# [Projects | nand2tetris](https://www.nand2tetris.org/course)

[GlenAMacdonald/Nand2TetrisPt1: https://www.coursera.org/learn/build-a-computer/home/welcome (github.com)](https://github.com/GlenAMacdonald/Nand2TetrisPt1)

# CPU Design -

# Layout:

# Diagram Description automatically generated

## Mux16-ARegister – Control Switch:

inputA[16]

* Instruction

inputB[16]

* ALUOutput (M)

C[1] =

* d1 – the d1 bit of the destination instruction indicates that the ARegister should store the results of the computation (Figure 4.4)
  + NOTE: There might need to be some way of indicating that the computation has been completed and needs to be stored – initially assume there no synchronization required.
* Otherwise let the input be the instruction.

Output[16]

* ARegister

## ARegister:

Input[16]

* Mux16-ARegister

C[1] =

* d1 - the d1 bit of the destination instruction indicates that the ARegister should store the results of the computation (Figure 4.4)
  + NOTE: There might need to be some way of indicating that the computation has been completed and needs to be stored – initially assume there no synchronization required.
* This also needs to load the new instruction, probably at the same time the program counter is incremented.

Output[16]

* Mux16ALU InputA
* AddressM (CPU output)
* ProgramCounter Input

## DRegister:

Input[16]

* ALUOutput (M)

C[1] =

* d2 - the d2 bit of the destination instruction indicates that the DRegister should store the results of the computation (Figure 4.4)

output[16]

* ALUInputX

## writeM:

C[1] =

* d3 - the d3 bit of the destination instruction indicates Memory[A] (memory register addressed by A) should store the results of the computation (Figure 4.4)

## Mux16ALU:

inputA[16]

* ARegisterOutput

inputB[16]

* M

C[1] =

* i - In the case of a C-instruction, the single a-bit determines whether the ALU will operate on the A register input or on the M input,
  + i == 0 for an A-instruction
  + i == 1 for a C-Instruction
* a ($[12]) (5.3.1)
  + When a == 0 – select A Register
  + When a == 1 – select M as the input

output[16]

* ALU input ‘Y’. – via comparison between figure 2.6 and figure 4.3

## Program Counter

input

## ALU

inputX[16]

* DRegisterOutput

inputY[16]

