CSARCH2 Case Study 2 Technical Doucment

T1 AY2025-2026

This document serves as the technical documentation of the Minecraft Computer in order to fully understand the entire template. We will cover every part from the basics of redstone, up to each component of the CPU.

1 Redstone Basics

Redstone is a form of wiring system of the game Minecraft where it can emulate real life wiring systems. This also includes computer architecture systems since these also compose electrical wiring systems structured to emulate logic.
There are multiple components that are used in redstone wiring but we will only be using certain components since other components can be buggy or are not used to create computer systems.

1.1 Redstone Dust

The redstone dust is a primary component of redstone that replicates wires. These dusts can be connected to each other to send signals from one point to another. Signals also have strength as the further the signal travels from the source, the lower the strength. The strength of the signal can go from 15(Directly from source) to 0(Off). Any signal from 15 to 1 will power components connected to these components, since it is in the "On" state.

1.2 Redstone Repeater

The redstone repeater serves multiple purposes.

- Buffer Gate
  - The repeater sends the input signal to the output signal, serving as a one directional gate. This allows the signal to go from input to output but not the other way around.
- Signal Extender
  - The repeater will take any signal level from 15 to 1 and output it as 15. This allows the signal to extend, travelling further.
- Delay
  - The repeater by default will delay the signal by 1 tick, but this can be configured from 1 to 4 ticks of delay. This delay allows for circuits to be activated on certain times in certain orders based on the arrival of signals.

- Lock
  - The repeater can also be locked to save a signal state, where the current state of the repeater will not change regardless of input. This replicates the behavior similar to the delay flip-flop, when state will only be allowed to change when turned off. This would require a redstone repeater to directly send a signal from the side of the repeater for it to lock.

1.3 Redstone Comparator

The comparator, similar to the repeater, also serves multiple purposes.

- Compare
  - Comparison mode sets the comparator to only output a signal when the signal from the back is greater than or equal to the signal from the side. An example to this is when a signal from the back is 15 and the signal from the side is 14, the output will be 15, since 15 is greater than or equal to 14. Another example would be when a signal from the back is 14 and the signal from the side is 15, the output will be 0 since 14 is not greater than or equal to 15.
- Subtraction
  - Subtraction mode sets the comparator to output the subtraction of the signals from the back and the side. This follows the back - side = output rule. Note that when the result is negative, it will only output 0. An example to this is when a signal from the back is 15 and the signal from the side is 14, the output will be 1 since 15-14 is 1.
- Close
  - With subtraction mode, you can send a signal from a repeater to the side of the comparator to have the input from the back be subtracted with 15 by default. With this, the signal will always be 0 since 15 as the max signal will be cancelled by -15, and any value below 15 will output 0 since the result would be negative.

1.4 Redstone Torch

The redstone torch also serves multiple purposes

- Power Source
  - The redstone torch, by default when connected to any component, will power those components. Connecting this to wires or repeaters or any other component will activate them.
- Negation
  - When a power source is directly sending a signal to a block that the redstone torch is directly connected to, this will turn the torch off. Since when there is no power going into the torch, the torch will activate, and when there is power going into the torch, the torch will deactivate, this essentially imitates the behavior of a not gate.

1.5 Redstone Lamp

The redstone lamp serves as an indicator when a signal is active or not. It will only turn on when the signal goes into the redstone lamp. This imitates the behavior of an LED where it lights up when it is powered. We can also treat signals as 0s and 1s where 0 means off and 1 means on, allowing redstone lamps to present binary values.

1.6 Lever

The lever is a source of power where it can be toggled on and off. When on, it will power up components connected to it until turned off.

1.7 Button

The button is a source of power where it will provide power for a certain duration until it turns off by itself. Unlike the level, this cannot be kept open permanently.

1.8 Solid Block

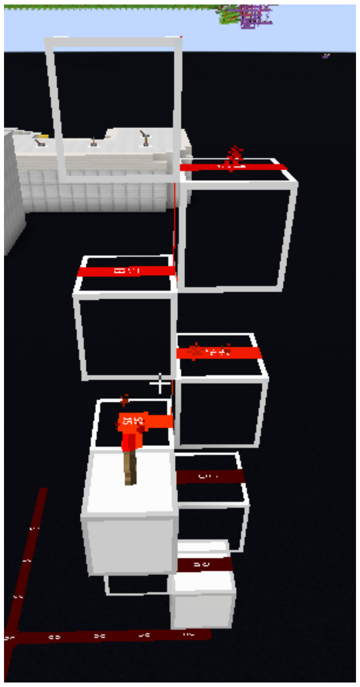
The solid block can pass signal through itself though not every full solid block can do this, blocks like glass cannot send signal through it unlike the solid block. When sending a signal from a wire into a block, this can be received by a torch, repeater, comparator, or redstone lamp. When receiving signal from a solid block, only a level, button, repeater, or comparator can send signal through that a redstone wire can receive.

