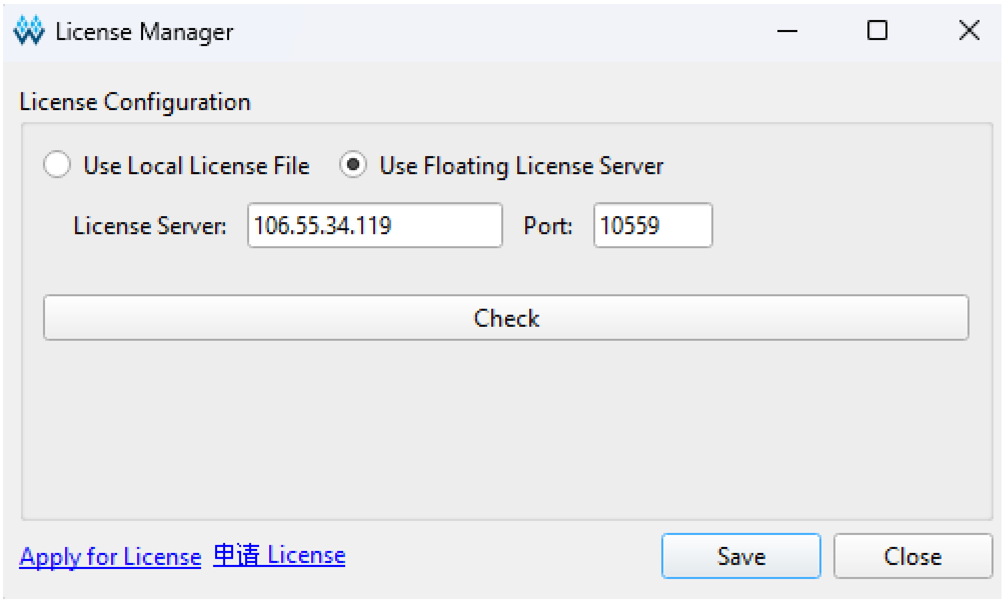
# Prerequisites

## 1) Downloading Gowin IDE and drivers

Gowin IDE is equivalent to Vivado IDE for the Sipeed FPGAs.
Download the Gowin IDE 1.9.10 or later from
https://www.gowinsemi.com/en/support/download_eda/

**For Windows:**
License:
For non educational versions (recommended) you need a license.
After installing the GOWIN IDE you can use the Floating Point license from Sipeed here.


License Manager

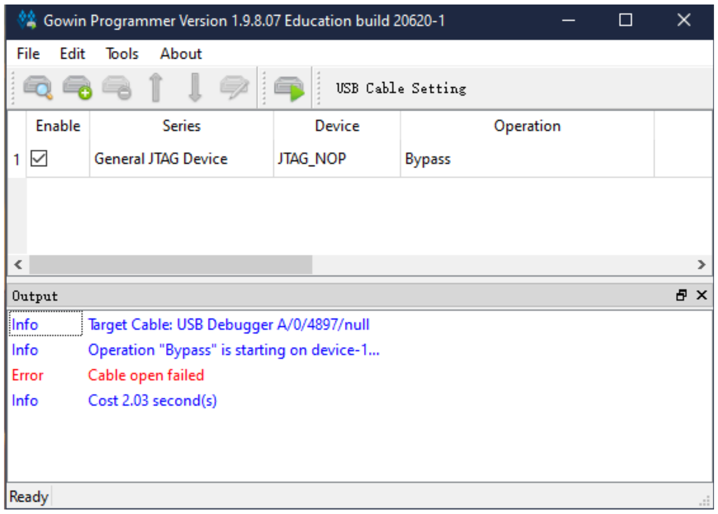
License Configuration

Use Local License File [✓ Use Floating License Server]