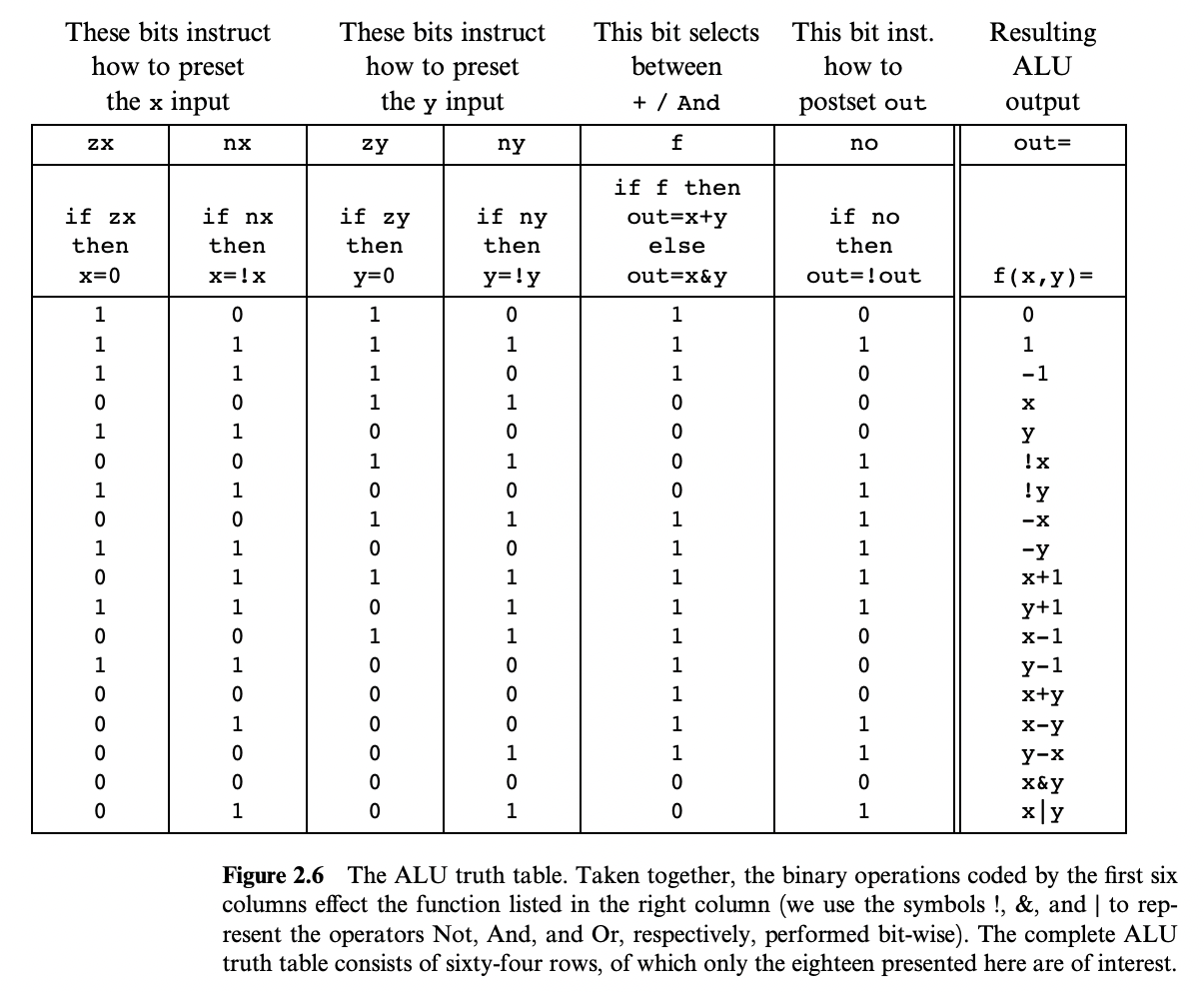
* Mux16ALUOutput

inputC’s[6]

* Instruction bits c1-c6

OutputC’s

* Zr[1] – computation resulted in a zero
* Ng[1] – computation resulted in it being less than zero
* Output[16]



# Section Overview:

* Section **5.2** gives its exact hardware specification
* Section **5.3** describes how the Hack platform can be implemented from previously built chips, in particular the ALU built in project 2 and the registers and memory systems built in project 3
* Section **5.4** compares the Hack machine with industrial-strength computers, and emphasizes the critical role that optimization plays in the latter.
* Section **5.5** gives an overview of the computer construction project.

## Component Description:

* Data Registers (D): (5.1) - a CPU internal register used for temporary storage
  + If we want to compute (a-b).c it can be used to store the result (a-b).
  + (5.2.1) - The D-register is used solely for storing data values
* Address Registers (A): (5.1) - a CPU internal register used to store an address. Can also be used to as a data register if needed.
  + Set to 17 by the commands: ‘@17, D=A’. Note that @17 sets the next memory location to 17 and then loads that value into the D Register.
  + (5.2.1) the A-register serves three different purposes,
    - storing a data value (just like the D-register),
    - pointing at an address in the instruction memory, or
    - pointing at an address in the data memory.

## Inputs (5.2.4)

### inM[16]

* + From Data Memory (16-bit)
  + Value of Mem[A], the instructions M input

### instruction[16]

* + From Instruction Memory (16-bit)
  + Instruction to execute

### reset[1]

* + reset == 1 continues executing, reset == 0 restarts the current program

## Outputs (5.2.4)

### outM[16]

* + To Data Memory
  + Value to write to Mem[addressM], the instructions ‘M’ output.

### address[15]

* + To Data Memory
  + In which address to write

### writeM[1]

* + Write To Data Memory?

### PC[15]

* + To Instruction Memory
  + Address of next instruction

# Instructions “ixxaccccccdddjjj”

(5.2.1)

## i. i == 0 for an A-instruction, i == 1 for a C-Instruction

### A-Instruction:

The instruction is treated as a 16-bit binary value which is loaded into the A register

### C- Instruction

The instruction is treated as a sequence of control bits that determine which function the ALU should computer, and in which registers the computed value should be stored.

