

Let's make a CPU

Made for IGCSE at Musashi International School Tokyo

What you'll need

- Access to a computer with Sebastian Lague's *Digital Logic Sim.*

What you'll need

- Access to a computer with Sebastian Lague's *Digital Logic Sim.*
- Knowledge of binary representation for unsigned and signed numbers using two's complement.

What you'll need

- Access to a computer with Sebastian Lague's *Digital Logic Sim.*
- Knowledge of binary representation for unsigned and signed numbers using two's complement.
- Basic knowledge of logic gates and truth tables.

What you'll need

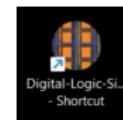
- Access to a computer with Sebastian Lague's *Digital Logic Sim*.
- Knowledge of binary representation for unsigned and signed numbers using two's complement.
- Basic knowledge of logic gates and truth tables.
- Motivation to tackle a series of small and manageable puzzles that will lead you to a fully working CPU of your own!

Outline

- 1 Project Setup
- 2 Basic Logic gates
- 3 Arithmetic
- 4 Memory 1
- 5 Control Unit part 1
- 6 Buses
- 7 Memory 2
- 8 Building up the CPU
- 9 Control Unit part 2
- 10 Testing the CPU and writing programs

Project Setup

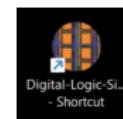
Open Sebastian Lague's *Digital Logic Sim.*



Digital-Logic-Si...
- Shortcut

Project Setup

Open Sebastian Lague's *Digital Logic Sim.*



Digital-Logic-Sim...
- Shortcut

Click on new project, name it 8-bit CPU.

NEW PROJECT

Outline

- 1 Project Setup
- 2 Basic Logic gates
- 3 Arithmetic
- 4 Memory 1
- 5 Control Unit part 1
- 6 Buses
- 7 Memory 2
- 8 Building up the CPU
- 9 Control Unit part 2
- 10 Testing the CPU and writing programs

How to use *Digital Logic Sim*

In *Digital Logic Sim*, we work with chips. We are given a few base chips, and we are going to make more of them. Let's start by making a chip for the NOT gate.

- Click on the bottom bar to add elements to your cursor. Let's grab an IN-1.

How to use *Digital Logic Sim*

In *Digital Logic Sim*, we work with chips. We are given a few base chips, and we are going to make more of them. Let's start by making a chip for the NOT gate.

- Click on the bottom bar to add elements to your cursor. Let's grab an IN-1.
- Click on the background to drop elements into the chip. Let's put our IN-1 on the left.

How to use *Digital Logic Sim*

In *Digital Logic Sim*, we work with chips. We are given a few base chips, and we are going to make more of them. Let's start by making a chip for the NOT gate.

- Click on the bottom bar to add elements to your cursor. Let's grab an IN-1.
- Click on the background to drop elements into the chip. Let's put our IN-1 on the left.
- Repeat for a NAND gate and an OUT-1.

Our first gate, the NOT gate

Wire it up to create your first handmade gate, the NOT gate. It should respect the following truth table:

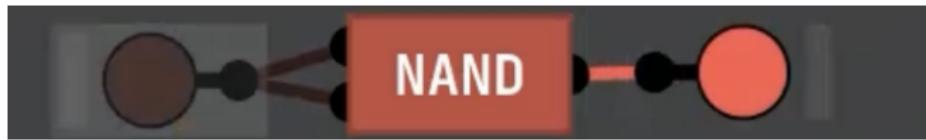
A	NOT A
0	1
1	0

You can check your chip is correct by clicking the In-1 element to toggle its value between 0 and 1, and looking at the value of your OUT-1 element.

[solution →](#)

NOT gate – Solution

Your NOT gate should look like that:



Saving a chip

- Once you checked it is correct, save it (CTRL + S or Command + S, or from the menu).

Saving a chip

- Once you checked it is correct, save it (CTRL + S or Command + S, or from the menu).
- Name it NOT.

Saving a chip

- Once you checked it is correct, save it (CTRL + S or Command + S, or from the menu).
- Name it NOT.
- Open the menu, go to library, then click new collection, name it GATES. **NEW COLLECTION**.

Saving a chip

- Once you checked it is correct, save it (CTRL + S or Command + S, or from the menu).
- Name it NOT.
- Open the menu, go to library, then click new collection, name it GATES. **NEW COLLECTION**.
- Find NOT in the OTHER collection, select it, and click JUMP DOWN until it is under GATES, do the same for NAND.
JUMP DOWN.

Saving a chip

- Once you checked it is correct, save it (CTRL + S or Command + S, or from the menu).
- Name it NOT.
- Open the menu, go to library, then click new collection, name it GATES. **NEW COLLECTION**.
- Find NOT in the OTHER collection, select it, and click JUMP DOWN until it is under GATES, do the same for NAND.
JUMP DOWN.
- ADD TO STARRED the GATES collection, and REMOVE NAND and NOT. **REMOVE FROM STARRED**

AND, OR, NOR, and XOR

Click on MENU, then NEW CHIP. Create the AND, OR, NOR, and XOR gates. Remember to **save each chip** !

A	B	A AND B
0	0	0
1	0	0
0	1	0
1	1	1

A	B	A OR B
0	0	0
1	0	1
0	1	1
1	1	1

A	B	A NOR B
0	0	1
1	0	0
0	1	0
1	1	0

A	B	A XOR B
0	0	0
1	0	1
0	1	1
1	1	0

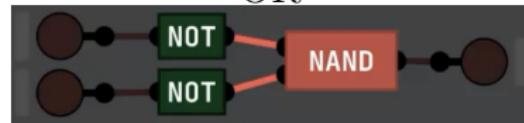
solutions →

AND, OR, NOR, and XOR – Solutions

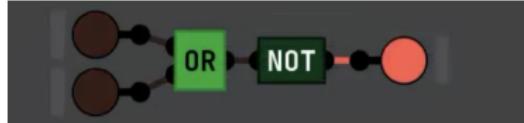
AND



OR



NOR



XOR



Bigger OR, Bigger AND

Create bigger AND and OR gates.

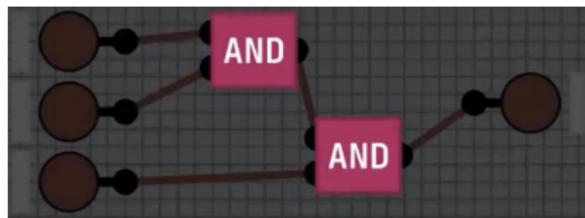
A	B	C	A.B.C
0	0	0	0
1	0	0	0
0	1	0	0
0	0	1	0
1	1	0	0
1	0	1	0
0	1	1	0
1	1	1	1

A	B	C	A+B+C
0	0	0	0
1	0	0	1
0	1	0	1
0	0	1	1
1	1	0	1
1	0	1	1
0	1	1	1
1	1	1	1

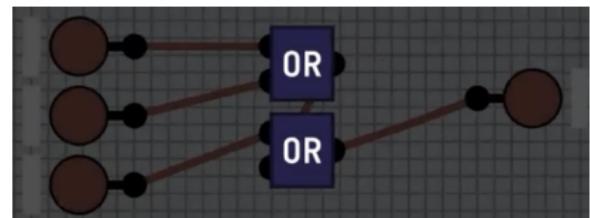
solutions →

Bigger OR, Bigger AND – Solutions

3-AND



3-OR



Logic gates are done!

Good jobs on completing your logic gates!

Logic gates are done!

Good jobs on completing your logic gates!

Next, we will tackle boolean arithmetic.

Logic gates are done!

Good jobs on completing your logic gates!

Next, we will tackle boolean arithmetic.

Take some time to **organize your chips** in the library, review **boolean addition** and **two's complement**, and **check with your peers and teacher** before starting the next section!

Outline

- 1 Project Setup
- 2 Basic Logic gates
- 3 Arithmetic
- 4 Memory 1
- 5 Control Unit part 1
- 6 Buses
- 7 Memory 2
- 8 Building up the CPU
- 9 Control Unit part 2
- 10 Testing the CPU and writing programs

Refresher: Unsigned Binary Numbers

Binary numbers use only two digits: **0** and **1**.

Refresher: Unsigned Binary Numbers

Binary numbers use only two digits: **0** and **1**. Each position represents a power of 2, starting from the right:

Refresher: Unsigned Binary Numbers

Binary numbers use only two digits: **0** and **1**. Each position represents a power of 2, starting from the right:

$$\underbrace{2^3}_{8} \quad \underbrace{2^2}_{4} \quad \underbrace{2^1}_{2} \quad \underbrace{2^0}_{1}$$

Refresher: Unsigned Binary Numbers

Binary numbers use only two digits: **0** and **1**. Each position represents a power of 2, starting from the right:

$$\begin{array}{cccc} 2^3 & 2^2 & 2^1 & 2^0 \\ \underbrace{}_8 & \underbrace{}_4 & \underbrace{}_2 & \underbrace{}_1 \end{array}$$

For example: $1011_2 = (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)$

$$= 8 + 0 + 2 + 1 = 11_{10}$$

Refresher: Unsigned Binary Numbers

Binary numbers use only two digits: **0** and **1**. Each position represents a power of 2, starting from the right:

$$\begin{array}{cccc} 2^3 & 2^2 & 2^1 & 2^0 \\ \underbrace{}_8 & \underbrace{}_4 & \underbrace{}_2 & \underbrace{}_1 \end{array}$$

For example: $1011_2 = (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)$
 $= 8 + 0 + 2 + 1 = 11_{10}$

4-bit unsigned numbers go from:

$$0000_2 = 0_{10} \quad \text{to} \quad 1111_2 = 15_{10}$$

Refresher: Unsigned Binary Numbers

Binary numbers use only two digits: **0** and **1**. Each position represents a power of 2, starting from the right:

$$\begin{array}{cccc} 2^3 & 2^2 & 2^1 & 2^0 \\ \underbrace{}_8 & \underbrace{}_4 & \underbrace{}_2 & \underbrace{}_1 \end{array}$$

For example: $1011_2 = (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)$
 $= 8 + 0 + 2 + 1 = 11_{10}$

4-bit unsigned numbers go from:

$$0000_2 = 0_{10} \quad \text{to} \quad 1111_2 = 15_{10}$$

In general, with n bits, the range is:

$$0 \text{ to } (2^n - 1)$$

Refresher: Signed Binary Numbers

Signed binary numbers can represent both positive and negative values, using **two's complement**.

Refresher: Signed Binary Numbers

Signed binary numbers can represent both positive and negative values, using **two's complement**.

In this system, the **most significant bit (MSB)** is given a **negative value**.

Refresher: Signed Binary Numbers

Signed binary numbers can represent both positive and negative values, using **two's complement**.

In this system, the **most significant bit (MSB)** is given a **negative value**.

For 4-bit numbers:

$$\underbrace{2^3}_{-8} \quad \underbrace{2^2}_4 \quad \underbrace{2^1}_2 \quad \underbrace{2^0}_1$$

Refresher: Signed Binary Numbers

Signed binary numbers can represent both positive and negative values, using **two's complement**.

In this system, the **most significant bit (MSB)** is given a **negative value**.

For 4-bit numbers:

$$\underbrace{2^3}_{-8} \quad \underbrace{2^2}_4 \quad \underbrace{2^1}_2 \quad \underbrace{2^0}_1$$

For positive numbers, it's the same as unsigned.

Refresher: Signed Binary Numbers

Signed binary numbers can represent both positive and negative values, using **two's complement**.

In this system, the **most significant bit (MSB)** is given a **negative value**.

For 4-bit numbers:

$$\begin{array}{cccc} \underbrace{2^3}_{-8} & \underbrace{2^2}_4 & \underbrace{2^1}_2 & \underbrace{2^0}_1 \end{array}$$

For positive numbers, it's the same as unsigned.

For negative numbers, we **invert all bits** and **add 1**:

$$1000_2 = -8_{10}, \quad 1111_2 = -1_{10}$$

Refresher: Signed Binary Numbers

Signed binary numbers can represent both positive and negative values, using **two's complement**.

In this system, the **most significant bit (MSB)** is given a **negative value**.

For 4-bit numbers:

$$\begin{array}{cccc} \underbrace{2^3}_{-8} & \underbrace{2^2}_4 & \underbrace{2^1}_2 & \underbrace{2^0}_1 \end{array}$$

For positive numbers, it's the same as unsigned.

For negative numbers, we **invert all bits** and **add 1**:

$$1000_2 = -8_{10}, \quad 1111_2 = -1_{10}$$

In general, with n bits, the range is:

$$-2^{(n-1)} \text{ to } 2^{(n-1)} - 1$$

Starting small

Let's start by adding two bits together. When adding two bits together, we might have a carry. try to fill in the truth table:

A	B	Sum	Carry
0	0		
1	0		
0	1		
1	1		

solutions →

This is the complete table of what we call a **half-adder** (for reason that will soon become clear)

A	B	Sum	Carry
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Now, try to create a new chip for this component. You'll need two IN-1 and two OUT-1. By the way, you can rename the input and output pin by right clicking on them, so try it, and rename the **top** output pin "carry", and the **bottom** one "sum".

hint →

If you split the truth table in two halves, you might recognize something...

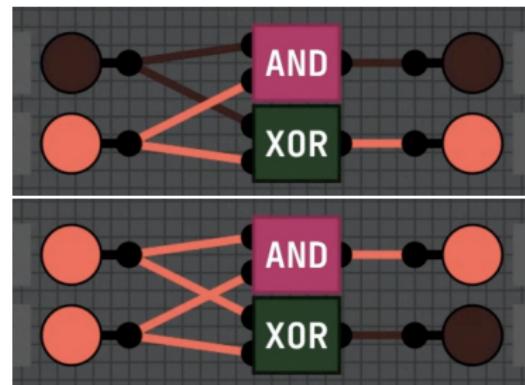
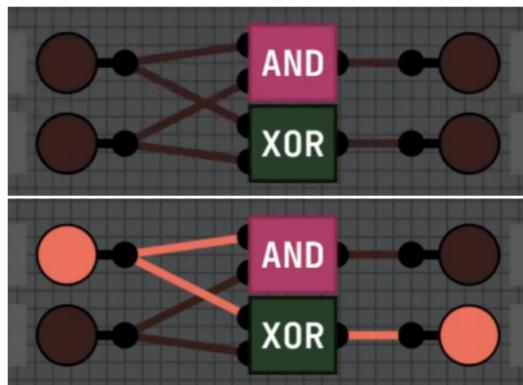
A	B	Carry
0	0	0
1	0	0
0	1	0
1	1	1

A	B	Sum
0	0	0
1	0	1
0	1	1
1	1	0

solution →

Starting Small – Solution

Here is what your chip should look like. Let's check it for all values of inputs:



Full-Adder

Let's move on to a full-adder. A full-adder takes one more input, for an eventual carry coming from the previous digit. Try to fill its truth table:

A	B	C_{in}	Sum	C_{out}
0	0	0		
1	0	0		
0	1	0		
0	0	1		
1	1	0		
1	0	1		
0	1	1		
1	1	1		

solution →

Here is the complete table, try to make the corresponding chip.
You can find some hint on the next slide.