License Server: 106.55.34.119 Port: 10559

**Programming :**

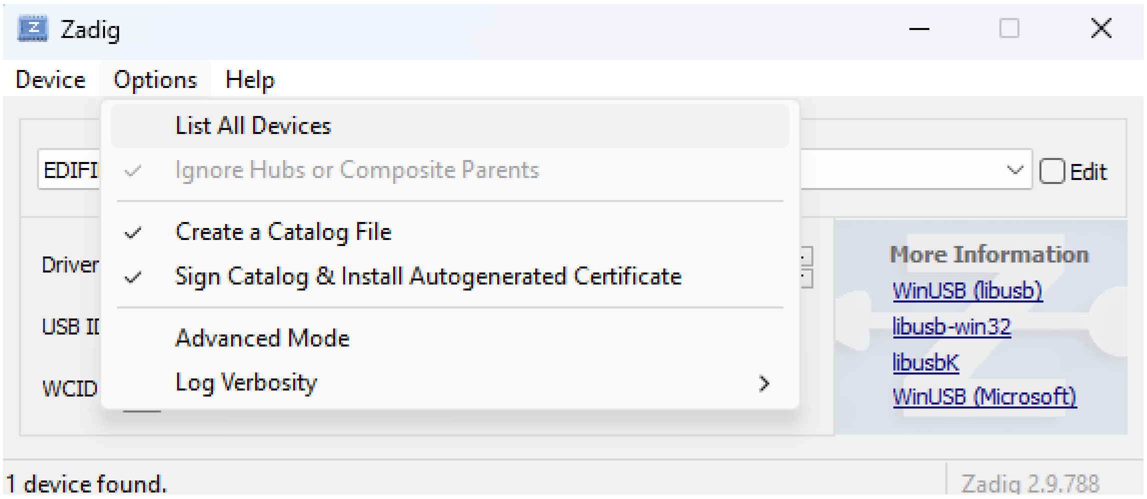
If when programming the bitstream you get the error



| | Enable | Series | Device | Operation |
|---|---|---|---|---|
| 1 | ✓ | General JTAG Device | JTAG_NOP | Bypass |

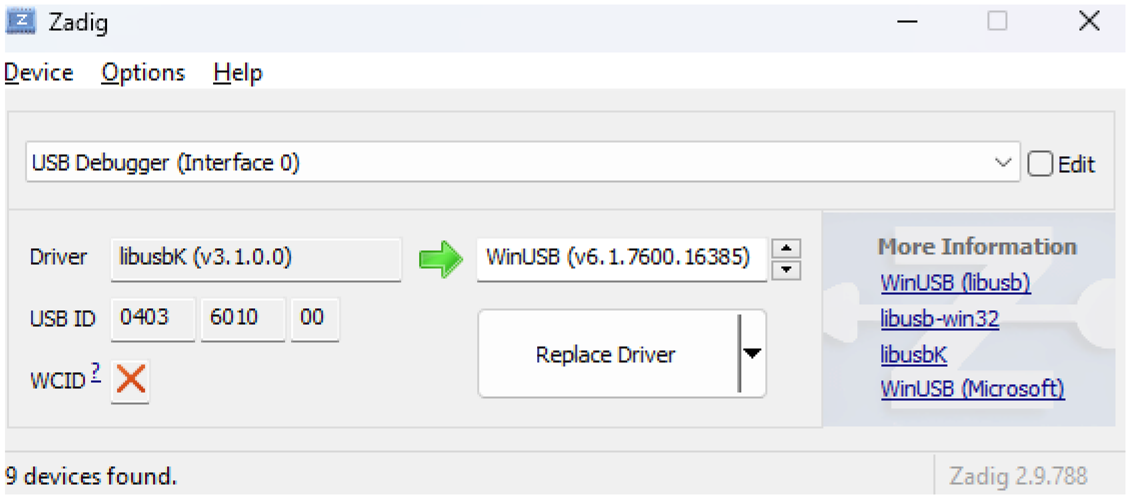
```
Output
Info    Target Cable: USB Debugger A/0/4897/null
Info    Operation "Bypass" is starting on device-1...
Error   Cable open failed
Info    Cost 2.03 second(s)
Ready
```

You will need to change the drivers of the FPGA.
First of all, install Zadig from [here](here).
After that, make sure the FPGA is plugged in the computer.  Open Zadig and enable "Options>List All Devices".



```
1 device found.                                   Zadig 2.9.788
```