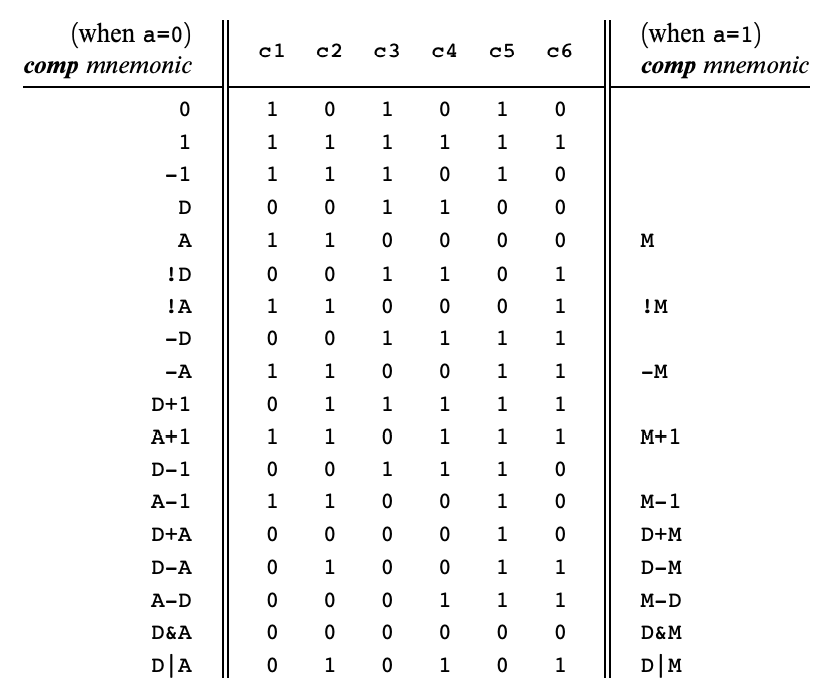
## xx. - These aren’t used

## a[1]

* + (5.3.1) ALU operates on the A register input or M input
  + a == 0, select A register (assumed
  + a == 1, select M input

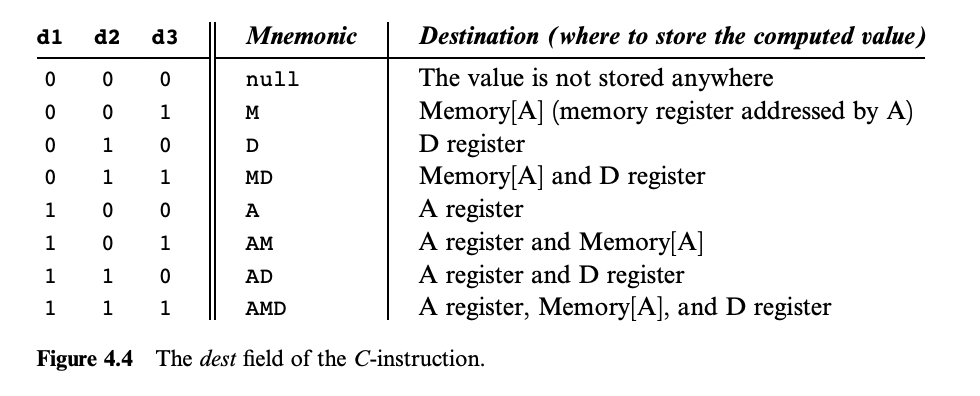
### c[6]

* + (5.3.1) Control bits – the commands specified for the ALU
  + (4.2.3) – Figure 4.3



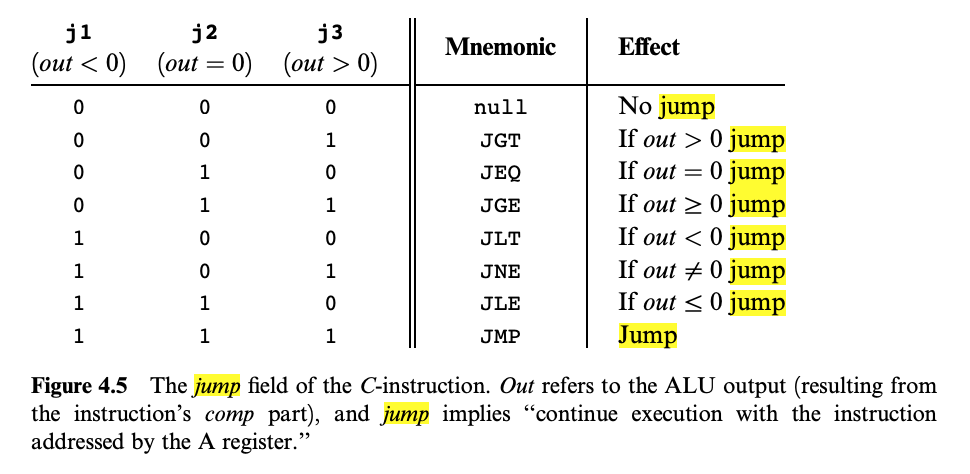
### d[3]

* + (5.3.1) Destination – Which Memory Register should accept the ALU’s output M
  + (4.2.3) – Figure 4.4



### j[3]

* + (5.3.1) Jump – Which Instruction Register should be loaded on a Jump command.
  + (4.2.3)



## Jump Operation (5.3.1):

But what if the instruction dictates to effect a “jump n” operation, where n is the address of an instruction located anywhere in the program? According to the Hack language specification, a “jump n” operation is realized using a sequence of two instructions. First, we issue the A-instruction @n, which sets the A register to n; next, we issue a C-instruction that includes a jump directive. According to the language specification, execution always branches to the instruction that the A register points at. Thus, when implementing the CPU, one of our challenges is to come up with a logic gate architecture that realizes the following behavior: if jump then PC = A else PC++. The value of the Boolean expression jump depends on the instruction’s j-bits and on the ALU output.