1.9 Target Block

The target block serves as a redirection of wire, since when wire travels, it will travel by the direction where it is connected. It will only redirect when beside another wire, repeater, comparator, torch, lever, or button. This means that redstone wires will not connect to other blocks like the redstone lamp, solid blocks, or glass blocks. Using the target block serves a similar purpose to a solid block but it can redirect dust into itself.

1.10 Glass Blocks

A glass block serves as a transparent block that allows signals to go through diagonally. Glass towers are then introduced, where this allows redstone signals to go up but not down. This is a special mechanism that only redstone minecraft has.

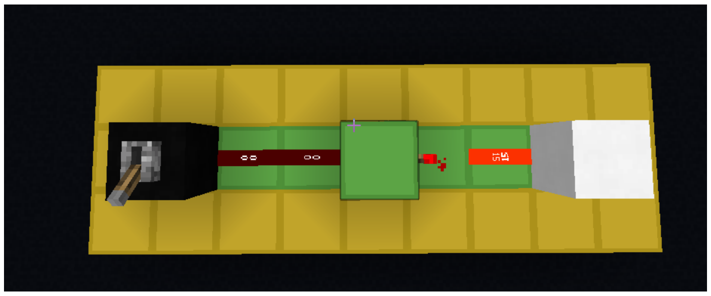


2 Logic Gates

With the combination of certain components, you can replicate the behavior of logic gates.

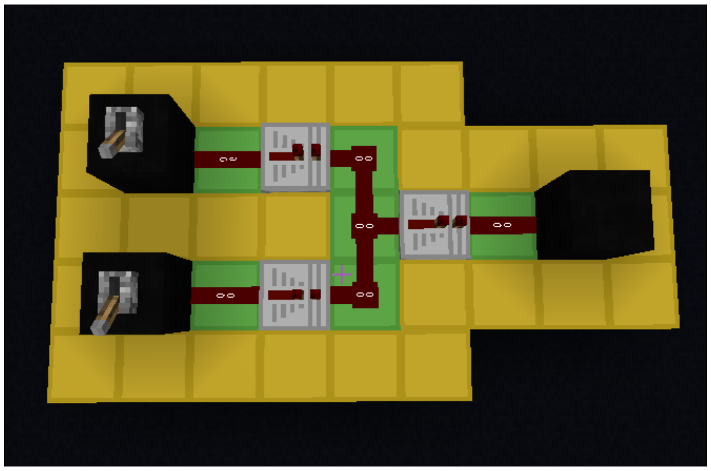
2.1 Not Gate

The not gate is made by sending redstone wire from the input into a block that is connected to a redstone torch, then the redstone torch is connected to the output.



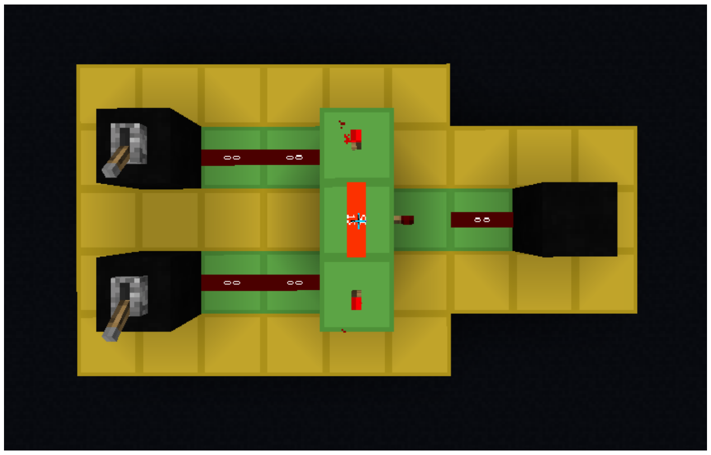
2.2 Or Gate

The or gate is made by connecting 2 sources of signal into one, so regardless of which signal source is active, the output will be active.



