

Contents

Overview	1
Hardware	2
Basic setup	2
IO Bus	2
Expanding ROM	3
Mb interrupts here	3
Expanding RAM	3
Handling Interrupts	4
Devices description	4
Peripheral Example	4
ROM Controller	7
RAM Controller	7
Interrupt Arbiter	7
Interrupt Enable Buffer	7
Address Decoder	7
Dynamic Interrupt Controller	7
IO Register	7
IO Hex Display Controller	7
IO Seven Segment Display Controller	7
IO Hardware Stack	7
IO Random Number Generator	7
Display Controller	7
Joystick Controller	7
Keypad Controller	7
Terminal Controller	9
Software	10
cocomake	10
VS Code Integration	12
Demonstration	12
Scheme Overview	12
Code Overview	13
Conclusion	13
Universal Modular Platform based on Cdm-8 processor	
Platform Description	

Overview

In our project we decided to build a universal platform that can be used for different purposes.

Hardware

In this section we will describe hardware part of this platform.

Basic setup

The bare minimum for this platform is cdm8 cpu, address decoder rom and ram

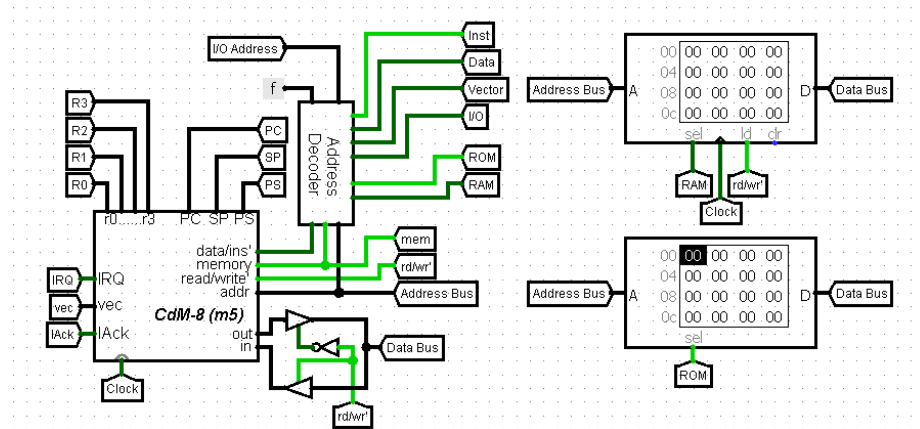


Figure 1: Minimal setup

IO Bus

To communicate with devices we need to define what IO bus looks like.

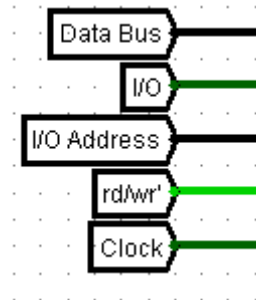


Figure 2: IO Bus

Bus lines:

- Data - processor data bus
- IO Address - lower 4 bits of processor address bus, generated by Address decoder

- IO Select - generated by **Address decoder**
- Read/Write - processor r/w' signal
- Clock - system clock signal

Expanding ROM

If we need more program memory we can use ROM controller to get more address space with memory paging technique.

We take **Address Out** signal of **ROM Controller** and connect it as higher bits of ROM's address input.

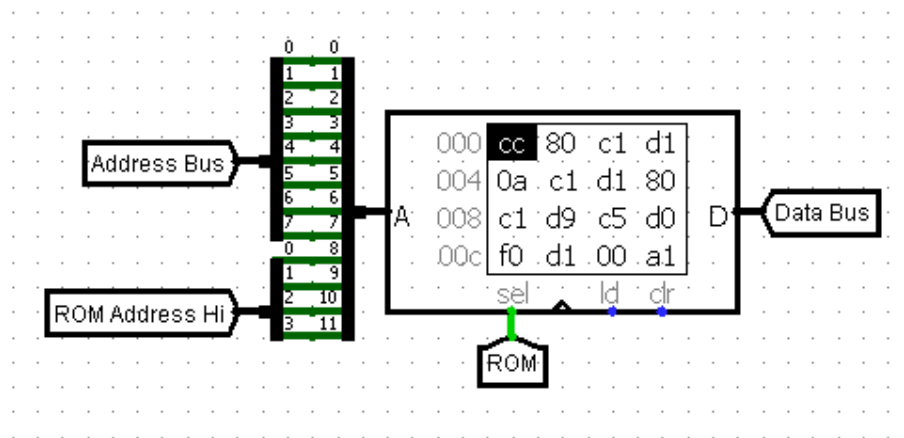


Figure 3: ROM chip with expanded address

scheme here

image here

Mb interrupts here

Expanding RAM

If we need more RAM we can use similar technique. The difference is that we divide RAM address space into two halves - lower half is global and upper half is paged.