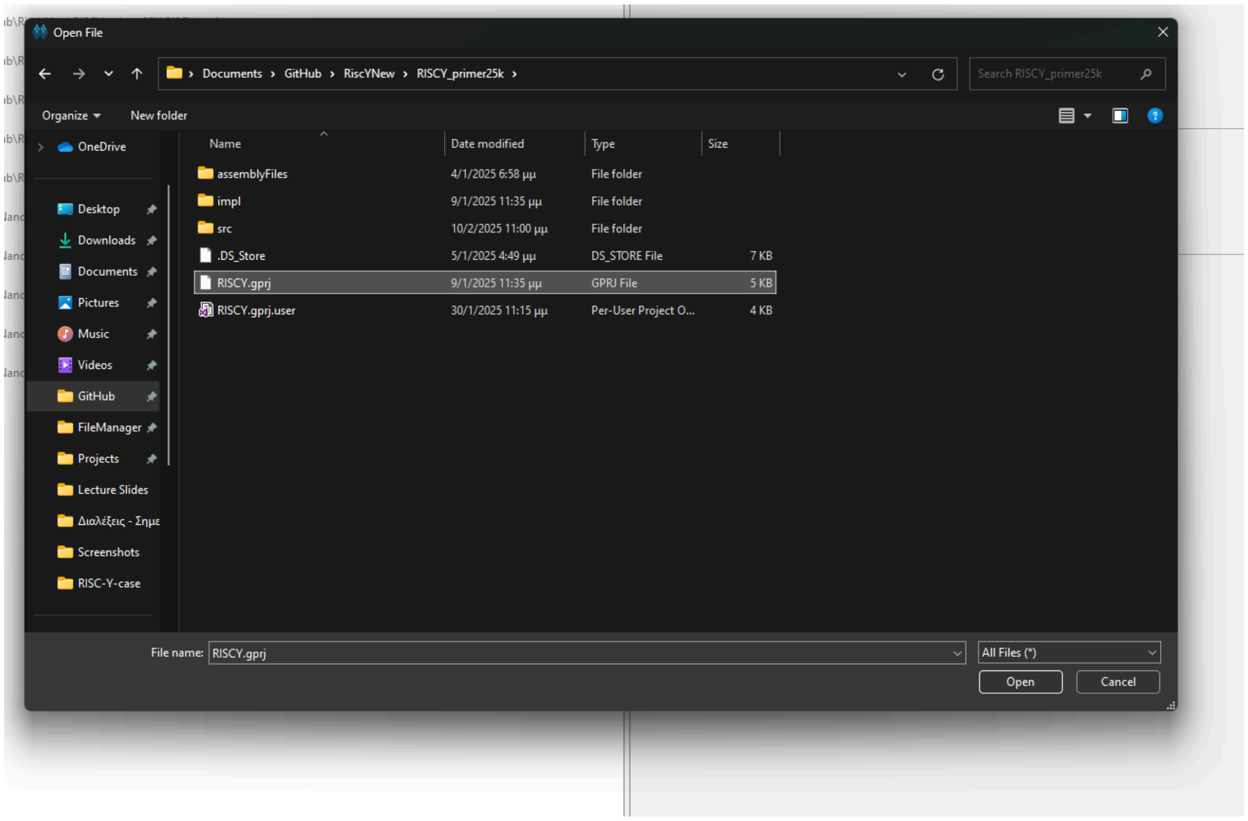
Then you will need to find two devices USB Debugger 0 and USB Debugger 1
Select them and replace the two drivers of 0 and 1 with WinUSB.



**For Linux:**
Download the Gowin IDE, extract it and go to Gowin>Drivers and use install.sh to install the drivers.
**For Mac:**
Download the Gowin IDE from the webpage for mac (under the linux tab) and follow the tutorial from this reddit post.

## 2) Generating Bitstream

Open the Gowin IDE either from the applications folder or by going to Gowin/Gowin_1.x.x/IDE/bin/gw_ide.exe (or gw_ide for linux)
Based on the board you are using you can use the project file for the board. In our case we will be using the Tang Primer 25k. Then press Open Project on Gowin IDE and navigate to the corresponding .grpj file.