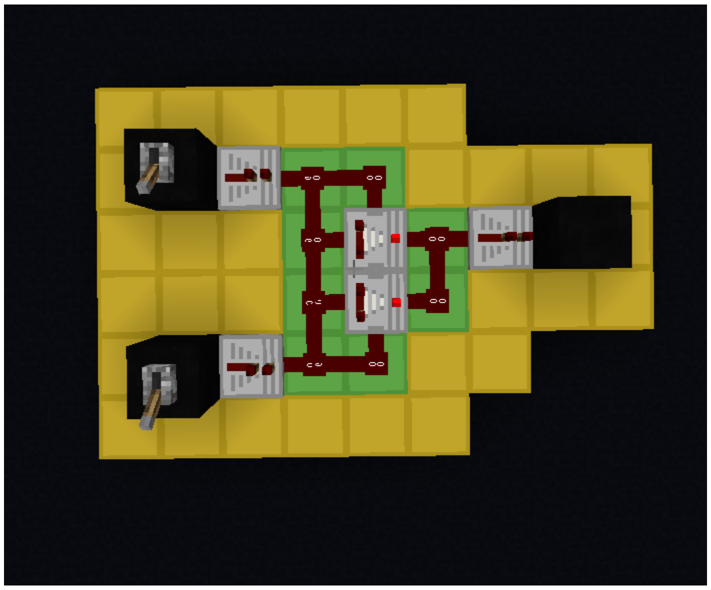
2.3 And Gate

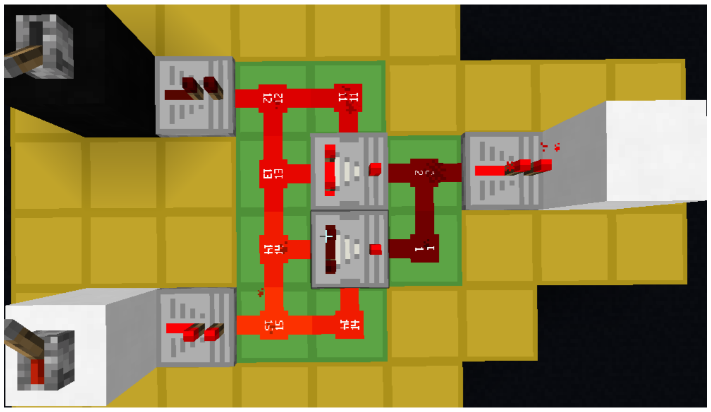
The and gate can be replicated by using not gates and or gates together. We send 2 inputs into not gates, where they are both connected to an or gate, then lastly connected to a not gate.



2.4 Xor Gate

The xor gate makes use of the comparator on subtraction mode. How it works is that it relies on the signal strength received by the comparators. If a signal is only coming from one input, then the comparator further away from the signal will activate, since the comparator near the signal will send a similar signal from the back and side, canceling the signal.