RAM Controller forms expanded address for RAM chip.

scheme here

image here

Handling Interrupts

Without ROM Controller In Cdm8 in harvard setup interrupt vectors are located in in upper 16 bytes of program memory and therefore these vectors are constant.

In our platform you can use it as is or connect Dynamic Interrupt Controller which allows you to change these vectors by masking their addresses with external registers.

But this device is incompatible with ROM controller

With ROM Controller ROM Controller takes part in interrupt handling process - when interrupt occurs controller changes memory page to one that is specified on corresponding controller pins.

The easiest way to specify page to handle interrupts is to connect a constant to these pins, however in this case you cannot change it.

Better solution is to connect a register to bus and its output to ISR Page pins. In that case you can set page dynamically in runtime.

Devices description

In this block we will describe each device more precisely.

Peripheral Example

Most of devices connect to IO bus and therefore have similar block and signals that are used to communicate with the bus.

images with description

- **Select** - high when someone 'talks' to device, IO selected and IO address is the same as device address.

Of course, address decoding typically implemented through **AND** gates, but there we decided to replace it with **logisim**'s comparator to have an ability to conveniently set the address of devices. (perf)

- **General bus signals** - pins for corresponding bus signals
- **Device data bus** - pins that connect to data bus. Signals **Write** and **Read** are also generated here. They show whether we writing to this device or reading from it.

Typically, devices have general signals on their's north side and data bus pins on west side.

