address). There are 2 ways of accessing memory: reading or loading from a memory location, and writing or storing data to a memory location. To access memory, we need to first load the address using **MAR**, and store/read the memory from/to **MDR**.


The processing unit lies in the core of a computer. It consist of many function unit and processes quantities that are referred to as **words**.<sup>2</sup> For fast computation, the computer provides a temporary storage of words for processing.<sup>3</sup>

The control unit conducts the **step-by-step process** of executing every instruction in a program in sequence. It keeps track of which instruction being processed via **instruction register**, and which instruction to process next via **instruction pointer**.

<sup>1</sup>In MIPS, the address space is  $2^{32}$  storing by a 32-bit address and is byte-addressable.

<sup>2</sup>Word length in MIPS is 32 bit

<sup>3</sup>MIPS has 32 general purpose registers each storing 32 bits(a word). GPRs can be accessed through 5-bit register id from  $R_0$  to  $R_{31}$

**Note**  von Neumann model has 2 key properties: stored program(unified memory between instruction and data, interpretation of a stored value depends on the control signal) and sequential instruction processing.

## 4.2 Data Processing with MIPS


Each instruction has an **opcode** with a fixed number of **operands** in MIPS<sup>4</sup>, while Intel architecture supports a variable number of operands. For example, we can do arithmetic operations using MIPS.

```
add a,b,c # a = b + c
# manipulating registers
f = (g+h)-(i+j)
add $t0,$s1,$s2 # g+h
add $t1,$s3,$s4 # i+j
sub $s0,$t0,$t1 # f
```

With operate instructions, the computer execute arithmetic computation in the ALU, and access the operands from memory, i.e. load them from memory to registers. **MIPS data processing instructions always operate on registers!** For the 32 general purpose registers in MIPS, there are saved registers \$s0,...\$s7(usually used to correspond to variables in a high level program), temporary registers \$t0,...\$t7 (additional registers for variables and temporary data), and \$0(read-only, holding a constant value 0).

We can define infinite variables in a program, but only with a fixed number of register, which requires a smart load and store operations.

```
# $s0 is the address of A
# address operations: offset(base-register)
lw $t0 32($s0) # load A[8] from memory to register
sw $t0 48($s0) # store from register to memory A[12]
```

**Note**  property of von Neumann model: registers can hold address as well!

<sup>4</sup>make the hardware simple and fast

### 4.3 Control Flow with MIPS

```
# instruction branching
beq $s0 $s1 L1 # if s0 and s1 are equal then branch to L1
add $s0 $s0 $s1
L1: sub $s0 $s0 $s1
```

Each block is constructed starting from a label and ending with a branching.

```
/* C++
if(a<b){a*=1}
*/
# MIPS
slt $t0 $s0 $s1 # $t0 is 1 if a<b
beq $t0 1 If # if a<b then go to If
If: sll $s0 $s0 2 # a=a*4
j Exit # jump to exit
```

MIPS instructions are encoded as a 32-bit instruction words

### 4.4 MIPS Registers and memory space

MIPS data processing operates on **registers**. MIPS has 32(numbered 0 to 31) general purpose registers, each is of 32 bits in size. This allows a fast processing of data processing by loading the data from the main memory to registers and store them back after processing. Each memory location storing 8-bit bytes(a memory cell) is indexed by a 16-bit **address**, allowing  $2^{16}$  bytes total memory capacity.

To specify the address of the memory location of an array element, **base address** and **offset** are used. For example, for an word array(each word is of 4 bytes) A with starting position \$s0,  $A[8] = 32(\$s0)$  #  $A[8] = \$s0 + 32$  bytes.

There are 2 standard, **big Indian**(tail at bigger address) and **Small Indian**(tail at smaller address), to store words(4-byte) into the memory.

### 4.5 MIPS Instructions

1. arithmetic operation:

```
add $t0, $t1, $t2 # t0 = t1 + t2
addi $t0, $t1, -1 # immediate operation with a constant or constant 0($0)
```

**Note**  **Simplicity favors regularity:** each MIPS instruction has a fixed number of operands.

## 2. memory operands:

```
lw $t0, $s0 # load the data at address $s0 to $t0
sw $t0, (4)$s0 # save the data
```

## 3. logical operation: bit-wise

```
and $t0, $t1, $t2 # bit-by-bit operation
sll $t0, $s0, 4# shift left logical, multiply the result by 16
srl $t1, $s1, 1# shift right logical, divide the result by 2
```

## 4. control flow:

```
beq reg1, reg2, Label1 # if the value in reg1 and reg2 are equal then branch
bne reg1, reg2, Label2
j Exit # unconditional jump to avoid re-execution
Label1:
Label2:
Exit:
```

Loops in MIPS: use simple control flow operations.

```
slt reg1, reg2, reg3 # reg1 is 1 if reg2 < reg3, otherwise 0
bne reg1, $0, Label1 # jump to label1 if reg2 < reg3
```

Jump register instruction jr

```

la $t0, Label
addi $t0, $t0, 8 #
jr $t0 # jump to instruction 3
Label:
instruction 1
instruction 2
instruction 3 # address: instruction1 + 8

```

## 4.6 Instruction Implementation

The instructions are translated into binary machine code. MIPS instructions are encoded in 32-bit and broken up into a number of fields. Since each MIPS instruction have a fixed number of operands(each operand requires 5-bit representation for the 32-bit register), all MIPS instruction with same number of operands could have the same format of storing instructions. There are 3 standard **instruction format**: R-format for register, I-format for Immediate operations or load and store, and J-format for jump.

R (Register) Format:

Opcode	Rs	Rt	Rd	Shamt	Funct
(6)	(5)	(5)	(5)	(5)	(6)

Most arithmetic and logic instructions (except 'immediate')

I (Immediate) Format:

Opcode	Rs	Rt	16-bit Immediate value
(6)	(5)	(5)	(16)

Data Transfer, Immediate, and Cond. Branch instructions

J (Jump) Format:

Opcode	26-bit word address
(6)	(26)

Unconditional Jump instructions

Figure 4: MIPS Instruction Formats

#### **4.7 Procedure Call**

#### **4.8 Memory Addressing Mode**

### **5 Computer Arithmetic**