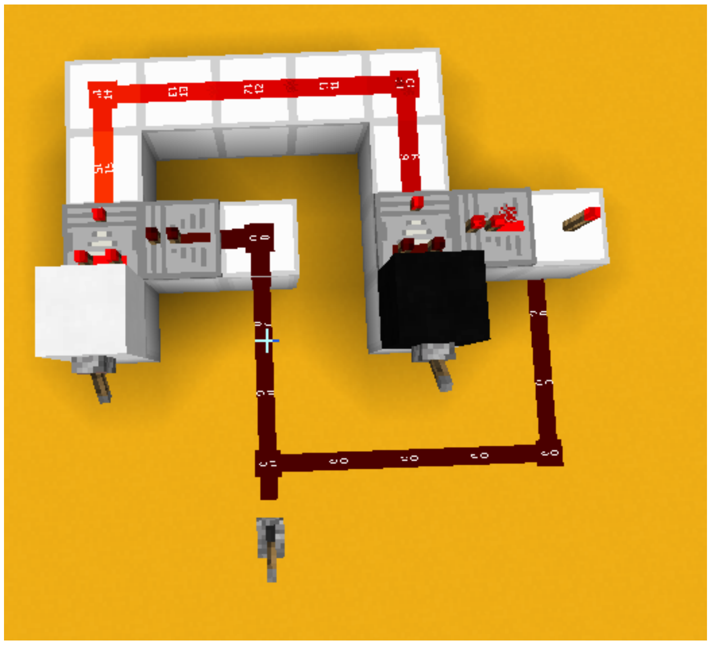




3 Combinational Logic Systems

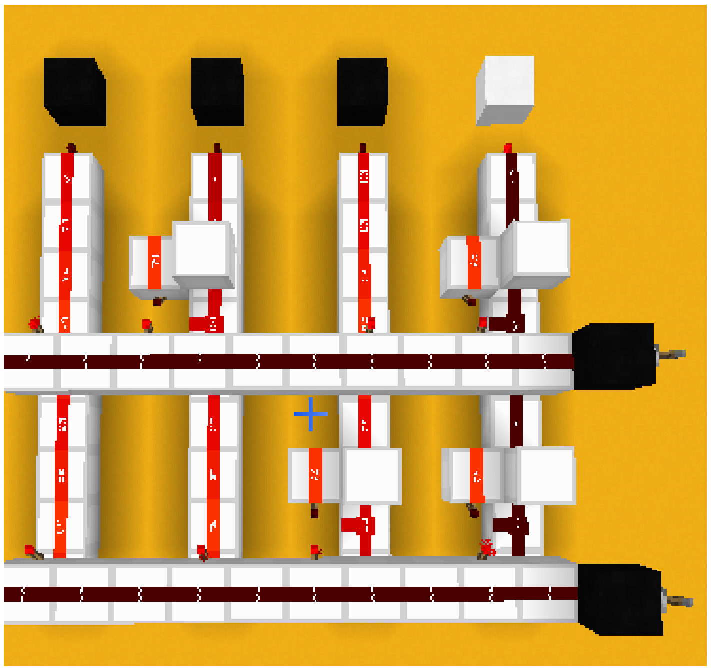
3.1 Multiplexer

Multiplexers makes use of the comparator on subtraction mode. How it works is that it has sides of all comparators powered by a repeater, and the only one closed is the selected input that will be outputted.



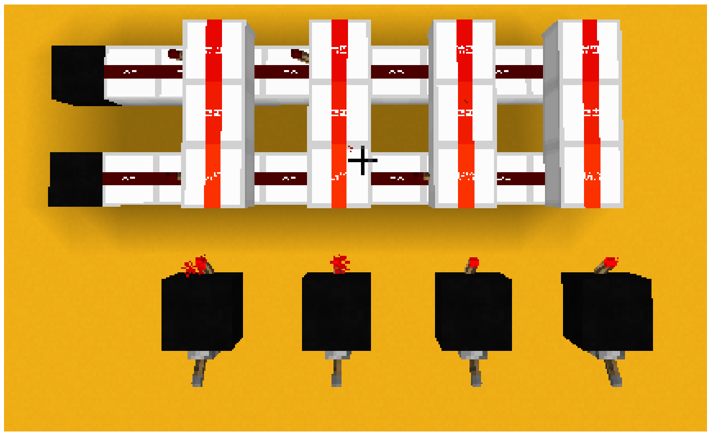
3.2 Decoder

Decoders makes use of not gates, buffer gates, and or gates. How it works is that buffers or not gates are mapped with each input, representing 0s and 1s. They are all connected by an or gate, which is then connected to a not gate.



3.3 Encoder

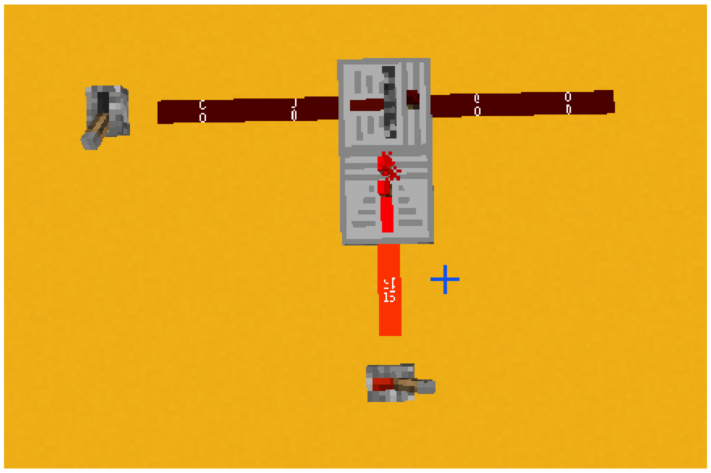
Encoders make use of not gates. How it works is that not gates are mapped with each input. When one of the inputs activate, the redstone wire will deactive, activating the redstone torches that are mapped to each output.