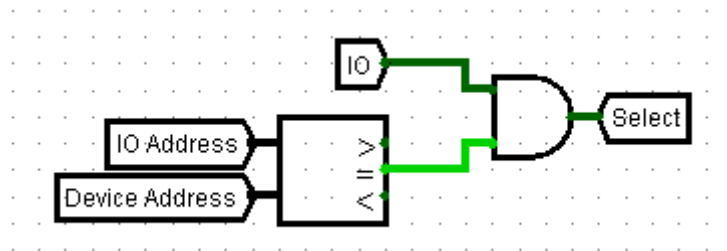


Figure 4: Forming of Select signal

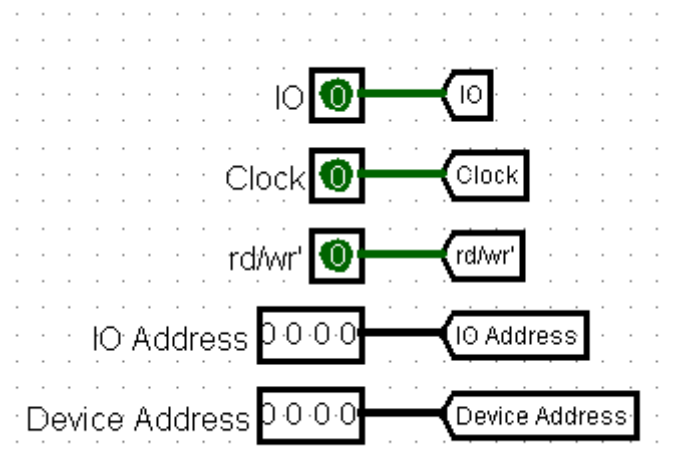


Figure 5: General bus singals

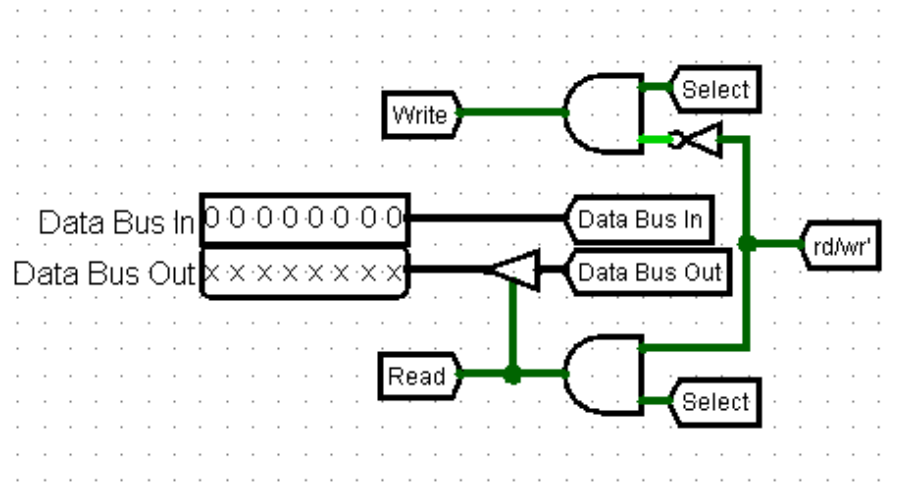


Figure 6: Device data bus

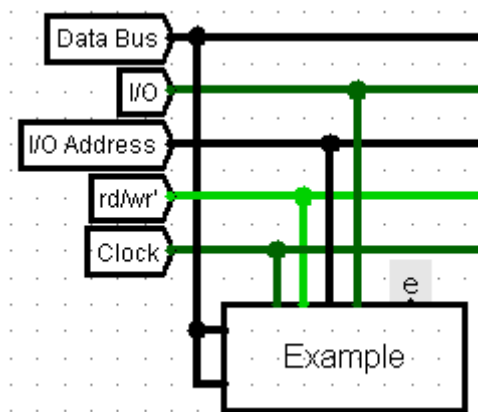


Figure 7: Connecting device to IO bus

ROM Controller

RAM Controller

Interrupt Arbiter

Interrupt Enable Buffer

Address Decoder

Dynamic Interrupt Controller

IO Register

IO Hex Display Controller

IO Seven Segment Display Controller

IO Hardware Stack

IO Random Number Generator

Display Controller

Joystick Controller

This controller drives 4-bit joystick.

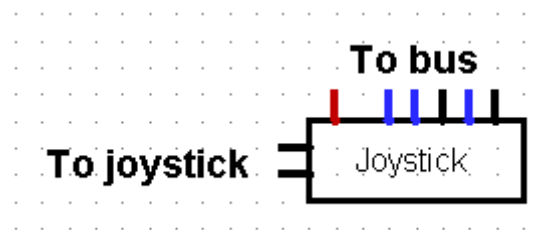


Figure 8: Joystick controller connection

Additional pins:

- X,Y (west) - pins to X and Y outputs of joystick

It just connects 4-bit X and Y pins to data bus.

When reading from it you get actual position of joystick.

Keypad Controller

This controller can drive up to 8 buttons. It can be used in polling mode or through interrupts.

Additional pins:

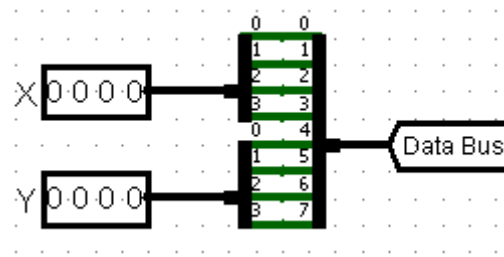


Figure 9: Joystick controller internals

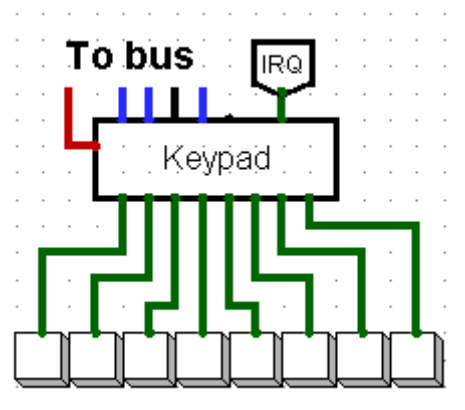


Figure 10: Keypad controller connection

- IRQ (north) - interrupt request line for this device, active when some buttons are pressed
- Button pins (south) - 8 pins for buttons