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
0	0	1	1	0
1	1	0	0	1
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

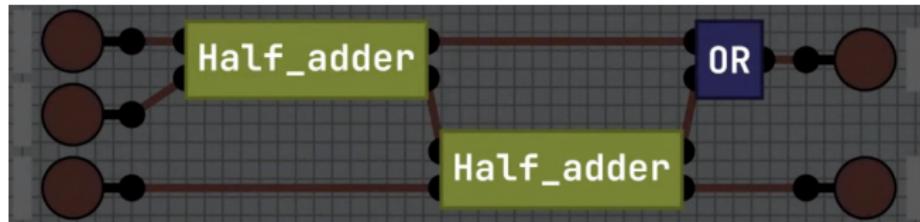
hint →

You can make a full-adder very simply using two half-adder (hence the name).

solution →

Full-Adder – Solution

Here what it looks like, using the half-adder chip we created earlier:



Let's remove the half-adder from our starred components (bottom bar), we won't need it anymore.

Working With Bytes

For this section, we're going to need the MERGE/SPLIT collection from the library. Go ahead and star this collection.

Working With Bytes

For this section, we're going to need the MERGE/SPLIT collection from the library. Go ahead and star this collection. Our new components are:

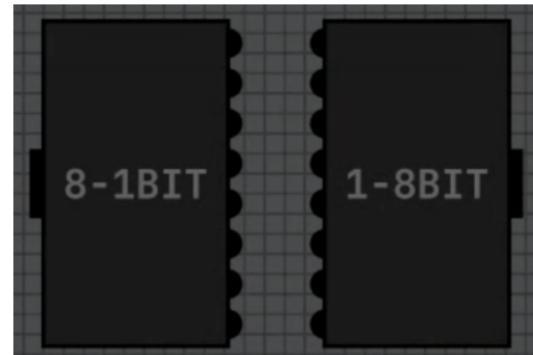
Working With Bytes

For this section, we're going to need the MERGE/SPLIT collection from the library. Go ahead and star this collection. Our new components are:

Byte inputs and outputs (left click on it to display value)



Byte to bits and bits to byte



Byte-Adder

This time, we're going to add two bytes together. This new chip should have:

Byte-Adder

This time, we're going to add two bytes together. This new chip should have:

- two IN-8 inputs
- one OUT-8 output

And it should output the sum of the two bytes.

Byte-Adder

This time, we're going to add two bytes together. This new chip should have:

- two IN-8 inputs
- one OUT-8 output

And it should output the sum of the two bytes.

For now, we will not take care of overflows.

Byte-Adder

This time, we're going to add two bytes together. This new chip should have:

- two IN-8 inputs
- one OUT-8 output

And it should output the sum of the two bytes.

For now, we will not take care of overflows.

Go ahead and try it, there's a hint on next slide if you need it.

hint →

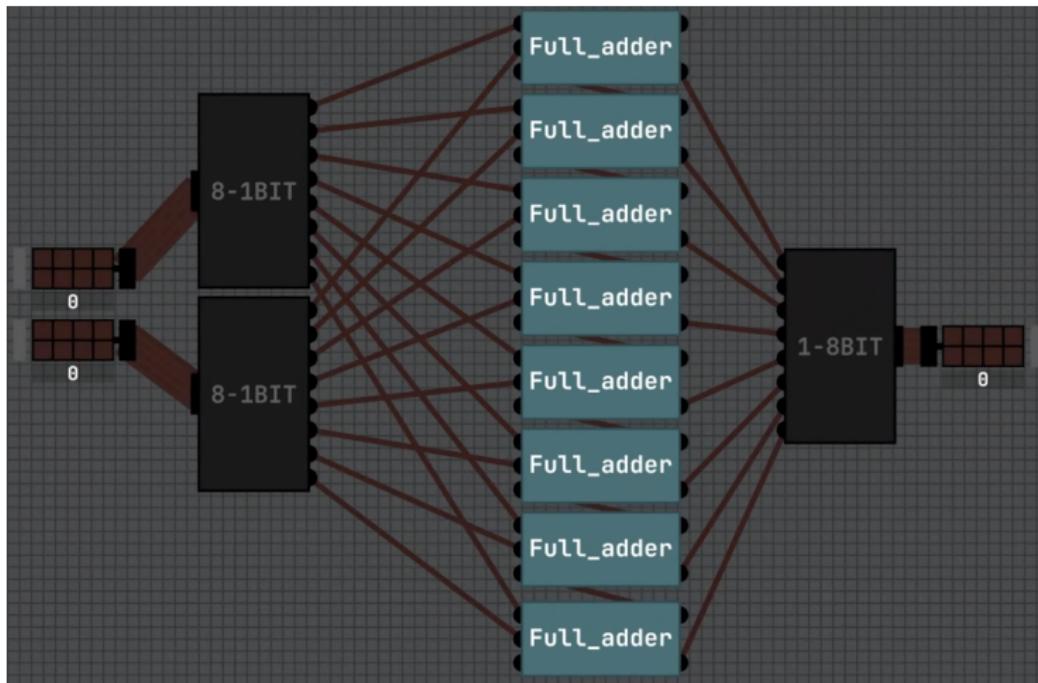
- It's basically just 8 full-adders.

- It's basically just 8 full-adders.
- The carry outs of full-adders are also carry ins for other full adders

solution →

Byte-Adder – Solution

Here is what it should look like (each carry output links to the carry input of the full adder above):



Before saving this component, let's do a small modification.
Let's add a carry input and a carry output.

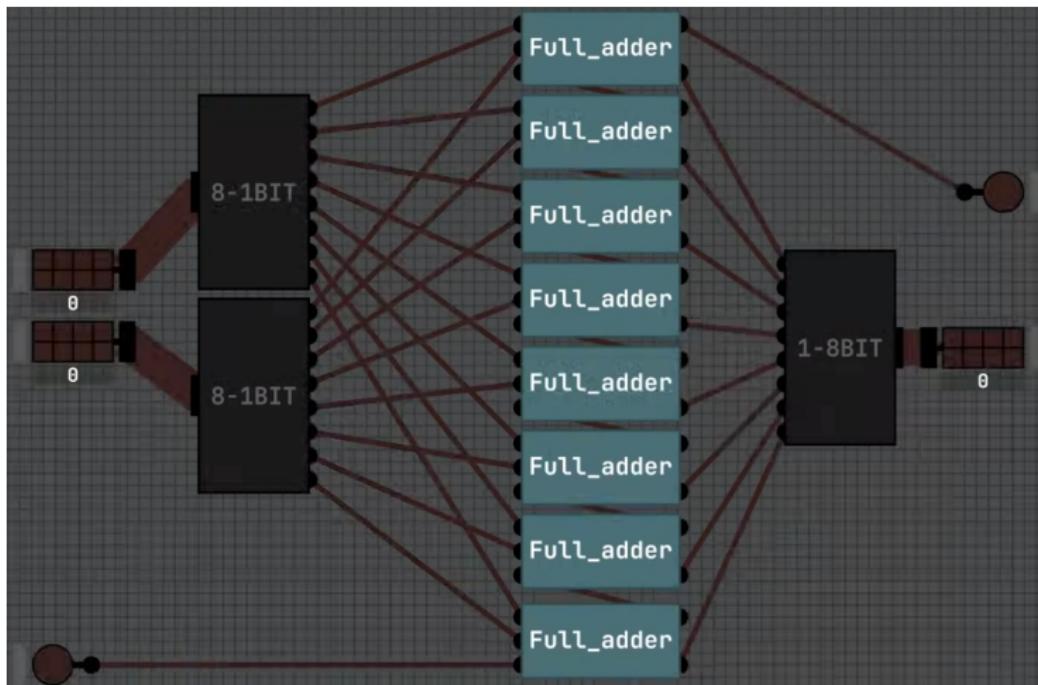
Before saving this component, let's do a small modification.
Let's add a carry input and a carry output.

This is a simple step, don't over-think it!

solution →

Byte-Adder – Solution

Let's save it as "byte_adder"!



ALU

We're almost done with our first CPU component, the ALU (for Arithmetic and Logic Unit).

ALU

We're almost done with our first CPU component, the ALU (for Arithmetic and Logic Unit).

Well, our ALU won't do any logic operation, since we're making a very simple CPU. But still, we're going to call it an ALU.

ALU

We're almost done with our first CPU component, the ALU (for Arithmetic and Logic Unit).

Well, our ALU won't do any logic operation, since we're making a very simple CPU. But still, we're going to call it an ALU.

Let's get over the full requirements for our ALU:

ALU

We're almost done with our first CPU component, the ALU (for Arithmetic and Logic Unit).

Well, our ALU won't do any logic operation, since we're making a very simple CPU. But still, we're going to call it an ALU.

Let's get over the full requirements for our ALU:

- Two byte inputs

ALU

We're almost done with our first CPU component, the ALU (for Arithmetic and Logic Unit).

Well, our ALU won't do any logic operation, since we're making a very simple CPU. But still, we're going to call it an ALU.

Let's get over the full requirements for our ALU:

- Two byte inputs
- One subtraction signal input (one bit)

ALU

We're almost done with our first CPU component, the ALU (for Arithmetic and Logic Unit).

Well, our ALU won't do any logic operation, since we're making a very simple CPU. But still, we're going to call it an ALU.

Let's get over the full requirements for our ALU:

- Two byte inputs
- One subtraction signal input (one bit)
- One byte output

ALU

We're almost done with our first CPU component, the ALU (for Arithmetic and Logic Unit).

Well, our ALU won't do any logic operation, since we're making a very simple CPU. But still, we're going to call it an ALU.

Let's get over the full requirements for our ALU:

- Two byte inputs
- One subtraction signal input (one bit)
- One byte output
- One zero result signal output (one bit)

Subtraction

To subtract, we're going to use the famous math trick:

Subtraction

To subtract, we're going to use the famous math trick:

$$a - b = a + (-b)$$

Subtraction

To subtract, we're going to use the famous math trick:

$$a - b = a + (-b)$$

So we can use our adder. We just need to turn the second operand into a negative one when the subtraction signal is on. Try to figure it out. There are hints on the next page.

hints →

- We need to flip all the bits of b, then add one.

- We need to flip all the bits of b, then add one.
- The adding one could be a carry in our adder.

- We need to flip all the bits of b, then add one.
- The adding one could be a carry in our adder.
- Taking a look at the XOR table might give you an idea on how to flip bits only when the signal is on.

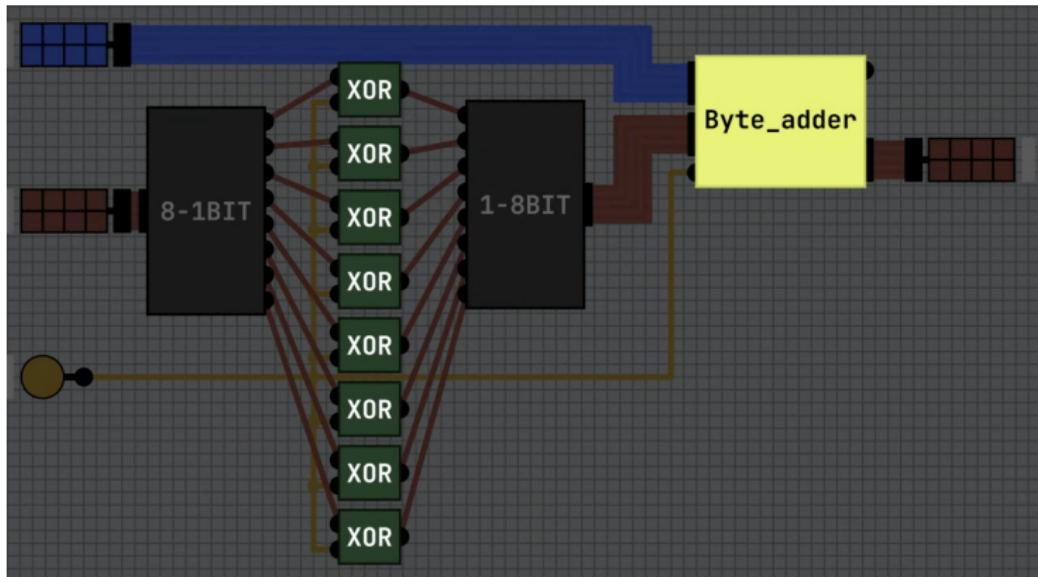
solution →

Almost Done

Here is what it should look like by now:

Almost Done

Here is what it should look like by now:



Adding a Zero Flag

We just need to add one thing to our ALU, a **zero flag**!

Adding a Zero Flag

We just need to add one thing to our ALU, a **zero flag**!

When the result of the computation is zero, we should output 1 on the one bit "zero_result" output.

Adding a Zero Flag

We just need to add one thing to our ALU, a **zero flag**!

When the result of the computation is zero, we should output 1 on the one bit "zero_result" output.