4 Flipflops

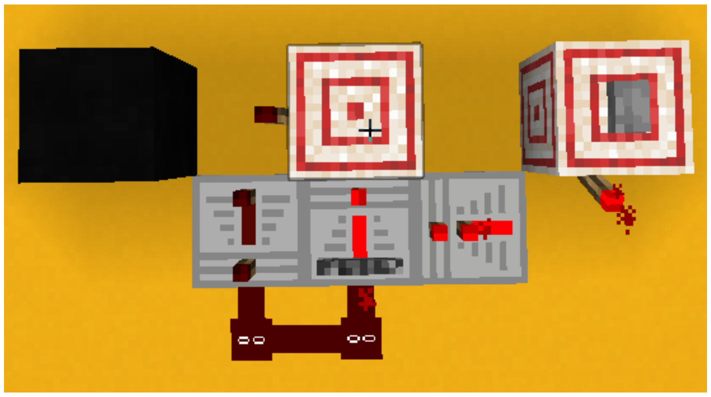
4.1 D Flipflop

D Flipflop makes use of the Locking system of a redstone repeater. When signal from another repeater is going into a repeater, it's state cannot change until the redstone repeater connected to it deactivates.



4.2 T Flipflop

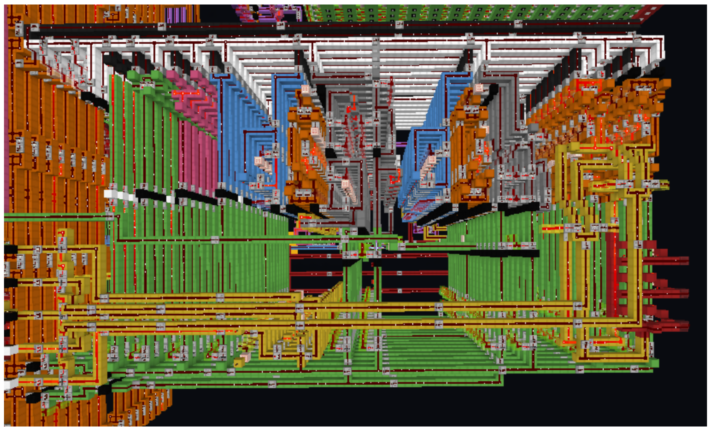
T flipflop makes use of a D flipflip with a looping not gate. The button will shortly deactivate the D flipflop, allowing the looping not gate to loop the signal, updating the state. With the timing of the button, the loop will only update once, thus changing a 0 to 1 or 1 to 0.



5 Arithmetic Logic Unit

The Arithmetic Logic Unit(ALU) is where every computation happens in the CPU. This consists of modules like ADD, SUB, MUL, DIV, etc. where it can accept 2 inputs and output the result of the operation selected. For this case study, the ALU contains the ff.

- ADD
- SUB
- MUL
- DIV
- AND
- OR
- XOR
- NOT



The Y Register serves as the temporary storage for the right side of the operation.
Bus OPERATION Y = Z

5.1 MUL and DIV (WAFC)

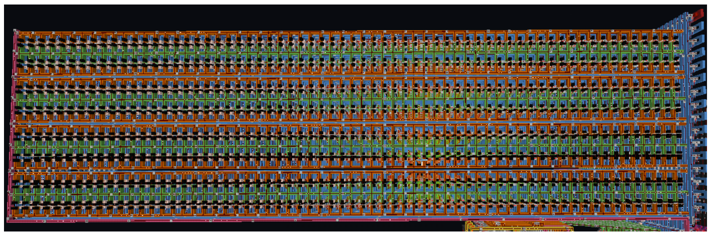
MUL and DIV are special operations that will take time so WAFC or Wait for Arithmetic Function to Complete was implemented for the control unit to wait for the operation to finish before resuming. Only MUL and DIV need this, other operations do not.