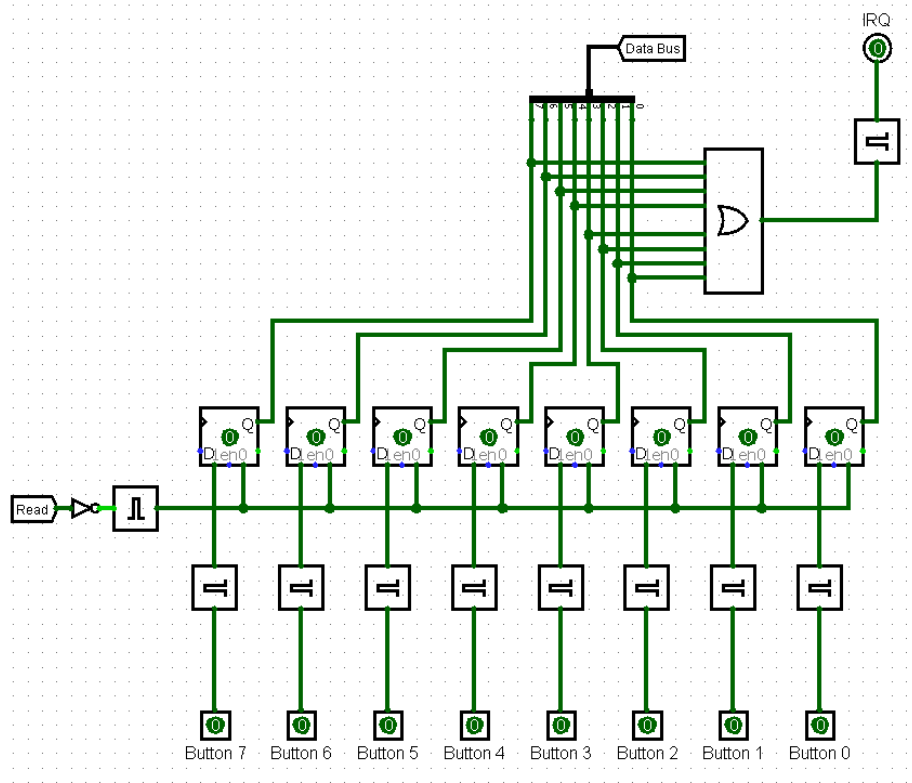


Figure 11: Keypad controller

It has 8 D-triggers each connected to a bit in a data bus. Buttons asynchronously set corresponding triggers. Triggers are reset on falling edge of **Read** signal (which is **rd/wr'** AND **Select**).

So, when reading from it, processor gets a byte that contains information about buttons that were pressed in the past (If some bit is 1, then corresponding button was pressed). After reading, all triggers are reset.

Moreover, if all triggers were zero and some button is pressed then a pulse occurs at IRQ output triggering interrupt.

Terminal Controller

This controller is used to drive terminal and keyboard.

Additional pins:

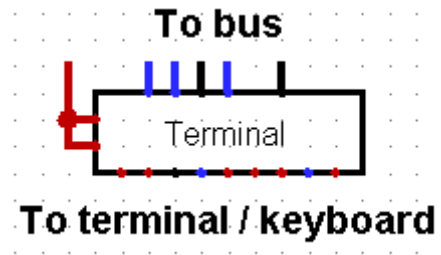


Figure 12: Terminal Controller Connnection

- Terminal/Keyboard pins (south) - pins that connect to terminal and keyboard

This controller basically just connects tarminal and keyboard to bus in a way that when writing, 7 bits of data (as ASCII symbol) goes to the terminal and **last bit of data AND Write** forms **Terminal Clear** signal. That means that we can write a character to terminal as well as clear it by sending 0x80.

When reading keyboard buffer connects to 7 bits of data bus and **Keybaord Available** goes to the last bit of data bus. That helps to read out a whole buffer. Just read from this device while data is not equal to 0x80.

This device supports interrupts. If keybaord buffer was empty and then there was some input, a pulse occurs on IRQ.

Software

In this part we will describe software part of this platform.

As we use more than 256 bytes of program memory and need to work with a lot of code default development tool (CocoIDE) is very unconfotable to use and that's why we developed some tools to make software development process easier.

cocomake

The main application that does hard work is cocomake. It is an incremental build system desined to work with multifile projects.

It is incremental, so only modified files get recompiled. That makes compiling much faster.

There, one bank(module) is one translation unit. Each file is compiled to an 256 byte image and then theese 256 byte images glued together to produce one big image that you load straight in logisim.