Try to think of something simple to check if the result is zero.

hint →

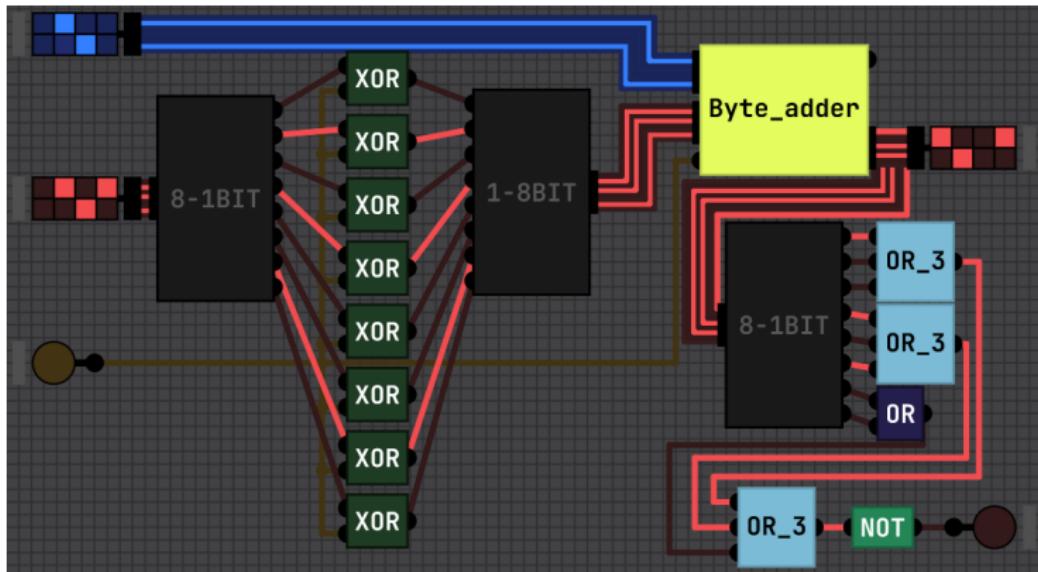
- Split the result into bits.

- Split the result into bits.
- The result is zero if none of the bits are one.

solution →

Your ALU should look something like that:

Your ALU should look something like that:



Arithmetic Done

Good jobs on completing your ALU! This is the first full component of your future CPU!

Arithmetic Done

Good jobs on completing your ALU! This is the first full component of your future CPU!

Next, we will tackle memory part 1, starting from a simple latch, we'll build registers, the smallest and fastest memory type, right inside the CPU.

Arithmetic Done

Good jobs on completing your ALU! This is the first full component of your future CPU!

Next, we will tackle memory part 1, starting from a simple latch, we'll build registers, the smallest and fastest memory type, right inside the CPU.

Take some time to **organize your chips** in the library, and **check with your peers and teacher** before starting the next section!

Outline

- 1 Project Setup
- 2 Basic Logic gates
- 3 Arithmetic
- 4 Memory 1
- 5 Control Unit part 1
- 6 Buses
- 7 Memory 2
- 8 Building up the CPU
- 9 Control Unit part 2
- 10 Testing the CPU and writing programs

A chip that remembers

Take a 5 minutes to try and create a chip that can remember an input.

By that I mean that once the input has been 1, even if you turn the input off, the circuit will keep outputting 1.

If you're stuck, you can get hints on the next slide!

hints →

1.

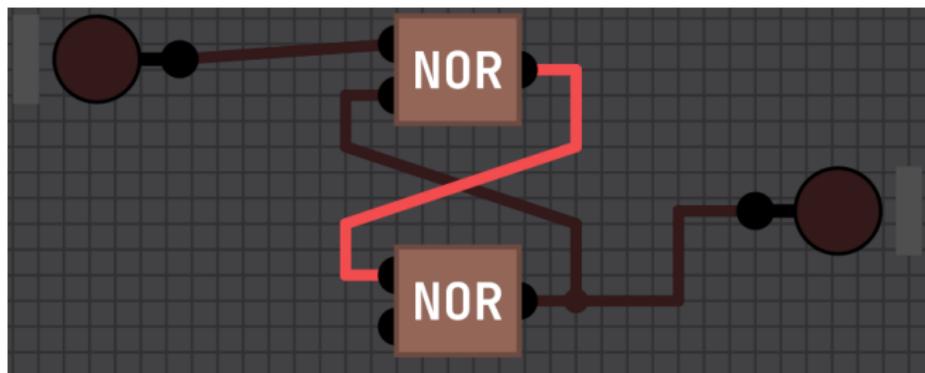
1. You need only two basic gates.
- 2.

1. You need only two basic gates.
2. It's twice the same gate.
- 3.

1. You need only two basic gates.
2. It's twice the same gate.
3. To "remember" you can feed the outputs into the inputs.

solution →

Your chip should look like that:



Let's make it forget

Now, modify your chip a little bit, so that it can forget.

By that I mean that there should be a forget input, and once the forget input has been 1, even if you turn it off, the circuit will keep outputting 0 again.

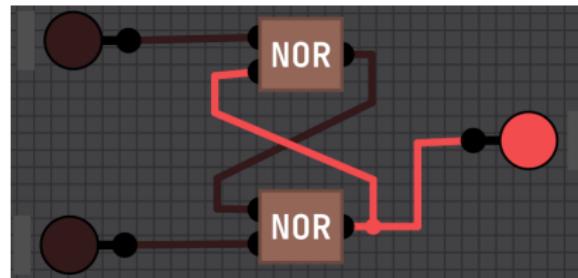
Well, at least until the remember input is activated again!

It's very very simple, don't overthink it, there's almost nothing to add.

solution →

The SR latch

This is an SR latch, the building bloc of memory! Save it as SR_Latch.



The "remember" signal is usually called "SET" (or S), and the "forget" signal is called "RESET" (or R). Hence the SR flip-flop name.

SR latch limitation

SR latches have a big limitation. Try to turn both inputs on.
This is an invalid state.

SR latch limitation

SR latches have a big limitation. Try to turn both inputs on.
This is an invalid state.

We're going to add something before the SR latch to turn it into something better: a data latch, or D-latch

SR latch limitation

SR latches have a big limitation. Try to turn both inputs on. This is an invalid state.

We're going to add something before the SR latch to turn it into something better: a data latch, or D-latch

A D-latch has two inputs, one for the data (0 or 1) and one called enable. When enable is off, nothing should happen when we toggle data. But when enable is on, the D-latch should output and remember the value of data (0 or 1), even if we turn enable off.

hint →

1.

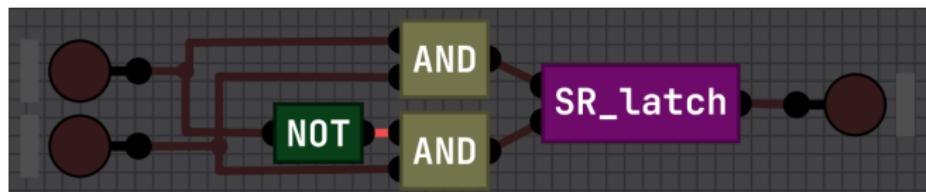
1. When enable is off, both SET and RESET should be off.
- 2.

1. When enable is off, both SET and RESET should be off.
2. When enable is on, SET should be the value of data, and RESET should be the opposite of data.

solution →

D-latch

Here is your completed D-latch. Remember to save it as D_latch !



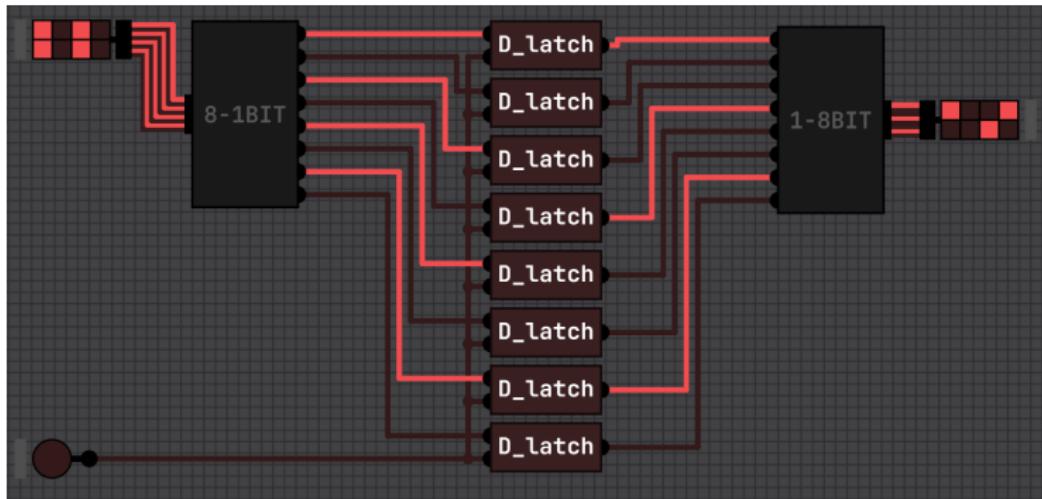
8 bit register version 1

Let's turn it into an 8 bit register. It should have one 8 bit input, one enable input, and one 8 bit output.

Don't overthink it, it pretty straightforward!

[solution](#) →

This is our first version of an 8 bit register. Don't get too attached to it, it's going to be replaced very soon. But for now, save it as `8bit_register`.



Register and ALU test

Let's test our register. Follow these steps carefully:

- Open a new chip, add a **IN-1** and a **IN-8**.

Register and ALU test

Let's test our register. Follow these steps carefully:

- Open a new chip, add a **IN-1** and a **IN-8**.
- Add a **register**. Link both **inputs** to the register, and save the number of your choice inside.

Register and ALU test

Let's test our register. Follow these steps carefully:

- Open a new chip, add a **IN-1** and a **IN-8**.
- Add a **register**. Link both **inputs** to the register, and save the number of your choice inside.
- Set the **IN-1** to false, then unlink the **IN-8** from the **register** (right click, delete).

Register and ALU test

Let's test our register. Follow these steps carefully:

- Open a new chip, add a **IN-1** and a **IN-8**.
- Add a **register**. Link both **inputs** to the register, and save the number of your choice inside.
- Set the **IN-1** to false, then unlink the **IN-8** from the **register** (right click, delete).
- Add the **ALU** we made earlier. Set a different value for the **IN-8**, then link it to one of the input of the **ALU**.

Register and ALU test

Let's test our register. Follow these steps carefully:

- Open a new chip, add a **IN-1** and a **IN-8**.
- Add a **register**. Link both **inputs** to the register, and save the number of your choice inside.
- Set the **IN-1** to false, then unlink the **IN-8** from the **register** (right click, delete).
- Add the **ALU** we made earlier. Set a different value for the **IN-8**, then link it to one of the input of the **ALU**.
- Link the output of the **register** to the other input of the **ALU**, and the output of the **ALU** to the input of the **register**. This is how most computer wire their **ALU**.

Register and ALU test

Let's test our register. Follow these steps carefully:

- Open a new chip, add a **IN-1** and a **IN-8**.
- Add a **register**. Link both **inputs** to the register, and save the number of your choice inside.
- Set the **IN-1** to false, then unlink the **IN-8** from the **register** (right click, delete).
- Add the **ALU** we made earlier. Set a different value for the **IN-8**, then link it to one of the input of the **ALU**.
- Link the output of the **register** to the other input of the **ALU**, and the output of the **ALU** to the input of the **register**. This is how most computer wire their **ALU**.
- Turn on the **IN-1** and observe our issue: the addition doesn't happen only once, it happens multiple times!

Synchronous Registers

We're going to solve our problem by replacing the D-latch with another component, that waits before changing its output when the input is changed.

Synchronous Registers

We're going to solve our problem by replacing the D-latch with another component, that waits before changing its output when the input is changed.

Let's first create a Data flip-flop or D_flip_flop.

Synchronous Registers

We're going to solve our problem by replacing the D-latch with another component, that waits before changing its output when the input is changed.

Let's first create a Data flip-flop or D_flip_flop.

It should have the same 2 inputs as the D-latch, and somewhat save the data value when the enable is off, but should output it only when the enable signal turns on again.

Synchronous Registers

We're going to solve our problem by replacing the D-latch with another component, that waits before changing its output when the input is changed.

Let's first create a Data flip-flop or D_flip_flop.

It should have the same 2 inputs as the D-latch, and somewhat save the data value when the enable is off, but should output it only when the enable signal turns on again.

This is a bit tricky, so there are hints on the next slide.

hint →

1.

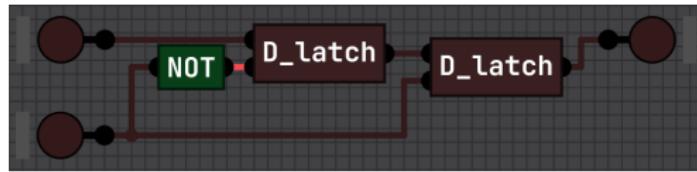
1. Two different things need to happen depending on the value of enable, so split it in two.
- 2.

1. Two different things need to happen depending on the value of enable, so split it in two.
2. The input activates on NOT enable, and the output on enable.
- 3.

1. Two different things need to happen depending on the value of enable, so split it in two.
2. The input activates on NOT enable, and the output on enable.
3. The trick is to use 2 D-latches.

solution →

This is a D_flip_flop, remember to save it!



1 bit register

1 bit registers should have 3 inputs and 1 output:

1 bit register

1 bit registers should have 3 inputs and 1 output:

- IN-1 for the data.
- IN-1 for the enable.
- IN-1 for the clock (yes, our CPU is going to need a clock to keep everything synchronized).

1 bit register

1 bit registers should have 3 inputs and 1 output:

- IN-1 for the data.
- IN-1 for the enable.
- IN-1 for the clock (yes, our CPU is going to need a clock to keep everything synchronized).
- OUT-1 for the data.

1 bit register

1 bit registers should have 3 inputs and 1 output:

- IN-1 for the data.
- IN-1 for the enable.
- IN-1 for the clock (yes, our CPU is going to need a clock to keep everything synchronized).
- OUT-1 for the data.

The register should **update** it's value **each time the clock turns on**. If **enable** is **on**, the new value is the value from **IN-1 data**. If **enable** if **off**, the new value is the **same as the old one**. Try to make that chip!

hints →

1.