5.2 Flags (ENA FLAG)

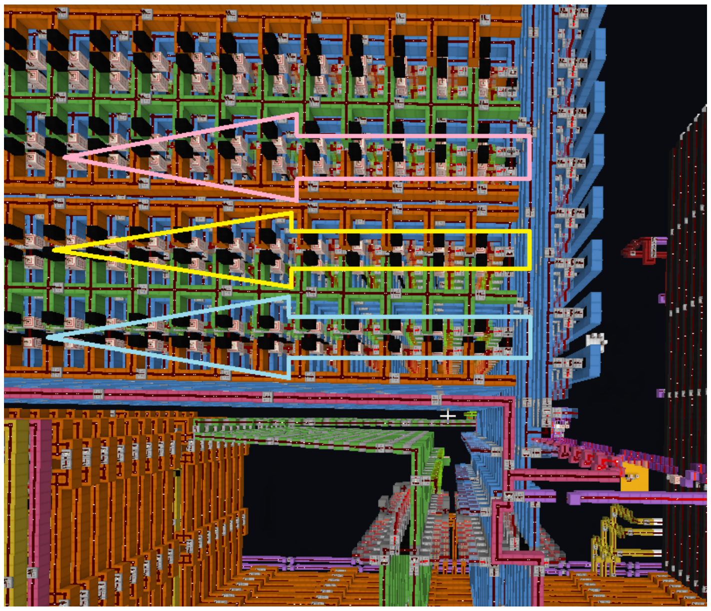
FLAGS are also implemented in the ALU which automatically computes based on the result, though unlike for CSARCH2, we need to enable ENA FLAG to update the flags for every operation that makes use of the FLAGS.

6 Main Memory

The Main Memory is a storage of bits where it can store data from address 0x0000 to 0x01FF. Each address contains 16 bits. The Main Memory has 2 functionalities, read and write. READ takes the address from the MAR and outputs whatever is stored in that address and stores it into the MDR. WRITE takes the address from the MAR and writes the data from MDR into the indicated address. After every operation, the memory will send a MEMORY FUNCTION COMPLETE signal, indicating that the operation is finished. WMFC is required to be called when doing a READ or WRITE since the computer may do another operation related to the READ or WRITE operation while the MEMORY OPERATION is not COMPLETE.



The address of the memory starts from 0x0000, indicated by the blue arrow in the image below. The blue arrow represents from 0x0000 to 0x003F. The yellow arrow represents 0x0040 to 0x007F. The pink arrow represents 0x0080 to 0x00BF.



IMPORTANT TO NOTE!!!

When doing WRITE, do not update the MAR in the same cycle as WRITE, as there may potentially be bugs due to the arrival time of the address.

DO NOT:
`REGout, MARin, WRITE, WMFC`

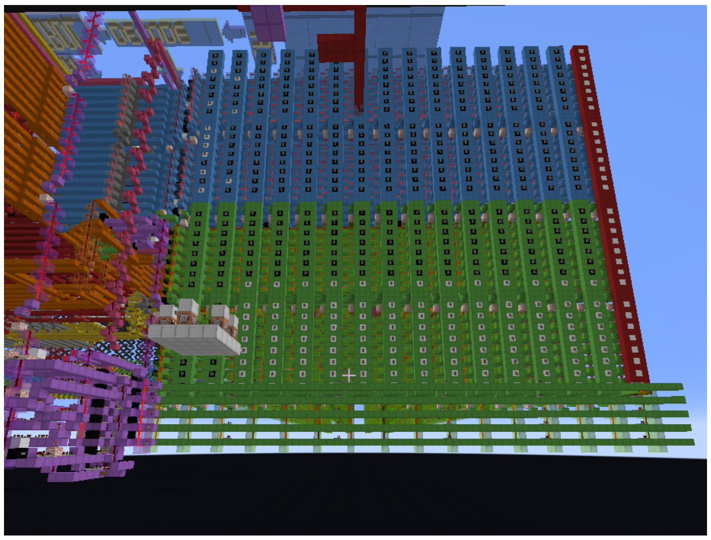
INSTEAD:
`REGout, MDRin, WRITE, WMFC`

7 Memory Loader

The Memory Loader is a component that allows users to write data directly into the memory. This will be further explained in the Module for the Memory Loader since using this component has strict requirements to follow.

IMPORTANT TO NOTE!!!

The address used when you do not wish to write anything is 0x01FF. You are given 15 inputs in the memory loader and if you only need to use 5 of them, the other 10 inputs should have the address field be set to 0x01FF.