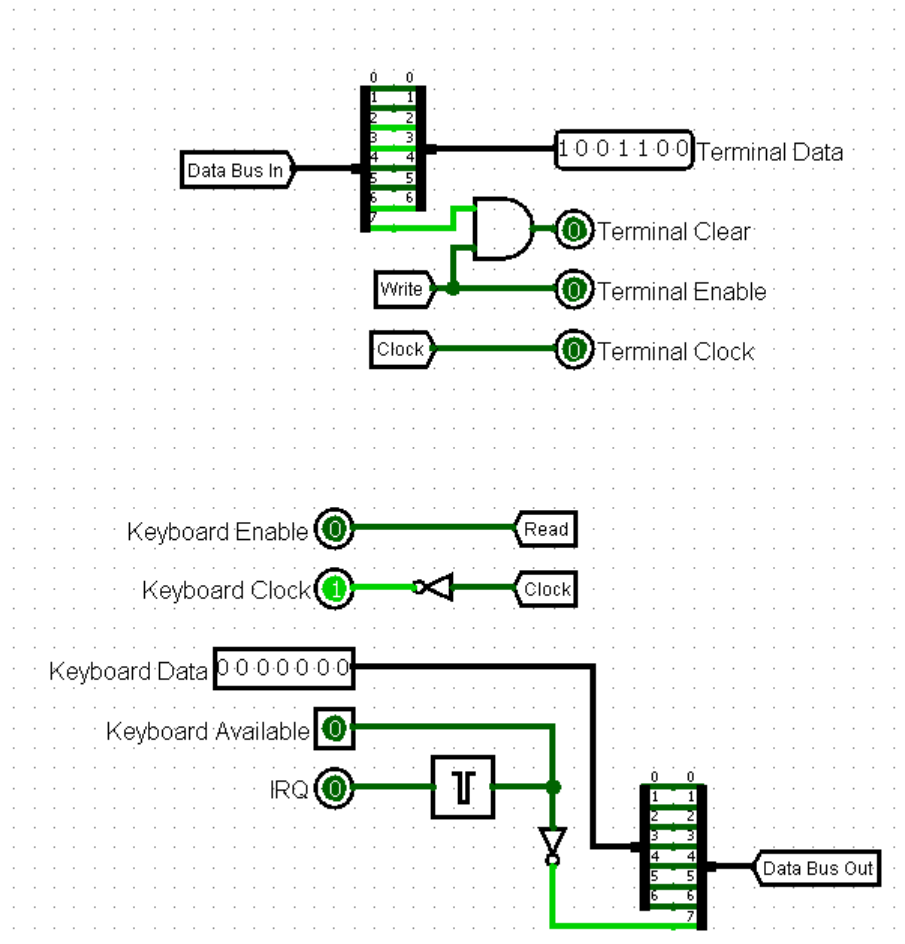


Figure 13: Terminal Controller

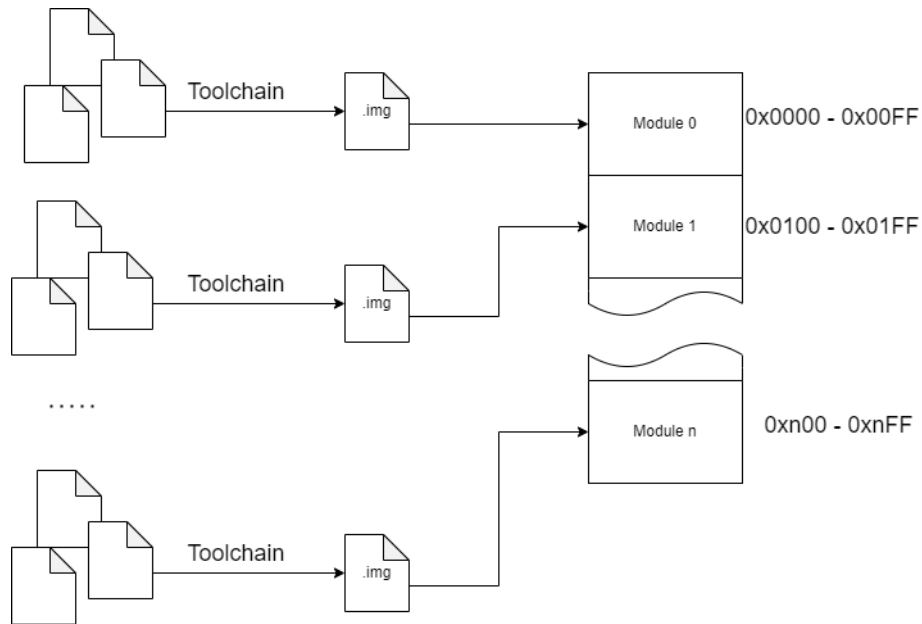


Figure 14: cocomake

So, you can have one big project with a lot of files spanning to many modules and you just execute one command and get your project compiled in one image.

VS Code Integration

For the text editor we decided to use VS Code as it is free modern software with a lot of customization options via extensions.

To make support for cdm8 assembler we developed an extension to VS Code that adds syntax highlighting for assembly and c preprocessor directives as well as code snippets.

Demonstration

In this section we will describe our demonstration setup.

Scheme Overview

image

We use this this this

Code Overview

We set up cocomake like this ...

code samples

Conclusion

idk