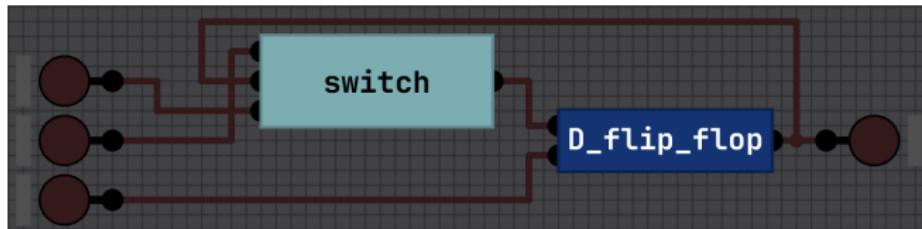
1. The name is tricky, but the enable input **doesn't** go to the enable of the D_flip_flop, **the clock is**.
- 2.

1. The name is tricky, but the enable input **doesn't** go to the enable of the D_flip_flop, **the clock is**.
2. You need some way to **choose between** the old output and the data input depending on enable, then send your selection into the D_flip_flop. You might want to make a dedicated chip outputting one of two entries depending on a third flag input.
- 3.

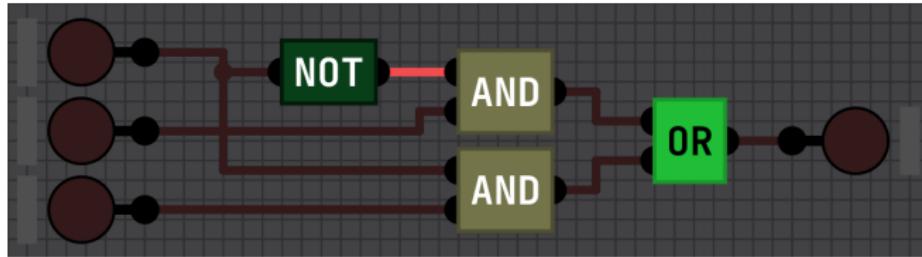
1. The name is tricky, but the enable input **doesn't** go to the enable of the D_flip_flop, **the clock is**.
2. You need some way to **choose between** the old output and the data input depending on enable, then send your selection into the D_flip_flop. You might want to make a dedicated chip outputting one of two entries depending on a third flag input.
3. The input selection can be done with a **NOT**, two **AND**, and a **OR**.

solution →

Here is our final 1-bit register, save it preciously!



where switch is:



Fixing the 8 bit register

Now fix the 8 bit register. It should now have an additional input, the clock.

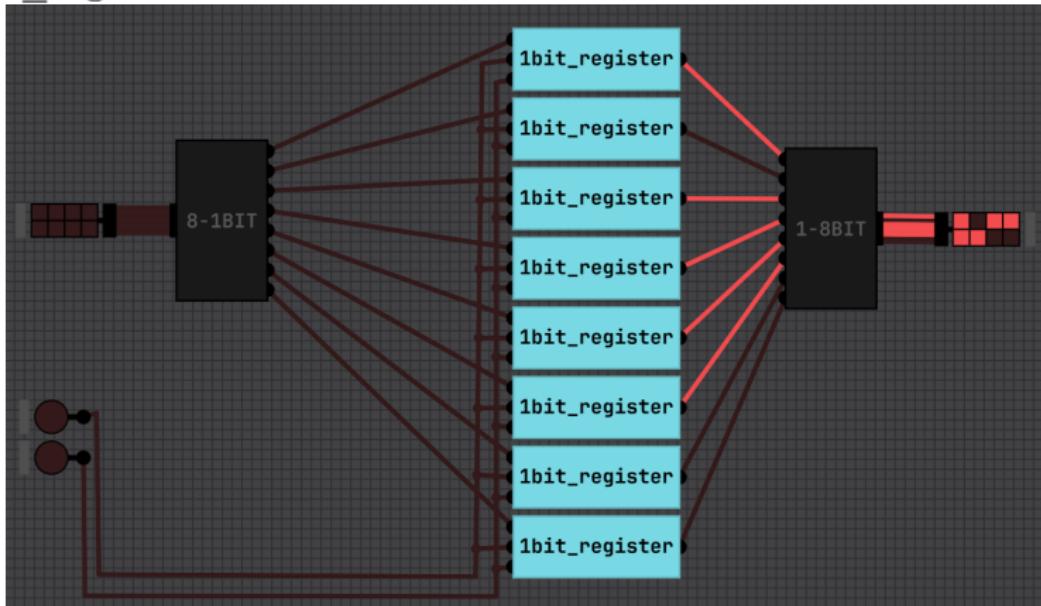
Fixing the 8 bit register

Now fix the 8 bit register. It should now have an additional input, the clock.

It is still basically eight 1 bit registers put together, so don't overthink it!

solution →

Here is the complete and final 8 bit register, save it as 8bit_register.



Register Done

Good jobs on completing your Register! This is also a full component of your future CPU! We're going to use both 1 bit and 8 bit registers.

Register Done

Good jobs on completing your Register! This is also a full component of your future CPU! We're going to use both 1 bit and 8 bit registers.

Next, we will start working on our Control Unit. This is the part of the CPU that decodes our program to turn it into signals for our ALU, registers, and later on, RAM.

Register Done

Good jobs on completing your Register! This is also a full component of your future CPU! We're going to use both 1 bit and 8 bit registers.

Next, we will start working on our Control Unit. This is the part of the CPU that decodes our program to turn it into signals for our ALU, registers, and later on, RAM.

Take some time to **organize your chips** in the library, and **check with your peers and teacher** before starting the next section!

Outline

- 1 Project Setup
- 2 Basic Logic gates
- 3 Arithmetic
- 4 Memory 1
- 5 Control Unit part 1
- 6 Buses
- 7 Memory 2
- 8 Building up the CPU
- 9 Control Unit part 2
- 10 Testing the CPU and writing programs

Decoding bits

We use binary to make information fit in a very limited space. That way, an 8 bit computer like ours could have up to 256^* different instructions. Our will only have seven.

Decoding bits

We use binary to make information fit in a very limited space. That way, an 8 bit computer like ours could have up to 256^* different instructions. Our will only have seven.

We need some kind of component to decode a single number into a set of signals that will go to other components.

Decoding bits

We use binary to make information fit in a very limited space. That way, an 8 bit computer like ours could have up to 256* different instructions. Our will only have seven.

We need some kind of component to decode a single number into a set of signals that will go to other components.

Let's start simple, by decoding one bit only.

* assuming we use the full 8 bit for opcodes...

1 bit decoder

Let's make a chip with one IN-1 seen as a binary number, and two OUT-1, seen as signals (or flags).

1 bit decoder

Let's make a chip with one IN-1 seen as a binary number, and two OUT-1, seen as signals (or flags).

One output should be on if the input is zero, and the other output should be on if the input is one.

1 bit decoder

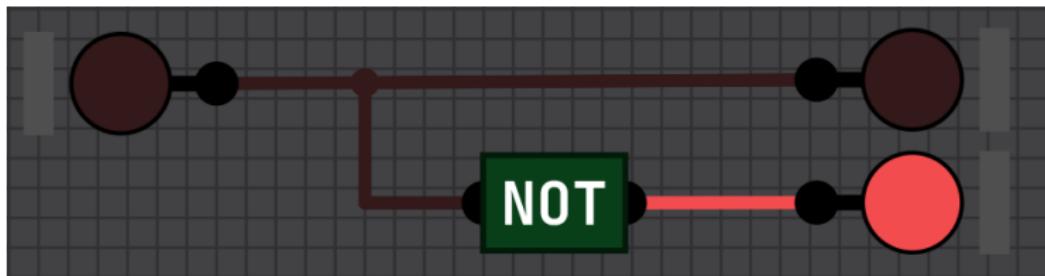
Let's make a chip with one IN-1 seen as a binary number, and two OUT-1, seen as signals (or flags).

One output should be on if the input is zero, and the other output should be on if the input is one.

We'll name these output "0" and "1". No hints this time, it's very simple!

solution →

Your chip should look like this. Save it as 1bit_decoder



But a one bit decoder is not enough, so let's make a two bit decoder!

2 bit decoder

Now, let's make a chip with two IN-1 seen as a binary number, and four OUT-1.

2 bit decoder

Now, let's make a chip with two IN-1 seen as a binary number, and four OUT-1.

Same as before, there should also be one and exactly one output on, depending on the binary value of the inputs.

2 bit decoder

Now, let's make a chip with two IN-1 seen as a binary number, and four OUT-1.

Same as before, there should also be one and exactly one output on, depending on the binary value of the inputs.

We'll name these output "0", "1", "2", and "3".

2 bit decoder

Now, let's make a chip with two IN-1 seen as a binary number, and four OUT-1.

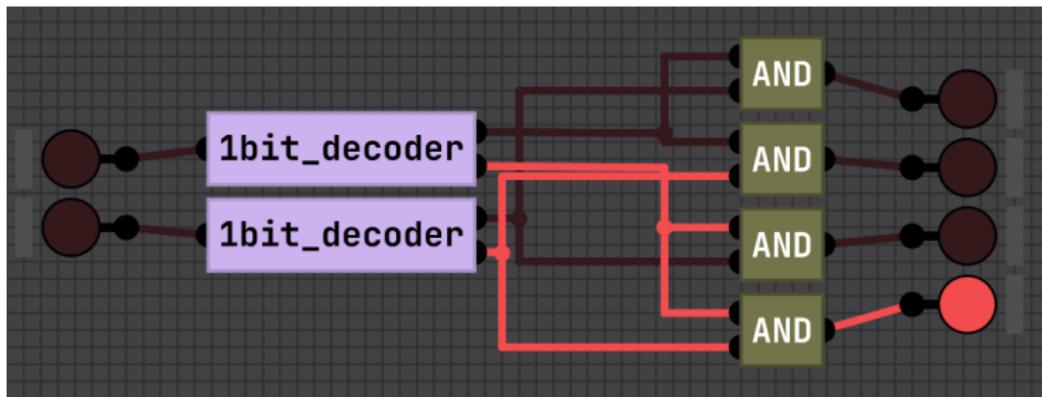
Same as before, there should also be one and exactly one output on, depending on the binary value of the inputs.

We'll name these output "0", "1", "2", and "3".

Remember that lazy is good, so use the one bit decoder we just made!

solution →

Your chip should look like this. Save it as 2bit_decoder



But two bits is still not enough, I want MORE! Let's make a three bit decoder!

3 bit decoder

Now, let's make a chip with three IN-1 seen as a binary number, and

3 bit decoder

Now, let's make a chip with three IN-1 seen as a binary number, and four OUT-1.

3 bit decoder

Now, let's make a chip with three IN-1 seen as a binary number, and four OUT-1.

Same as before, there should also be one and exactly one output on, depending on the binary value of the inputs.

3 bit decoder

Now, let's make a chip with three IN-1 seen as a binary number, and four OUT-1.

Same as before, there should also be one and exactly one output on, depending on the binary value of the inputs.

We'll name these output

3 bit decoder

Now, let's make a chip with three IN-1 seen as a binary number, and four OUT-1.

Same as before, there should also be one and exactly one output on, depending on the binary value of the inputs.

We'll name these output "0", "1", "2", "3", "4", "5", "6", and "7".

3 bit decoder

Now, let's make a chip with three IN-1 seen as a binary number, and four OUT-1.

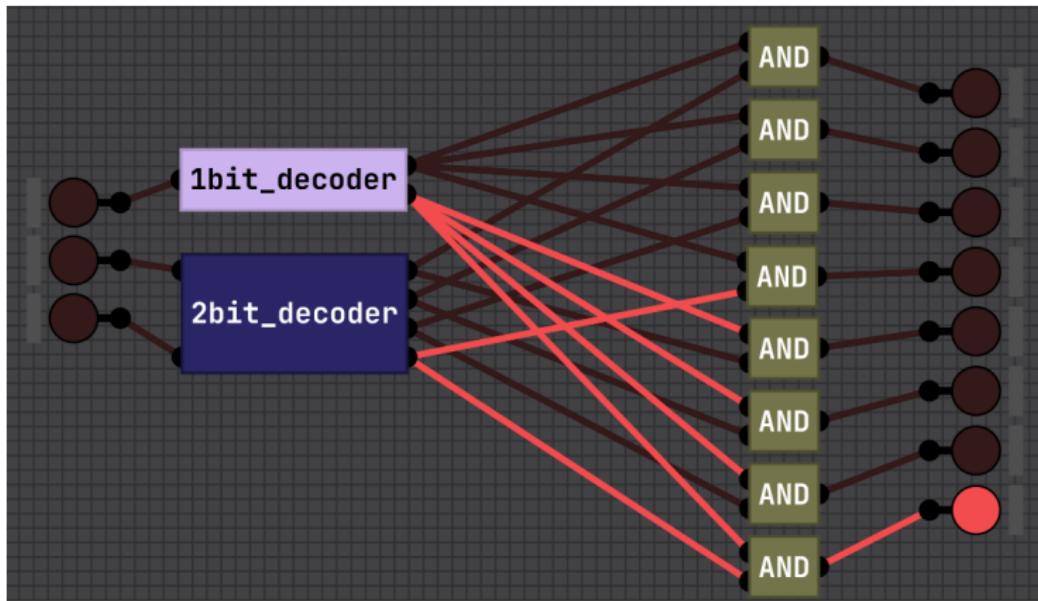
Same as before, there should also be one and exactly one output on, depending on the binary value of the inputs.

We'll name these output "0", "1", "2", "3", "4", "5", "6", and "7".

Remember that lazy is good, so use the two bit decoder we just made!

solution →

Your chip should look like this. Save it as 3bit_decoder



Don't worry, three bits are going to be enough for this project, but I think you know how to do more if you need it!

The clock

Another key sub-component of the Control Unit is the clock.

The clock

Another key sub-component of the Control Unit is the clock.

The clock is a simple component that outputs a signal that turns on and off at a regular interval. Since this is NOT made with transistors, but rather with a quartz cristal, it is given by Digital Logic Sim as a base component.

The clock

Another key sub-component of the Control Unit is the clock.

The clock is a simple component that outputs a signal that turns on and off at a regular interval. Since this is NOT made with transistors, but rather with a quartz cristal, it is given by Digital Logic Sim as a base component.

However, our CPU is going to operate under some sort of Fetch-Decode-Execute cycle, and the clock will help us synchronize everything.

The clock

Another key sub-component of the Control Unit is the clock.