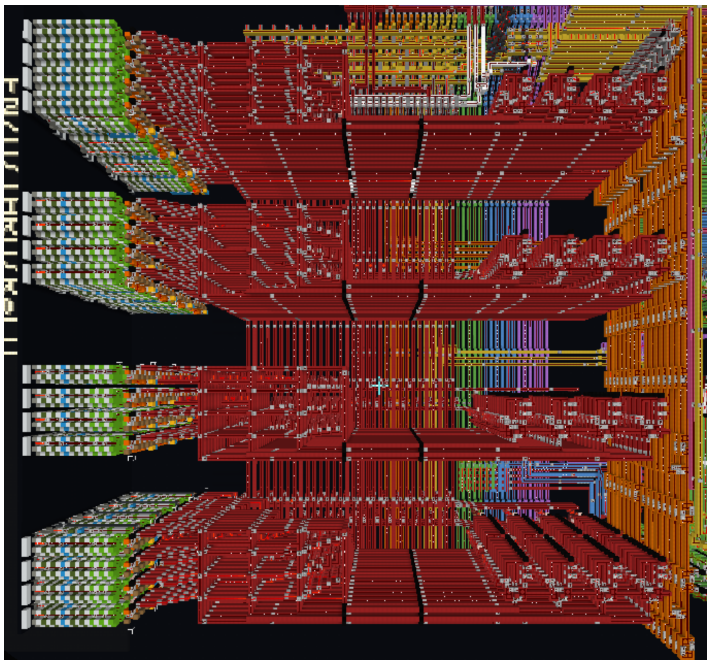


8 Register File

The Register File is a component consisting of 16 registers from R0 to RF. Some of these registers also have a counterpart label which is indicated in the table below.

Register	Label	Binary Equivalent
R0	AX	0000
R1	BX	0001
R2	CX	0010
R3	DX	0011
R4	SI	0100
R5	DI	0101
R6	BP	0110
R7	SP	0111
R8		1000
R9		1001
RA		1010
RB		1011
RC		1100
RD		1101
RE		1110
RF		1111

Each register has an In and Out operation where selecting In will automatically change the saved data with whatever data is written in the CPU bus. Out will output the data saved in the register out to the CPU bus.



9 Control Unit

The Control Unit is a component that controls the entire CPU. This component will be further explained in the modules since this component require a lot of visuals to understand.

Instruction Register	
IRin	Inputs into the Instruction Register to Decode
IRDFout	Outputs the Data Field from the Decoded Instruction
IRAFout	Outputs the Address Field from the Decoded Instruction
SEL OTR	Selects either the 2nd and 3rd nibble or the 3rd and 4th nibble from the IR

Arithmetic Logic Unit (ALU)	
SELECT Y	Selects the Y Register for ALU operations
SELECT 4	Selects constant value 4 as ALU input
Yin	Inputs into the Y Register for ALU computation
SET C	Sets the Carry Flag to 1
ALU MUL	Performs multiplication (X * Y)
ALU DIV	Performs division (X / Y)
ALU ADD	Performs addition (X + Y)
ALU SUB	Performs subtraction (X - Y)
ALU AND	Performs bitwise AND ($X \wedge Y$)
ALU OR	Performs bitwise OR ($X \vee Y$)
ALU XOR	Performs bitwise XOR ($X \oplus Y$)
ALU NOT	Performs bitwise NOT ($\neg$ X)
Zin	Inputs the ALU result into the Z Register
Z1out	Outputs lower 16 bits of the Z Register
Z2out	Outputs upper 16 bits of the Z Register
ENABLE FLAG	Enables or updates the ALU status flags
WAFC	Wait for ALU Function Complete signal

Main Memory	
MARin	Inputs to the Memory Address Register
MARout	Outputs the Memory Address Register contents
MDRin	Inputs data to the Memory Data Register
MDRout	Outputs data from the Memory Data Register
READ	Reads data from address in MAR into MDR
WRITE	Writes data from MDR into address in MAR
WMFC	Wait for Memory Function Complete signal

Program Counter	
PCin	Inputs address into the Program Counter
PCout	Outputs the current instruction address
RESET PC	Resets the Program Counter to 0

Registers	
REGAin	Inputs data into Register A
REGBout	Outputs data from Register B
REGAout	Outputs data from Register A
SPin	Inputs data into the Stack Pointer Register (Modifiable if needed)
SPout	Outputs data from the Stack Pointer Register (Modifiable if needed)

9.1 Program Counter

The program counter holds the address of the next instruction to be fetched from the Main Memory.