The clock is a simple component that outputs a signal that turns on and off at a regular interval. Since this is NOT made with transistors, but rather with a quartz cristal, it is given by Digital Logic Sim as a base component.

However, our CPU is going to operate under some sort of Fetch-Decode-Execute cycle, and the clock will help us synchronize everything.

So we need to make a counter that counts the clock ticks and outputs the current step of the cycle.

our FDE cycle has 4 steps, so we're going to make a 4 step counter.

our FDE cycle has 4 steps, so we're going to make a 4 step counter.

So let's make a 2 bit binary counter.

our FDE cycle has 4 steps, so we're going to make a 4 step counter.

So let's make a 2 bit binary counter.

It should have

our FDE cycle has 4 steps, so we're going to make a 4 step counter.

So let's make a 2 bit binary counter.

It should have one IN-1 input for the clock,

and

our FDE cycle has 4 steps, so we're going to make a 4 step counter.

So let's make a 2 bit binary counter.

It should have one IN-1 input for the clock,

and two OUT-1 outputs for the current count (in binary).

our FDE cycle has 4 steps, so we're going to make a 4 step counter.

So let's make a 2 bit binary counter.

It should have one IN-1 input for the clock,

and two OUT-1 outputs for the current count (in binary).

As a first hint, let me tell you that you will need to use **bit registers** to make this component.

hint →

1.

1. You will need two 1 bit registers, one for each bit of the count.
- 2.

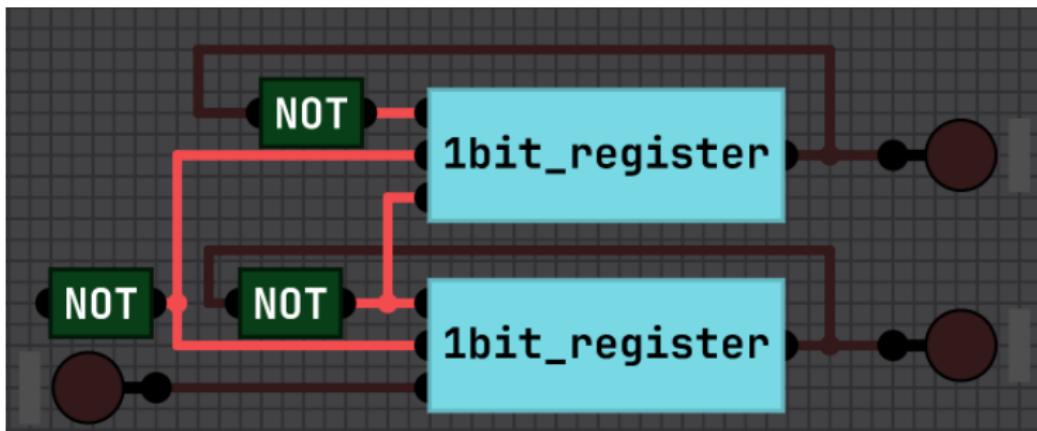
1. You will need two 1 bit registers, one for each bit of the count.
2. The least significant bit (LSB) toggles at each clock tick.
- 3.

1. You will need two 1 bit registers, one for each bit of the count.
2. The least significant bit (LSB) toggles at each clock tick.
3. Toggling means setting it to NOT its current value.
- 4.

1. You will need two 1 bit registers, one for each bit of the count.
2. The least significant bit (LSB) toggles at each clock tick.
3. Toggling means setting it to NOT its current value.
4. The most significant bit (MSB) toggles when the LSB goes from 1 to 0.

solution →

Here is what your counter should look like. Save it as 2bit_counter.



FDE controller

Our cycle follows these steps:

FDE controller

Our cycle follows these steps:

Fetch: Load the instruction from ROM into the instruction register.

Increment: Increment the program counter.

Decode: Decode the instruction opcode in the instruction register.

Execute: Execute the instruction.

FDE controller

Our cycle follows these steps:

Fetch: Load the instruction from ROM into the instruction register.

Increment: Increment the program counter.

Decode: Decode the instruction opcode in the instruction register.

Execute: Execute the instruction.

So let's make a component that outputs which step of the cycle we're in.

It should have:

It should have:

Inputs:

It should have:

Inputs: None.

It should have:

Inputs: None.

Outputs:

It should have:

Inputs: None.

Outputs:

- one OUT-1 for the Fetch step.
- one OUT-1 for the Increment step.
- one OUT-1 for the Decode step.
- one OUT-1 for the Execute step.

It should have:

Inputs: None.

Outputs:

- one OUT-1 for the Fetch step.
- one OUT-1 for the Increment step.
- one OUT-1 for the Decode step.
- one OUT-1 for the Execute step.

At any time, **one and only one** output should be on, depending on the current step of the cycle.

hint →

1.

1. Use the 2 bit counter we just made to keep track of the current step.
- 2.

1. Use the 2 bit counter we just made to keep track of the current step.
2. Use the 2 bit decoder we made earlier to decode the counter output into the four step signals.

solution →

Here is what your FDE controller should look like. Name the outputs: Fetch, Increment, Decode, Execute.

Save it as FDE_controller.

Instructions

This is the set of instructions our CPU will be able to execute.

Instructions

This is the set of instructions our CPU will be able to execute.

Name	Opcode	Operand	Description
LOAD	000	Address	Load a value from memory into the accumulator
STORE	001	Address	Store the value of the accumulator into memory
ADD	010	Address	Add a value in memory to the accumulator
SUB	011	Address	Subtract a value in memory from the accumulator
JUMP	100	Value	Jump to a specific instruction address
JZ	101	Value	Jump to a specific instruction address if zero flag is set
IMM	110	Value	Load a direct value into the accumulator

instruction decoder

Let's make a chip that decodes the instruction opcode.

instruction decoder

Let's make a chip that decodes the instruction opcode.

It should have:

instruction decoder

Let's make a chip that decodes the instruction opcode.

It should have:

Inputs:

instruction decoder

Let's make a chip that decodes the instruction opcode.

It should have:

Inputs:

- one IN-8 for the instruction opcode.

instruction decoder

Let's make a chip that decodes the instruction opcode.

It should have:

Inputs:

- one IN-8 for the instruction opcode.

Outputs:

instruction decoder

Let's make a chip that decodes the instruction opcode.

It should have:

Inputs:

- one IN-8 for the instruction opcode.

Outputs:

- one OUT-1 for each instruction (7 total, name them correctly).

instruction decoder

Let's make a chip that decodes the instruction opcode.

It should have:

Inputs:

- one IN-8 for the instruction opcode.

Outputs:

- one OUT-1 for each instruction (7 total, name them correctly).

Don't overthink it, the name should be a pretty good hint!

solution →

Here is what your instruction decoder should look like. Save it as `instruction_decoder`.

Control Unit draft

Let's make a first draft of our Control Unit (CU).

Control Unit draft

Let's make a first draft of our Control Unit (CU).

For now, it's going to be very simple, and full of holes.

Control Unit draft

Let's make a first draft of our Control Unit (CU).

For now, it's going to be very simple, and full of holes.

We will just setup the instruction decoder and the
FDE_controller for now.

Control Unit draft

Let's make a first draft of our Control Unit (CU).

For now, it's going to be very simple, and full of holes.

We will just setup the instruction decoder and the FDE_controller for now.

We will continue to come back to it regularly during the next sections to add outputs (AKA control signals) and wire them up as we need them.

Control Unit draft

Let's make a first draft of our Control Unit (CU).

For now, it's going to be very simple, and full of holes.

We will just setup the instruction decoder and the FDE_controller for now.

We will continue to come back to it regularly during the next sections to add outputs (AKA control signals) and wire them up as we need them.

For now, it should have:

For now, it should have:

Inputs:

For now, it should have:

Inputs:

- one IN-8 for the instruction opcode.
- one IN-1 for the zero flag from the ALU.

For now, it should have:

Inputs:

- one IN-8 for the instruction opcode.
- one IN-1 for the zero flag from the ALU.

Sub-components:

For now, it should have:

Inputs:

- one IN-8 for the instruction opcode.
- one IN-1 for the zero flag from the ALU.

Sub-components:

- our instruction_decoder.
- our FDE_controller.

For now, it should have:

Inputs:

- one IN-8 for the instruction opcode.
- one IN-1 for the zero flag from the ALU.

Sub-components:

- our instruction_decoder.
- our FDE_controller.

Outputs:

For now, it should have:

Inputs:

- one IN-8 for the instruction opcode.
- one IN-1 for the zero flag from the ALU.

Sub-components:

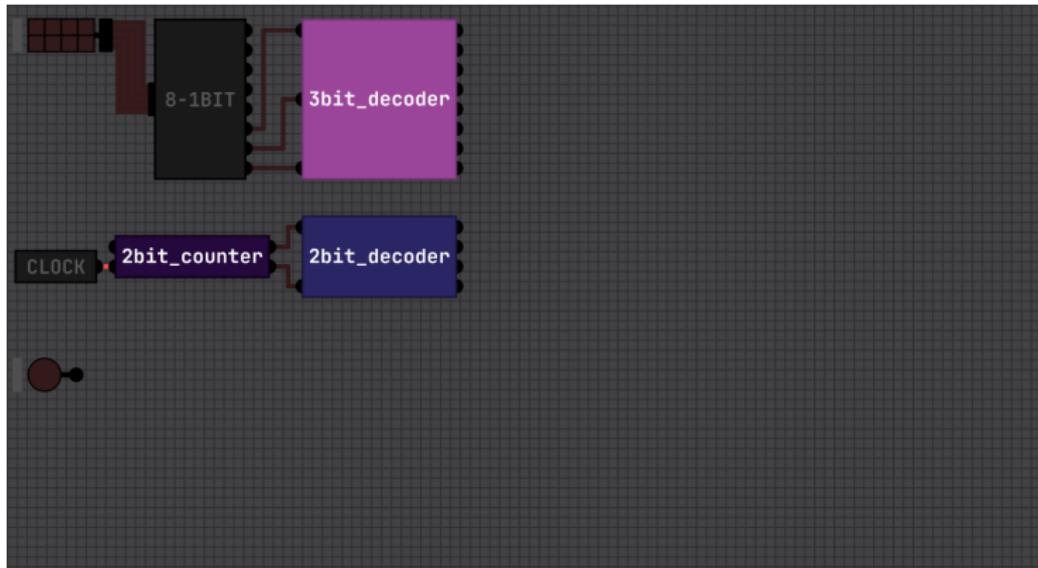
- our instruction_decoder.
- our FDE_controller.

Outputs:

- (none for now)

solution →

Here is what your Control Unit draft should look like. Save it as CU.



If you want, you can start thinking about which control signals will be needed to execute the instructions.

If you want, you can start thinking about which control signals will be needed to execute the instructions.

For now, you know about the ALU, and that we will have a accumulator register in the CPU. You can already start thinking about which opcodes require to set the enable flag of the accumulator on.

If you want, you can start thinking about which control signals will be needed to execute the instructions.

For now, you know about the ALU, and that we will have a accumulator register in the CPU. You can already start thinking about which opcodes require to set the enable flag of the accumulator on.

But don't spend too much time on it, we will come back to this in the next sections.

Control Unit Done... for now

Good jobs on completing your Control Unit draft! This is also a top level component of your future CPU!

Control Unit Done... for now

Good jobs on completing your Control Unit draft! This is also a top level component of your future CPU!

Next, we will work on the buses that will connect all our components together.

Control Unit Done... for now

Good jobs on completing your Control Unit draft! This is also a top level component of your future CPU!

Next, we will work on the buses that will connect all our components together.

Take some time to **organize your chips** in the library, and **check with your peers and teacher** before starting the next section!

Outline

- 1 Project Setup
- 2 Basic Logic gates
- 3 Arithmetic
- 4 Memory 1
- 5 Control Unit part 1
- 6 Buses
- 7 Memory 2
- 8 Building up the CPU
- 9 Control Unit part 2
- 10 Testing the CPU and writing programs

What are buses exactly?

By now, you might have experienced a weird bug where wires flicker on and off.

What are buses exactly?

By now, you might have experienced a weird bug where wires flicker on and off.

This is because when multiple outputs are connected to a single input, the simulator can't decide which output to listen to, and randomly picks one.

What are buses exactly?

By now, you might have experienced a weird bug where wires flicker on and off.

This is because when multiple outputs are connected to a single input, the simulator can't decide which output to listen to, and randomly picks one.

In real computers, this would be because multiple components are trying to output to this cable at different voltages, which would damage the components.

What are buses exactly?

By now, you might have experienced a weird bug where wires flicker on and off.

This is because when multiple outputs are connected to a single input, the simulator can't decide which output to listen to, and randomly picks one.

In real computers, this would be because multiple components are trying to output to this cable at different voltages, which would damage the components.

Cables actually have 3 states: high voltage (1), low voltage (0), and disconnected (deep black).

Buses are a way to centralise connections between multiple components.

Buses are a way to centralise connections between multiple components.

For example, the data bus is the main bus of the CPU, it connects the ALU, registers, and memory together.

Buses are a way to centralise connections between multiple components.

For example, the data bus is the main bus of the CPU, it connects the ALU, registers, and memory together.

At any time, only one component should be allowed to output to the data bus, while all the others should be disconnected.

Buses are a way to centralise connections between multiple components.

For example, the data bus is the main bus of the CPU, it connects the ALU, registers, and memory together.

At any time, only one component should be allowed to output to the data bus, while all the others should be disconnected.

This means we need a component that changes an output to disconnected when we don't want it to output to the bus.

Buses are a way to centralise connections between multiple components.

For example, the data bus is the main bus of the CPU, it connects the ALU, registers, and memory together.

At any time, only one component should be allowed to output to the data bus, while all the others should be disconnected.

This means we need a component that changes an output to disconnected when we don't want it to output to the bus.

Meet the tri-state buffer!

Tri-state buffer

Go to library, and star the tri-state buffer.

Tri-state buffer

Go to library, and star the tri-state buffer.

Experiment with it a bit to understand how it works.

Tri-state buffer

Go to library, and star the tri-state buffer.

Experiment with it a bit to understand how it works.

When you're ready, make a byte version of it.

[solution](#) →

Here is what your byte tri-state buffer should look like. Save it as byte_tri_state.

Bus practice

Find the bus collection form the library, and star it.

Bus practice

Find the bus collection form the library, and star it.

Let's first practice a one bit bus.

Bus practice

Find the bus collection from the library, and star it.

Let's first practice a one bit bus.

Make a chip with two IN-1 inputs, one OUT-1 output, and two control IN-1 inputs.

Bus practice

Find the bus collection from the library, and star it.

Let's first practice a one bit bus.

Make a chip with two IN-1 inputs, one OUT-1 output, and two control IN-1 inputs.

Each pair of input/control should be linked to a tri-state buffer, and the outputs of both buffers should be linked to the output of the bus.

Bus practice

Find the bus collection form the library, and star it.

Let's first practice a one bit bus.

Make a chip with two IN-1 inputs, one OUT-1 output, and two control IN-1 inputs.

Each pair of input/control should be linked to a tri-state buffer, and the outputs of both buffers should be linked to the output of the bus.

When a control input is on, the corresponding input should be outputed to the bus. If both control inputs are off, the bus should be disconnected. If both control inputs are on... well, don't do that!

Byte practice

Just for practice, do the same with byte inputs and outputs.

Byte practice

Just for practice, do the same with byte inputs and outputs.

The control inputs are still one bit each.

Byte practice

Just for practice, do the same with byte inputs and outputs.

The control inputs are still one bit each.

Does it behave as we would expect? good!

Byte practice

Just for practice, do the same with byte inputs and outputs.

The control inputs are still one bit each.

Does it behave as we would expect? good!

Buses done!

Good jobs on completing your buses! These are also key components of your future CPU!

Buses done!

Good jobs on completing your buses! These are also key components of your future CPU!

Next, we will work on the second part of memory, the RAM.

Buses done!

Good jobs on completing your buses! These are also key components of your future CPU!

Next, we will work on the second part of memory, the RAM.

Take some time to **organize your chips** in the library, and **check with your peers and teacher** before starting the next section!

Outline

- 1 Project Setup
- 2 Basic Logic gates
- 3 Arithmetic
- 4 Memory 1
- 5 Control Unit part 1
- 6 Buses
- 7 Memory 2
- 8 Building up the CPU
- 9 Control Unit part 2
- 10 Testing the CPU and writing programs

RAM requirements

We're going to make a small RAM for our CPU.

RAM requirements

We're going to make a small RAM for our CPU.

It should have:

RAM requirements

We're going to make a small RAM for our CPU.

It should have:

- one IN-8 input for the data to write.
- one IN-8 input for the address to read/write.
- one IN-1 input for the write enable signal.
- one OUT-8 output for the data read.

RAM requirements

We're going to make a small RAM for our CPU.

It should have:

- one IN-8 input for the data to write.
- one IN-8 input for the address to read/write.
- one IN-1 input for the write enable signal.
- one OUT-8 output for the data read.

When write enable is on, the data at the IN-8 input should be written into memory at the given address.

RAM requirements

We're going to make a small RAM for our CPU.

It should have:

- one IN-8 input for the data to write.
- one IN-8 input for the address to read/write.
- one IN-1 input for the write enable signal.
- one OUT-8 output for the data read.

When write enable is on, the data at the IN-8 input should be written into memory at the given address.

At all time, the data at the given address should be outputted on the OUT-8 output. We don't have to worry about timing issues for the RAM. So we won't need the fancy logic of registers.

Making the RAM

Doing all of that at once would be too much though! so let's break it down.

Making the RAM

Doing all of that at once would be too much though! so let's break it down.

You might have realised that our RAM isn't connected to a clock. This means that we can't use registers to store the data, since they need a clock signal to update their value.

Making the RAM

Doing all of that at once would be too much though! so let's break it down.

You might have realised that our RAM isn't connected to a clock. This means that we can't use registers to store the data, since they need a clock signal to update their value.

Instead, we're going to use D-latches to store each bit of data.

1 bit RAM cell

So first, let's make a 1 bit RAM cell. It should have:

1 bit RAM cell

So first, let's make a 1 bit RAM cell. It should have:

- one IN-1 input for the data to write.
- one IN-1 input for the write enable signal.
- two one bit inputs, named row and column.
- one OUT-1 output for the data read.

1 bit RAM cell

So first, let's make a 1 bit RAM cell. It should have:

- one IN-1 input for the data to write.
- one IN-1 input for the write enable signal.
- two one bit inputs, named row and column.
- one OUT-1 output for the data read.

The cell should be active only when both row and column inputs are on.

1 bit RAM cell

So first, let's make a 1 bit RAM cell. It should have:

- one IN-1 input for the data to write.
- one IN-1 input for the write enable signal.
- two one bit inputs, named row and column.
- one OUT-1 output for the data read.

The cell should be active only when both row and column inputs are on.

If active, the data stored in the cell should be outputted on the OUT-1 output. When not active, the output should be disconnected. Be careful, I said disconnected, not just zero!

1 bit RAM cell

So first, let's make a 1 bit RAM cell. It should have:

- one IN-1 input for the data to write.
- one IN-1 input for the write enable signal.
- two one bit inputs, named row and column.
- one OUT-1 output for the data read.

The cell should be active only when both row and column inputs are on.

If active, the data stored in the cell should be outputted on the OUT-1 output. When not active, the output should be disconnected. Be careful, I said disconnected, not just zero!

If active and enabled, the data at the IN-1 input should be written into the cell.

hint →

1.

1. The combined row and column inputs act together as a control signal for both writing and outputting.
- 2.

1. The combined row and column inputs act together as a control signal for both writing and outputting.
2. For writing, we **also** need enable to be on for the D-latch.
- 3.

1. The combined row and column inputs act together as a control signal for both writing and outputting.
2. For writing, we **also** need enable to be on for the D-latch.
3. For outputting, you will need a tri-state buffer to disconnect the output when not active.

solution →

Here is what your 1 bit RAM cell should look like. Save it as
1bit_RAM_cell.

MEM 16x1

Now, let's make a small memory with 16 of these cells.

MEM 16x1

Now, let's make a small memory with 16 of these cells.

It should have:

MEM 16x1

Now, let's make a small memory with 16 of these cells.

It should have:

- one IN-1 input for the data to write.
- one IN-4 input for the address to read/write.
- one IN-1 input for the write enable signal.
- one OUT-1 output for the data read.
- its own row and column inputs to decide if this 16 cell is active.

MEM 16x1

Now, let's make a small memory with 16 of these cells.

It should have:

- one IN-1 input for the data to write.
- one IN-4 input for the address to read/write.
- one IN-1 input for the write enable signal.
- one OUT-1 output for the data read.
- its own row and column inputs to decide if this 16 cell is active.

The address input should be used to select which 1 cell is active.

MEM 16x1

Now, let's make a small memory with 16 of these cells.

It should have:

- one IN-1 input for the data to write.
- one IN-4 input for the address to read/write.
- one IN-1 input for the write enable signal.
- one OUT-1 output for the data read.
- its own row and column inputs to decide if this 16 cell is active.

The address input should be used to select which 1 cell is active.

The row and column input should be used to check if this 16 cell is active

MEM 16x1

Now, let's make a small memory with 16 of these cells.

It should have:

- one IN-1 input for the data to write.
- one IN-4 input for the address to read/write.
- one IN-1 input for the write enable signal.
- one OUT-1 output for the data read.
- its own row and column inputs to decide if this 16 cell is active.

The address input should be used to select which 1 cell is active.

The row and column input should be used to check if this 16 cell is active

hint →

1.

1. Use 16 of the MEM 1 cells we made earlier.

1. Use 16 of the MEM 1 cells we made earlier.
- 2.

1. Use 16 of the MEM 1 cells we made earlier.
2. You will need to decode the 4 bit address into 4 row and 4 column signals.
- 3.

1. Use 16 of the MEM 1 cells we made earlier.
2. You will need to decode the 4 bit address into 4 row and 4 column signals.
3. Use two bits for the row, and two bits for the column.
- 4.

1. Use 16 of the MEM 1 cells we made earlier.
2. You will need to decode the 4 bit address into 4 row and 4 column signals.
3. Use two bits for the row, and two bits for the column.
4. Each cell will get the same write enable and data input.
- 5.

1. Use 16 of the MEM 1 cells we made earlier.
2. You will need to decode the 4 bit address into 4 row and 4 column signals.
3. Use two bits for the row, and two bits for the column.
4. Each cell will get the same write enable and data input.
5. The outputs of all cells should be linked to a bus, so that only the active cell outputs to the bus.

solution →

Here is what your 16x1 RAM should look like. Save it as
MEM_16x1.

MEM 256x1

Let's scale it up to 256 bits!

MEM 256x1

Let's scale it up to 256 bits!

It should have:

MEM 256x1

Let's scale it up to 256 bits!

It should have:

- one IN-1 input for the data to write.
- one IN-8 input for the address to read/write.
- one IN-1 input for the write enable signal.
- one OUT-1 output for the data read.

MEM 256x1

Let's scale it up to 256 bits!

It should have:

- one IN-1 input for the data to write.
- one IN-8 input for the address to read/write.
- one IN-1 input for the write enable signal.
- one OUT-1 output for the data read.

It's basically the same as before, but with 256 cells instead of 16.

MEM 256x1

Let's scale it up to 256 bits!

It should have:

- one IN-1 input for the data to write.
- one IN-8 input for the address to read/write.
- one IN-1 input for the write enable signal.
- one OUT-1 output for the data read.

It's basically the same as before, but with 256 cells instead of 16.

In case you didn't notice, $256 = 16 \times 16 \dots$

hint →

1.

1. You will make a grid of 4 rows and 4 columns of MEM_16x1.
- 2.

1. You will make a grid of 4 rows and 4 columns of MEM_16x1.
2. Some of the address byte will be decoded at this step, and some passed on to the MEM_16x1.
- 3.

1. You will make a grid of 4 rows and 4 columns of MEM_16x1.
2. Some of the address byte will be decoded at this step, and some passed on to the MEM_16x1.
3. The outputs of all MEM_16x1 should be linked to a bus, so that only the active one outputs to the bus.

solution →

Here is what your 256x1 RAM should look like. Save it as
MEM_256x1.

Byte RAM

Finally, let's make the byte version of our RAM!

Byte RAM

Finally, let's make the byte version of our RAM!

It should have:

Byte RAM

Finally, let's make the byte version of our RAM!

It should have:

- one IN-8 input for the data to write.
- one IN-8 input for the address to read/write.
- one IN-1 input for the write enable signal.
- one OUT-8 output for the data read.

Byte RAM

Finally, let's make the byte version of our RAM!

It should have:

- one IN-8 input for the data to write.
- one IN-8 input for the address to read/write.
- one IN-1 input for the write enable signal.
- one OUT-8 output for the data read.

It's basically 8 of the MEM_256x1 we just made, one for each bit of the byte.

Byte RAM

Finally, let's make the byte version of our RAM!

It should have:

- one IN-8 input for the data to write.
- one IN-8 input for the address to read/write.
- one IN-1 input for the write enable signal.
- one OUT-8 output for the data read.

It's basically 8 of the MEM_256x1 we just made, one for each bit of the byte.

Don't forget to link all the inputs and outputs correctly!

solution →

Here is what your byte RAM should look like. Save it as RAM.

ROM

Here, our CPU will differ from Von Neumann architecture computers.

ROM

Here, our CPU will differ from Von Neumann architecture computers.

Instead of having a single memory for both instructions and data, we will have a separate ROM for the instructions.

ROM

Here, our CPU will differ from Von Neumann architecture computers.

Instead of having a single memory for both instructions and data, we will have a separate ROM for the instructions.

We will use the ROM component from the library. It's basically the same as the RAM we just made, but without the write capability.

ROM

Here, our CPU will differ from Von Neumann architecture computers.

Instead of having a single memory for both instructions and data, we will have a separate ROM for the instructions.

We will use the ROM component from the library. It's basically the same as the RAM we just made, but without the write capability.

We will then later write our program directly into the ROM using the built-in editor.

The program counter

Another key component of the memory system is the program counter (PC).

The program counter

Another key component of the memory system is the program counter (PC).

The PC is a register that holds the address of the next instruction to execute.

The program counter

Another key component of the memory system is the program counter (PC).

The PC is a register that holds the address of the next instruction to execute.

After fetching each instruction, the PC is incremented to point to the next instruction in memory.

The program counter

Another key component of the memory system is the program counter (PC).

The PC is a register that holds the address of the next instruction to execute.

After fetching each instruction, the PC is incremented to point to the next instruction in memory.

However, some instructions (like JUMP and JZ) can modify the PC to point to a different address.

The program counter

Another key component of the memory system is the program counter (PC).

The PC is a register that holds the address of the next instruction to execute.

After fetching each instruction, the PC is incremented to point to the next instruction in memory.

However, some instructions (like JUMP and JZ) can modify the PC to point to a different address.

This is our next component to make!

Program Counter

The PC should have:

Program Counter

The PC should have:

- one In-1 control signal named "jump".
- one IN-8 input for the jump address.
- one IN-1 input name "inc/save", that either increments the PC or saves the jump address depending on the control signal.
- one OUT-8 output for the current address in the PC.

Program Counter

The PC should have:

- one IN-1 control signal named "jump".
- one IN-8 input for the jump address.
- one IN-1 input name "inc/save", that either increments the PC or saves the jump address depending on the control signal.
- one OUT-8 output for the current address in the PC.

When "inc/save" turns on, the PC should either increment its value by one (if jump is off), or save the jump address (if jump is on).

hint →

1.

1. Use an 8 bit register to store the current address.
- 2.

1. Use an 8 bit register to store the current address.
2. Use an adder to increment the current address by one.
- 3.

1. Use an 8 bit register to store the current address.
2. Use an adder to increment the current address by one.
3. Use a switch to chose between the incremented value and the jump address depending on the jump signal. We did a one bit switch before, so make an 8 bit switch and save it as 8bit_MUX.
- 4.

1. Use an 8 bit register to store the current address.
2. Use an adder to increment the current address by one.
3. Use a switch to choose between the incremented value and the jump address depending on the jump signal. We did a one bit switch before, so make an 8 bit switch and save it as 8bit_MUX.
4. Link the output of the switch to the input of the register, and use "inc/save" as the clock signal of the register.

solution →

Here is what your Program Counter should look like. Save it as PC.