9.2 Instruction Register

The instruction register holds the current running instruction. The format of the binaries of an instruction splits each nibble to sections that represent the ff.

```
OPERATION OPERAND BUFFER OPERAND
```

Operation is the Op Code that selects which operation will it run. An example for this would be

```
MOV REG, REG
```

If the `MOV REG, REG` in your specifications is mapped to OP CODE 3, then when the Control Unit receives an instruction with the ff.

```
0011 XXXX XXXX XXXX
```

As for the `OPERAND` section, these contain the binary equivalent of the registers. By default, it follows this format.

```
OPERATION REG_A BUFFER REG_B
```

Note that Instructions that require the usage of 2 registers like `ADD DX, BX` will make use of both REG_A and REG_B.
`1100 0011 0000 0001`
`1100` -> ADD REG, REG
`0011` -> DX
`0000` -> BUFFER
`0001` -> BX

For Instructions that only use one register like `INC CX`, this will only use REG_A
`1101 0010 0000 0000`
`1101` -> INC REG
`0010` -> CX
`0000` -> BUFFER
`0000` -> BUFFER

As for the `BUFFER` section, this is used only for immediate data or labels like the ff.

- `MOV AX, 0x67`
- `MOV [0x21], BX`
- `JMP L3`
- `JNZ L2`

When dealing with immediate data like this, the `BUFFER` will be used to fit the data.
Immediate data and labels can only be from `0x00` to `0xFF`

IMPORTANT TO NOTE!!!

`SELECT OTHER` will be used for Immediate Values if the Immediate Value is located in the left operand of the instruction.

Ex. `MOV [0x21], DX`
Since if this in Binary is `0111 0010 0001 0011`, simply doing `IRAFout` will output `0001 0011` which is 0x13, not 0x21.
Doing `SELECT OTHER` together with `IRAFout` will output the 2nd and 3rd nibble, `0010 0001`, which is 0x21.

IMPORTANT TO NOTE!!!

Any `JMP` related instruction will have L1 be treated as an address and not an offset.

Ex. `JMP L3`
Since if this in Binary is `1000 0000 0001 1011`, the label will be located in the 3rd and 4th nibble of the instruction. `0001 1011` or 0x1B is the label that represents L3
This means that 0x1B contains the instruction after the Label(Instruction to run after JUMPING).

9.3 Register Controller

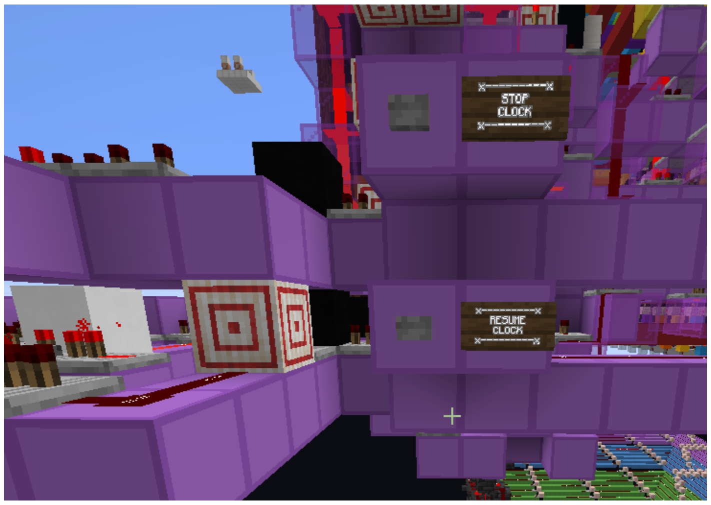
The register controller selects the registers based on the binary values in the Instruction Register.
By default, it will select the 2nd nibble as REG_A then the 4th nibble as REG_B.

`ADD DX, BX`
`1100 0011 0000 0001`
`1100` -> ADD REG, REG
`0011` -> DX
`0000` -> BUFFER
`0001` -> BX

`INC CX`
`1101 0010 0000 0000`
`1101` -> INC REG
`0010` -> CX
`0000` -> BUFFER
`0000` -> AX (Does not matter)

9.4 CPU Clock

The CPU Clock has a Pause and Resume button. When pressing the pause button, it will pause at the cycle, then resume will simply just resume it where it was last paused. This is also how `WMFC` and `WAFC` works.



IMPORTANT TO NOTE!!!

DO NOT Pause the clock at Cycle 0 or at any Cycle that does `READ` or `WRITE`.