Memory done!

Good jobs on completing your Memory system! In this chapter, you made both the RAM and the Program Counter, two key components of your future CPU!

Memory done!

Good jobs on completing your Memory system! In this chapter, you made both the RAM and the Program Counter, two key components of your future CPU!

Digital logic sim also provides us with a ROM component that we will use for our instruction memory.

Memory done!

Good jobs on completing your Memory system! In this chapter, you made both the RAM and the Program Counter, two key components of your future CPU!

Digital logic sim also provides us with a ROM component that we will use for our instruction memory.

That means we just added 3 top level components to our future CPU!

Memory done!

Good jobs on completing your Memory system! In this chapter, you made both the RAM and the Program Counter, two key components of your future CPU!

Digital logic sim also provides us with a ROM component that we will use for our instruction memory.

That means we just added 3 top level components to our future CPU!

Take some time to **organize your chips** in the library, and **check with your peers and teacher** before starting the next section!

Outline

- 1 Project Setup
- 2 Basic Logic gates
- 3 Arithmetic
- 4 Memory 1
- 5 Control Unit part 1
- 6 Buses
- 7 Memory 2
- 8 Building up the CPU
- 9 Control Unit part 2
- 10 Testing the CPU and writing programs

Adding control signals

Now that we have all top level components made, let's make the highest level component in our project. Yes, this is time to create the CPU itself!

Adding control signals

Now that we have all top level components made, let's make the highest level component in our project. Yes, this is time to create the CPU itself!

It's not going to be complete immediately, but we can already start wiring the components we have made so far.

Adding control signals

Now that we have all top level components made, let's make the highest level component in our project. Yes, this is time to create the CPU itself!

It's not going to be complete immediately, but we can already start wiring the components we have made so far.

We're first going to setup the components, without wiring anything yet.

Adding control signals

Now that we have all top level components made, let's make the highest level component in our project. Yes, this is time to create the CPU itself!

It's not going to be complete immediately, but we can already start wiring the components we have made so far.

We're first going to setup the components, without wiring anything yet.

You can find the list of components to add on the next slide.

CPU draft

- our Program Counter (PC).
- our ROM for instructions.
- an 8 bit register for the instruction (right click on it and rename it "CIR" for "Current Instruction Register").
- on the same level as the CIR, another 8 bit register for the operand (right click on it and rename it "OPERAND").
- our CU.
- our ALU.
- an 8 bit register for the accumulator (right click on it and rename it "ACC").
- on the same level as the ACC, a one bit register for the zero flag (right click on it and rename it "ZFLAG").
- our RAM.

Don't worry about wiring anything yet, just place the components for now.

Here is what your CPU draft should look like. Save it as CPU.

Let's wire things up! (part 1/3)

Now, let's start wiring things up!

Let's wire things up! (part 1/3)

Now, let's start wiring things up!

First, wire the PC output to the ROM address input.

Let's wire things up! (part 1/3)

Now, let's start wiring things up!

First, wire the PC output to the ROM address input.

Then, wire the top ROM output to the CIR input, and the bottom ROM output to the OPERAND input.

Let's wire things up! (part 1/3)

Now, let's start wiring things up!

First, wire the PC output to the ROM address input.

Then, wire the top ROM output to the CIR input, and the bottom ROM output to the OPERAND input.

Finally, wire the CIR output to the CU instruction input.

Here is what the left-most part of your CPU should look like so far.

Let's wire things up! (part 2/3)

Now, wire the ACC output to the ALU input A.

Let's wire things up! (part 2/3)

Now, wire the ACC output to the ALU input A.

Then, wire the ALU output to the ACC input, and the ALU zero flag output to the ZFLAG input.

Finally, wire the ZFLAG output to the CU zero flag input.

Here is what the right-most part of your CPU should look like so far.

Let's wire things up! (part 3/3)

Now for our registers enable signals.

Let's wire things up! (part 3/3)

Now for our registers enable signals.

The model we made was actually more complicated than needed, since our registers will always be enabled. So just wire all the enable signals of the CIR, OPERAND, ACC, and ZFLAG registers to 1.

Let's wire things up! (part 3/3)

Now for our registers enable signals.

The model we made was actually more complicated than needed, since our registers will always be enabled. So just wire all the enable signals of the CIR, OPERAND, ACC, and ZFLAG registers to 1.

Our next step will be to add buses to connect the components together.

Data bus

First, let's make the data bus.

Data bus

First, let's make the data bus.

For now, the data bus receives data from the RAM, the ALU, and the ACC.

Data bus

First, let's make the data bus.

For now, the data bus receives data from the RAM, the ALU, and the ACC.

Remember, never connect outputs directly to a bus, they should go through tri-state buffers!

Data bus

First, let's make the data bus.

For now, the data bus receives data from the RAM, the ALU, and the ACC.

Remember, never connect outputs directly to a bus, they should go through tri-state buffers!

The second operand input of the ALU receives data from the data bus.

Data bus

First, let's make the data bus.

For now, the data bus receives data from the RAM, the ALU, and the ACC.

Remember, never connect outputs directly to a bus, they should go through tri-state buffers!

The second operand input of the ALU receives data from the data bus.

Finally, the RAM data input receives data from the data bus.

Here is what the data bus part of your CPU should look like so far.

Address bus

First, let's make the address bus.

Address bus

First, let's make the address bus.

The address bus receives address only from the OPERAND register, so no need for tri-state buffers.

Address bus

First, let's make the address bus.

The address bus receives address only from the OPERAND register, so no need for tri-state buffers.

The value on the address bus is then sent to both the RAM and the PC (for jumps).

Here is what the address bus part of your CPU should look like so far.

Immediate instructions

For immediate instructions (IMM), the second part of the instruction is not an address, but a direct value to load into the ACC.

Immediate instructions

For immediate instructions (IMM), the second part of the instruction is not an address, but a direct value to load into the ACC.

For that, we need to allow the address bus to output into the data bus when executing an IMM instruction. This in connecting into a bus, so remember to use tri-state buffers!

Immediate instructions

For immediate instructions (IMM), the second part of the instruction is not an address, but a direct value to load into the ACC.

For that, we need to allow the address bus to output into the data bus when executing an IMM instruction. This involves connecting into a bus, so remember to use tri-state buffers!

Lastly, we need to wire the data bus directly to the input of the ACC register, but this input is already used by the ALU.

Immediate instructions

For immediate instructions (IMM), the second part of the instruction is not an address, but a direct value to load into the ACC.

For that, we need to allow the address bus to output into the data bus when executing an IMM instruction. This involves connecting into a bus, so remember to use tri-state buffers!

Lastly, we need to wire the data bus directly to the input of the ACC register, but this input is already used by the ALU.

For that, we will need a MUX to choose between the ALU output and the data bus output depending on whether we're executing an IMM instruction or not.

Here is what the immediate instruction setup of your CPU should look like so far.

CPU draft done!

Good jobs on completing your CPU draft! This is the top level component of your project!

CPU draft done!

Good jobs on completing your CPU draft! This is the top level component of your project!

Next, we will come back to the Control Unit to add the control signals needed to make everything work together.

CPU draft done!

Good jobs on completing your CPU draft! This is the top level component of your project!

Next, we will come back to the Control Unit to add the control signals needed to make everything work together.

Take some time to **organize your chips** in the library, and **check with your peers and teacher** before starting the next section!

Outline

- 1 Project Setup
- 2 Basic Logic gates
- 3 Arithmetic
- 4 Memory 1
- 5 Control Unit part 1
- 6 Buses
- 7 Memory 2
- 8 Building up the CPU
- 9 Control Unit part 2
- 10 Testing the CPU and writing programs

Making a list of control signals

Now that we have the CPU draft made, we can start thinking about which control signals we need to add to the Control Unit.

Making a list of control signals

Now that we have the CPU draft made, we can start thinking about which control signals we need to add to the Control Unit.

Go back to the CPU draft, and look at all the components we have added.

Making a list of control signals

Now that we have the CPU draft made, we can start thinking about which control signals we need to add to the Control Unit.

Go back to the CPU draft, and look at all the components we have added.

For each component, think about which control signals are needed to make it work properly.

Making a list of control signals

Now that we have the CPU draft made, we can start thinking about which control signals we need to add to the Control Unit.

Go back to the CPU draft, and look at all the components we have added.

For each component, think about which control signals are needed to make it work properly.

For example, registers need a clock signal to tell them when to update their value.

Making a list of control signals

Now that we have the CPU draft made, we can start thinking about which control signals we need to add to the Control Unit.

Go back to the CPU draft, and look at all the components we have added.

For each component, think about which control signals are needed to make it work properly.

For example, registers need a clock signal to tell them when to update their value.

Write down all the control signals you think are needed on a piece of paper. That's basically unconnected pins on each component.

hint →

1.

1. Start with the registers, they usually need an clock signal.
- 2.

1. Start with the registers, they usually need an clock signal.
2. Then, look at the tri-state buffers on the buses, they need enable signals.
- 3.

1. Start with the registers, they usually need an clock signal.
2. Then, look at the tri-state buffers on the buses, they need enable signals.
3. MUXes also need select signals.
- 4.

1. Start with the registers, they usually need an clock signal.
2. Then, look at the tri-state buffers on the buses, they need enable signals.
3. MUXes also need select signals.
4. Finally, the ALU needs operation signals to chose which operation to perform.
- 5.

1. Start with the registers, they usually need an clock signal.
2. Then, look at the tri-state buffers on the buses, they need enable signals.
3. MUXes also need select signals.
4. Finally, the ALU needs operation signals to chose which operation to perform.
5. You should end up with 12, but two can be combined to bring it down to 11 signals.

solution →

Here is the list of control signals you should have ended up with:

- PC_jump
- PC_inc/save
- CIR_clock and OPERAND_clock combined
- ACC_clock
- ZFLAG_clock
- RAM_write_enable
- ALU_SUB
- ADDRESS_BUS_TO_DATA_BUS_TRI_STATE
- RAM_TO_DATA_BUS_TRI_STATE
- ACC_TO_DATA_BUS_TRI_STATE
- DATA_BUS_TO_ACC_MUX

Adding control signals to the CU

Now that we have the list of control signals, let's add them as outputs to the CU.

Adding control signals to the CU

Now that we have the list of control signals, let's add them as outputs to the CU.

Go back to the CU draft, and add one OUT-1 for each control signal.

Adding control signals to the CU

Now that we have the list of control signals, let's add them as outputs to the CU.

Go back to the CU draft, and add one OUT-1 for each control signal.

Don't forget to rename them correctly!

Adding control signals to the CU

Now that we have the list of control signals, let's add them as outputs to the CU.

Go back to the CU draft, and add one OUT-1 for each control signal.

Don't forget to rename them correctly!

Once you're done, save the updated CU.

Wiring the control signals

Now, go back to the CPU draft, and wire each control signal output from the CU to the corresponding input on each component.

Wiring the control signals

Now, go back to the CPU draft, and wire each control signal output from the CU to the corresponding input on each component.

Be careful to not mix up the signals!

Wiring the control signals

Now, go back to the CPU draft, and wire each control signal output from the CU to the corresponding input on each component.

Be careful to not mix up the signals!

Once you're done, save the updated CPU.

Fetch-Increment-Decode-Execute cycle

Let's me give you more detail about how our Fetch-Increment-Decode-Execute (FDE) cycle works.

Fetch-Increment-Decode-Execute cycle

Let's me give you more detail about how our Fetch-Increment-Decode-Execute (FDE) cycle works.

During the **Fetch** step, the value in ROM are saved into the CIR and OPERAND registers.

Fetch-Increment-Decode-Execute cycle

Let's me give you more detail about how our Fetch-Increment-Decode-Execute (FDE) cycle works.

During the **Fetch** step, the value in ROM are saved into the CIR and OPERAND registers.

During the **Increment** step, the PC is incremented to point to the next instruction.

Fetch-Increment-Decode-Execute cycle

Let's me give you more detail about how our Fetch-Increment-Decode-Execute (FDE) cycle works.

During the **Fetch** step, the value in ROM are saved into the CIR and OPERAND registers.

During the **Increment** step, the PC is incremented to point to the next instruction.

During the **Decode** step, the CU sends control signal to get the right data into the right places depending on the instruction opcode. These signals stay on for the next phase.

Fetch-Increment-Decode-Execute cycle

Let's me give you more detail about how our Fetch-Increment-Decode-Execute (FDE) cycle works.

During the **Fetch** step, the value in ROM are saved into the CIR and OPERAND registers.

During the **Increment** step, the PC is incremented to point to the next instruction.

During the **Decode** step, the CU sends control signal to get the right data into the right places depending on the instruction opcode. These signals stay on for the next phase.

Finally, during the **Execute** step, the appropriate register or RAM save the result of the operation.

Fetch control signals

This step doesn't depend of the instruction opcode, so the CU should always output the same control signals during this step.

Fetch control signals

This step doesn't depend of the instruction opcode, so the CU should always output the same control signals during this step.

Try to figure out which control signal(s) should be on during the Fetch step.

Fetch control signals

This step doesn't depend of the instruction opcode, so the CU should always output the same control signals during this step.

Try to figure out which control signal(s) should be on during the Fetch step.

Then, wire them up in the CU so that they are on during the Fetch step of the FDE controller.

hint →

1.

1. During Fetch, we want to load the instruction from ROM into the CIR and OPERAND registers.
- 2.

1. During Fetch, we want to load the instruction from ROM into the CIR and OPERAND registers.
2. This means we need to enable the clock signal of both registers.
- 3.

1. During Fetch, we want to load the instruction from ROM into the CIR and OPERAND registers.
2. This means we need to enable the clock signal of both registers.
3. The opcode doesn't matter, so we shouldn't connect anything coming out of the instruction decoder for now.
- 4.

1. During Fetch, we want to load the instruction from ROM into the CIR and OPERAND registers.
2. This means we need to enable the clock signal of both registers.
3. The opcode doesn't matter, so we shouldn't connect anything coming out of the instruction decoder for now.
4. Nothing else happens during Fetch.

solution →

Only the CIR_clock and OPERAND_clock signals should be on during Fetch.

Only the CIR_clock and OPERAND_clock signals should be on during Fetch.

So here is what the Fetch part of your CU should look like so far.

Increment control signals

This step also doesn't depend of the instruction opcode, so the CU should always output the same control signals during this step.

Increment control signals

This step also doesn't depend of the instruction opcode, so the CU should always output the same control signals during this step.

Try to figure out which control signal(s) should be on during the Increment step.

Increment control signals

This step also doesn't depend of the instruction opcode, so the CU should always output the same control signals during this step.

Try to figure out which control signal(s) should be on during the Increment step.

Then, wire them up in the CU so that they are on during the Increment step of the FDE controller.

hint →

1.

1. During Increment, we want to increment the PC to point to the next instruction.
- 2.

1. During Increment, we want to increment the PC to point to the next instruction.
2. Nothing else happens during Increment.

solution →

Only the PC_inc/save signal should be on during Increment.

Only the PC_inc/save signal should be on during Increment.

So here is what the Increment part of your CU should look like so far.

Decode and Execute control signals

Looking at the description of the decode and the execute steps from earlier, signals need to be on either for both steps, or just execute.

Decode and Execute control signals

Looking at the description of the decode and the execute steps from earlier, signals need to be on either for both steps, or just execute.

So go inside the FDE controller and tweek it up a little so that the outputs are:

- Execute only.
- Both Decode and Execute.
- Increment only.
- Fetch only.

Here is what your updated FDE controller should look like.

From now on, we will wire each instruction one by one.

From now on, we will wire each instruction one by one.

Remember that for each instruction, so signals need to turn on during Decode and Execute, while others only during Execute.

From now on, we will wire each instruction one by one.

Remember that for each instruction, so signals need to turn on during Decode and Execute, while others only during Execute.

Usually, the signals that move data around (like tri-state buffer enables and MUX selects) will be on during **Decode and Execute**.

Signals that store data (like register clocks and RAM write enable) will only be on during **Execute**.

From now on, we will wire each instruction one by one.

Remember that for each instruction, so signals need to turn on during Decode and Execute, while others only during Execute.

Usually, the signals that move data around (like tri-state buffer enables and MUX selects) will be on during **Decode and Execute**.

Signals that store data (like register clocks and RAM write enable) will only be on during **Execute**.

Since one control signal can be on for multiple instructions, we will use OR gates to combine all cases where a signal should be on. So lets prepare some OR gates!

Instruction recap

Name	Opcode	Operand	Description
LOAD	000	Address	Load a value from RAM into the accumulator
STORE	001	Address	Store the value of the accumulator into RAM
ADD	010	Address	Add a value in RAM to the accumulator
SUB	011	Address	Subtract a value in RAM from the accumulator
JUMP	100	Value	Jump to a specific instruction address
JZ	101	Value	Jump to a specific instruction address if zero flag is set
IMM	110	Value	Load a direct value into the accumulator

Preparing OR gates

For each control signal, try to make a list of all the instructions that require this signal to be on.

Preparing OR gates

For each control signal, try to make a list of all the instructions that require this signal to be on.

Then, depending on how many instructions need this signal, add an OR gate with the right number of inputs inside the CU.

Preparing OR gates

For each control signal, try to make a list of all the instructions that require this signal to be on.

Then, depending on how many instructions need this signal, add an OR gate with the right number of inputs inside the CU.

solution →

Here is the number of inputs each signal OR gate should have:

Control Signal	Number of inputs
PC_jump	2
PC_inc/save	3
CIR_clock and OPERAND_clock	1
ACC_clock	4
ZFLAG_clock	2
RAM_write_enable	1
ALU_SUB	1
ADDRESS_BUS_TO_DATA _BUS_TRI_STATE	1
RAM_TO_DATA_BUS_TRI_STATE	3
ACC_TO_DATA_BUS_TRI_STATE	1
DATA_BUS_TO_ACC_MUX	2

Here is what your CU should look like with all the OR gates added.

Wiring 000 - LOAD

Finally, let's wire the control signals for each instruction!

Wiring 000 - LOAD

Finally, let's wire the control signals for each instruction!

Starting with LOAD (000), try to figure out which control signals should be on during Decode and Execute.

Wiring 000 - LOAD

Finally, let's wire the control signals for each instruction!

Starting with LOAD (000), try to figure out which control signals should be on during Decode and Execute.

000 - LOAD: Load a value from RAM into the accumulator.

Wiring 000 - LOAD

Finally, let's wire the control signals for each instruction!

Starting with LOAD (000), try to figure out which control signals should be on during Decode and Execute.

000 - LOAD: Load a value from RAM into the accumulator.

Then, wire the corresponding instruction decoder output to the right OR gates inputs.

hint →

During LOAD:

- During Decode and Execute:
 - RAM_TO_DATA_BUS_TRI_STATE
 - DATA_BUS_TO_ACC_MUX
- During Execute:
 - ACC_clock

solution →

Here is what your CU should look like after wiring LOAD.

001 - STORE

Now, try to figure out which control signals should be on during Decode and Execute for STORE (001).

001 - STORE

Now, try to figure out which control signals should be on during Decode and Execute for STORE (001).

001 - STORE: Store the value of the accumulator into RAM.

001 - STORE

Now, try to figure out which control signals should be on during Decode and Execute for STORE (001).

001 - STORE: Store the value of the accumulator into RAM.

Then, wire the corresponding instruction decoder output to the right OR gates inputs.

hint →

During STORE:

- During Decode and Execute:
 - ACC_TO_DATA_BUS_TRI_STATE
- During Execute:
 - RAM_write_enable

solution →

Here is what your CU should look like after wiring STORE.

010 - ADD

Now, try to figure out which control signals should be on during Decode and Execute for ADD (010).

010 - ADD

Now, try to figure out which control signals should be on during Decode and Execute for ADD (010).

010 - ADD: Add a value in RAM to the accumulator.

010 - ADD

Now, try to figure out which control signals should be on during Decode and Execute for ADD (010).

010 - ADD: Add a value in RAM to the accumulator.

Then, wire the corresponding instruction decoder output to the right OR gates inputs.

hint →

During ADD:

- During Decode and Execute:
 - RAM_TO_DATA_BUS_TRI_STATE
- During Execute:
 - ACC_clock

solution →

Here is what your CU should look like after wiring ADD.

011 - SUB

Now, try to figure out which control signals should be on during Decode and Execute for SUB (011).

011 - SUB

Now, try to figure out which control signals should be on during Decode and Execute for SUB (011).

011 - SUB: Subtract a value in RAM from the accumulator.

011 - SUB

Now, try to figure out which control signals should be on during Decode and Execute for SUB (011).

011 - SUB: Subtract a value in RAM from the accumulator.

Then, wire the corresponding instruction decoder output to the right OR gates inputs.

hint →

During SUB:

- During Decode and Execute:
 - RAM_TO_DATA_BUS_TRI_STATE
 - ALU_SUB
- During Execute:
 - ACC_clock

solution →

Here is what your CU should look like after wiring SUB.

100 - JUMP

Now, try to figure out which control signals should be on during Decode and Execute for JUMP (100).

100 - JUMP

Now, try to figure out which control signals should be on during Decode and Execute for JUMP (100).

100 - JUMP: Jump to a specific instruction address.

100 - JUMP

Now, try to figure out which control signals should be on during Decode and Execute for JUMP (100).

100 - JUMP: Jump to a specific instruction address.

Then, wire the corresponding instruction decoder output to the right OR gates inputs.

hint →

During JUMP:

- During Decode and Execute:
 - ADDRESS_BUS_TO_DATA_BUS_TRI_STATE
 - PC_jump
- During Execute:
 - PC_inc/save

solution →

Here is what your CU should look like after wiring JUMP.

101 - JZ

Now, try to figure out which control signals should be on during Decode and Execute for JZ (101).

101 - JZ

Now, try to figure out which control signals should be on during Decode and Execute for JZ (101).

101 - JZ: Jump to a specific instruction address if zero flag is set.

101 - JZ

Now, try to figure out which control signals should be on during Decode and Execute for JZ (101).

101 - JZ: Jump to a specific instruction address if zero flag is set.

Then, wire the corresponding instruction decoder output to the right OR gates inputs.

hint →

During JZ:

- During Decode and Execute:
 - ADDRESS_BUS_TO_DATA_BUS_TRI_STATE
 - PC_jump
- During Execute:
 - PC_inc/save but **only** if ZFLAG is on

solution →

Here is what your CU should look like after wiring JZ.

110 - IMM

Now, try to figure out which control signals should be on during Decode and Execute for IMM (110).

110 - IMM

Now, try to figure out which control signals should be on during Decode and Execute for IMM (110).

110 - IMM: Load a direct value into the accumulator.

110 - IMM

Now, try to figure out which control signals should be on during Decode and Execute for IMM (110).

110 - IMM: Load a direct value into the accumulator.

Then, wire the corresponding instruction decoder output to the right OR gates inputs.

hint →

During IMM:

- During Decode and Execute:
 - ADDRESS_BUS_TO_DATA_BUS_TRI_STATE
 - DATA_BUS_TO_ACC_MUX
- During Execute:
 - ACC_clock

solution →

Here is what your CU should look like after wiring IMM.

CU... no wait... CPU done!

Congratulations on completing your Control Unit and CPU!

CU... no wait... CPU done!

Congratulations on completing your Control Unit and CPU!

You have now built a simple 8-bit CPU from the ground up, including its ALU, Control Unit, Memory system, and Program Counter!

CU... no wait... CPU done!

Congratulations on completing your Control Unit and CPU!

You have now built a simple 8-bit CPU from the ground up, including its ALU, Control Unit, Memory system, and Program Counter!

Take some time to **organize your chips** in the library, and **check with your peers and teacher** before starting the next section!

Outline

- 1 Project Setup
- 2 Basic Logic gates
- 3 Arithmetic
- 4 Memory 1
- 5 Control Unit part 1
- 6 Buses
- 7 Memory 2
- 8 Building up the CPU
- 9 Control Unit part 2
- 10 Testing the CPU and writing programs

Writting programs

Now that the CPU is done, it's time to write some programs to test it!

Writting programs

Now that the CPU is done, it's time to write some programs to test it!

First, open the ROM component inside the CPU by right clicking on it.

Writting programs

Now that the CPU is done, it's time to write some programs to test it!

First, open the ROM component inside the CPU by right clicking on it.

Put the number format to hexadecimal.

Writting programs

Now that the CPU is done, it's time to write some programs to test it!

First, open the ROM component inside the CPU by right clicking on it.

Put the number format to hexadecimal.

Then, you can write your program directly into the ROM as machine code!

Simple program example

Here is a simple program that adds two numbers together and stores the result in memory.

```
a = 5  
b = 10  
c = a + b
```

To turn this into machine code, we first need to change the variable names into memory addresses, and the number in hexadecimal. In order not to get confused, we'll write them like this: [address] = value

```
[00] = 5  
[01] = 0A  
[02] = [00] + [01]
```

Simple program example (cont.)

Now, we can turn this into instructions from our instruction set (in hexadecimal):

```
IMM    05    ; Load immediate value 5 into ACC
STORE  00    ; Store ACC value into RAM address 00
IMM    0A    ; Load immediate value 10 into ACC
STORE  01    ; Store ACC value into RAM address 01
LOAD   00    ; Load value from RAM address 00 into ACC
ADD    01    ; Add value from RAM address 01 to ACC
STORE  02    ; Store ACC value into RAM address 02
```

Simple program example (cont.)

Finally, we can convert this into machine code using the opcodes:

```
06 05 ; IMM 5
01 00 ; STORE 00
06 0A ; IMM 10
01 01 ; STORE 01
00 00 ; LOAD 00
02 01 ; ADD 01
01 02 ; STORE 02
```

Testing the CPU

Put the simulation on pause with Crtl + Space or Command + Space.

Testing the CPU

Put the simulation on pause with Crtl + Space or Command + Space.

Write this small program into the ROM component of your CPU.

Testing the CPU

Put the simulation on pause with Crtl + Space or Command + Space.

Write this small program into the ROM component of your CPU.

Then, put the CU to 0 using an IN-1 and an IN-8 connected in the right spot (left as exercise).

Testing the CPU

Put the simulation on pause with Crtl + Space or Command + Space.

Write this small program into the ROM component of your CPU.

Then, put the CU to 0 using an IN-1 and an IN-8 connected in the right spot (left as exercise).

Finally, start the simulation and watch your CPU execute the program step by step!

Testing the CPU

Put the simulation on pause with Crtl + Space or Command + Space.

Write this small program into the ROM component of your CPU.

Then, put the CU to 0 using an IN-1 and an IN-8 connected in the right spot (left as exercise).

Finally, start the simulation and watch your CPU execute the program step by step!

After the program is done, check the RAM contents to see if the result (0F in hexadecimal) is stored at address 02.

Congratulations!

Congratulations on completing your simple 8-bit CPU project!

Congratulations!

Congratulations on completing your simple 8-bit CPU project!

You have built a functional CPU from the ground up, including its ALU, Control Unit, Memory system, and Program Counter!

Congratulations!

Congratulations on completing your simple 8-bit CPU project!

You have built a functional CPU from the ground up, including its ALU, Control Unit, Memory system, and Program Counter!

You have also written and executed machine code programs on your CPU!

Congratulations!

Congratulations on completing your simple 8-bit CPU project!

You have built a functional CPU from the ground up, including its ALU, Control Unit, Memory system, and Program Counter!

You have also written and executed machine code programs on your CPU!

Take some time to reflect on what you have learned and how you can apply this knowledge in future projects.

Congratulations!

Congratulations on completing your simple 8-bit CPU project!

You have built a functional CPU from the ground up, including its ALU, Control Unit, Memory system, and Program Counter!

You have also written and executed machine code programs on your CPU!

Take some time to reflect on what you have learned and how you can apply this knowledge in future projects.

Great job, and keep up the good work!

For the bravest of all

If you want to practice assembly programming further, try to think about how to make the following structures and function in assembly language using our instruction set:

For the bravest of all

If you want to practice assembly programming further, try to think about how to make the following structures and function in assembly language using our instruction set:

- Conditional statements (if-else).
- Loops (for, while).
- multiplications.