Open File

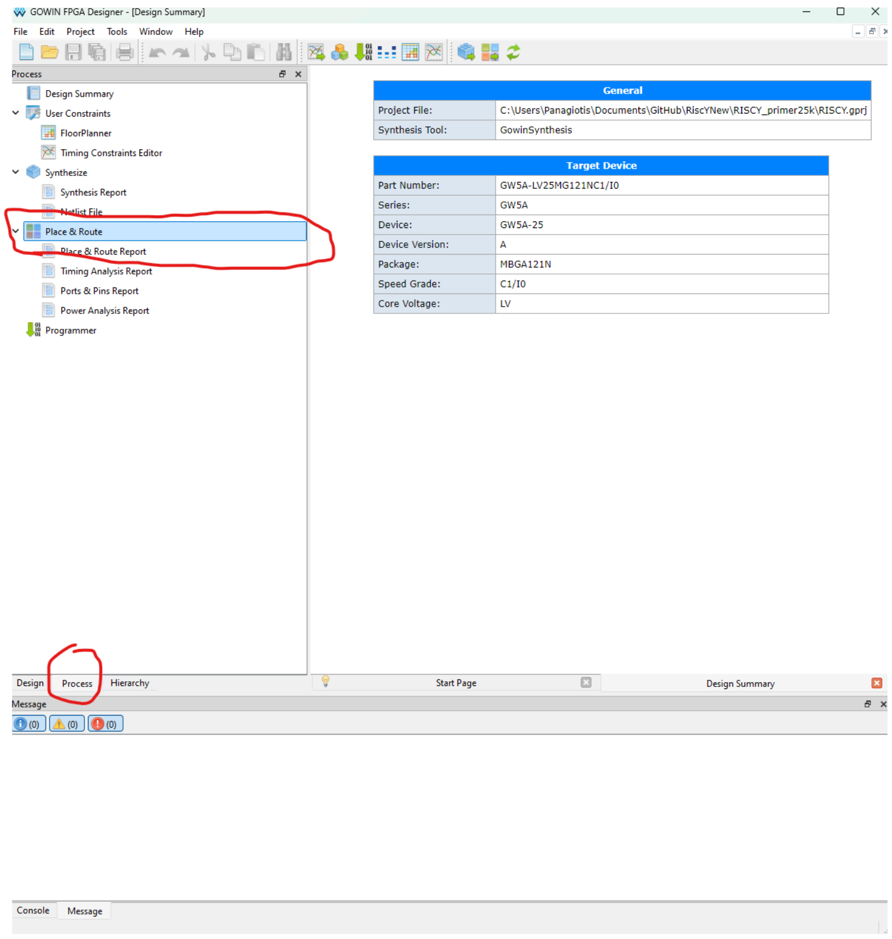
Documents › GitHub › RiscYNew › RISCY_primer25k ›

| Name | Date modified | Type | Size |
|---|---|---|---|
| assemblyFiles | 4/1/2025 6:58 μμ | File folder | |
| impl | 9/1/2025 11:35 μμ | File folder | |
| src | 10/2/2025 11:00 μμ | File folder | |
| .DS_Store | 5/1/2025 4:49 μμ | DS_STORE File | 7 KB |
| RISCY.gprj | 9/1/2025 11:35 μμ | GPRJ File | 5 KB |
| RISCY.gprj.user | 30/1/2025 11:15 μμ | Per-User Project O... | 4 KB |

File name: RISCY.gprj

All Files (*)

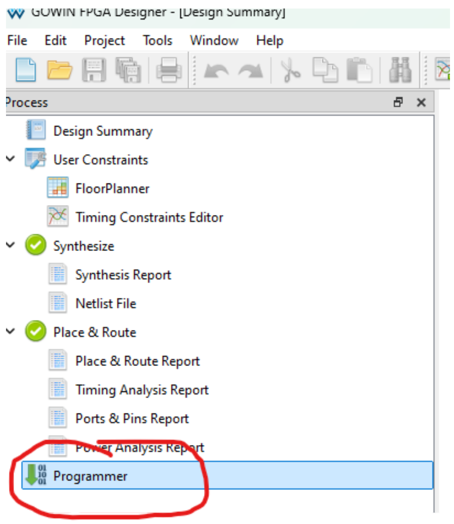
After that you can run the Synthesis and PNR by going to the process window and double clicking the Place and Route Button:



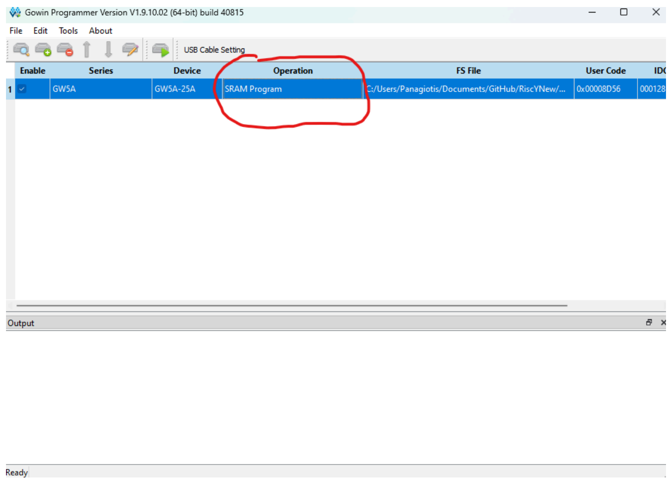
| General | |
|---|---|
| Project File: | C:\Users\Panagiotis\Documents\GitHub\RiscYNew\RISCY_primer25k\RISCY.gprj |
| Synthesis Tool: | GowinSynthesis |

| Target Device | |
|---|---|
| Part Number: | GW5A-LV25MG121NC1/I0 |
| Series: | GW5A |
| Device: | GW5A-25 |
| Device Version: | A |
| Package: | MBGA121N |
| Speed Grade: | C1/I0 |
| Core Voltage: | LV |

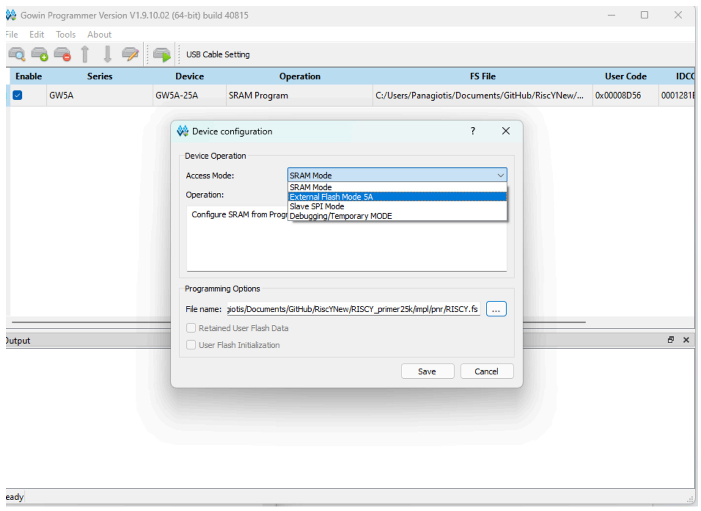
After the Placement and Routing has been completed open the programmer by double clicking the Programmer Button



This should open the programmer where you should double click on the Operation column and select Gowin External Flash Mode in order to save the bitstream on the FPGA. If you want to test the bitstream leaving it to SRAM Mode makes sure the bitstream is put on the SRAM and is not kept after a power off.



| | Enable | Series | Device | Operation | FS File | User Code | IDC |
|---|---|---|---|---|---|---|---|
| 1 | ✓ | GW5A | GW5A-25A | SRAM Program | C:/Users/Panagiotis/Documents/GitHub/RiscYNew/... | 0x00008D56 | 000128 |



| Enable | Series | Device | Operation | FS File | User Code | IDCO |
|---|---|---|---|---|---|---|
| ✓ | GW5A | GW5A-25A | SRAM Program | C:/Users/Panagiotis/Documents/GitHub/RiscYNew/... | 0x00008D56 | 0001281 |

Access Mode: SRAM Mode

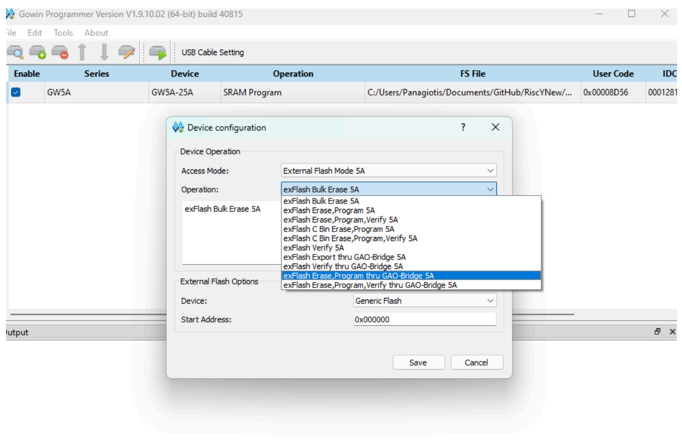
Operation:

Configure SRAM from Progr

File name: giotis/Documents/GitHub/RiscYNew/RISCY_primer25k/impl/pnr/RISCY.fs

Retained User Flash Data

User Flash Initialization


Gowin Programmer Version V1.9.10.02 (64-bit) build 40815

| Enable | Series | Device | Operation | FS File | User Code | IDC |
|---|---|---|---|---|---|---|
| ✓ | GW5A | GW5A-25A | SRAM Program | C:/Users/Panagiotis/Documents/GitHub/RiscYNew/... | 0x00008D56 | 0001281 |

Device configuration

Device Operation

Access Mode: External Flash Mode 5A

Operation: exFlash Bulk Erase 5A

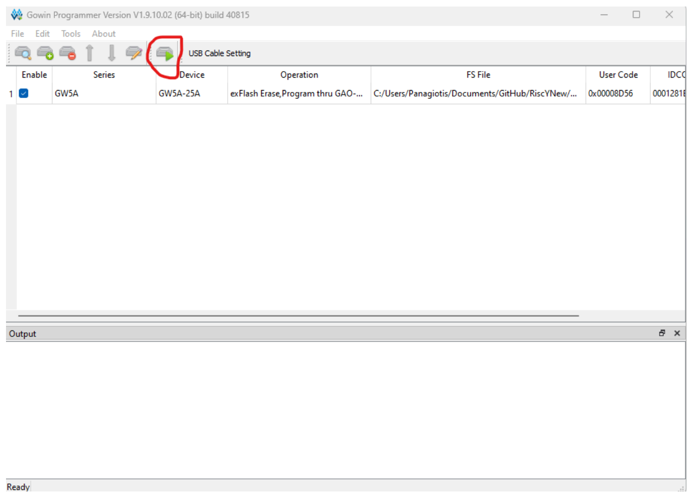
- exFlash Bulk Erase 5A
- exFlash Erase,Program 5A
- exFlash Erase,Program,Verify 5A
- exFlash C Bin Erase,Program 5A
- exFlash C Bin Erase,Program,Verify 5A
- exFlash Verify 5A
- exFlash Export thru GAO-Bridge 5A
- exFlash Verify thru GAO-Bridge 5A
- exFlash Erase,Program thru GAO-Bridge 5A
- exFlash Erase,Program,Verify thru GAO-Bridge 5A

exFlash Bulk Erase 5A

External Flash Options

Device: Generic Flash

Start Address: 0x000000


Gowin Programmer Version V1.9.10.02 (64-bit) build 40815

| | Enable | Series | Device | Operation | FS File | User Code | IDCC |
|---|---|---|---|---|---|---|---|
| 1 | ✓ | GW5A | GW5A-25A | exFlash Erase,Program thru GAO-... | C:/Users/Panagiotis/Documents/GitHub/RiscYNew/... | 0x00008D56 | 0001281E |

Then you program the FPGA by pressing the program button.
If  you are using the TangPrimer 25k make sure the HDMI PMOD is plugged in the the middle PMOD and the buttons are plugged into the PMOD closest to the usb-c connection. Along with that you can also use a keyboard to navigate the bootloader.

## 2) Software

### GCC Compiler:

You will need to install the *riscv64-unknown-elf-gcc*. You can either install a precompiled binary or compile it from source.
**For Linux:**

```
sudo apt update
sudo apt install gcc-riscv64-unknown-elf
```

Then to keep the gcc on the bash

```
export PATH=/opt/riscv/bin:$PATH
source ~/.bashrc
```

To make sure the compiler is correctly installed you can run

```
riscv64-unknown-elf-gcc --version
```

This should show you the version of gcc you are using

**For MacOs:**

You will need to install homebrew. Follow this tutorial to download it.

```
brew tap riscv-software-src/riscv
brew install riscv-tools
```

To make sure the compiler is correctly installed you can run

```
riscv64-unknown-elf-gcc --version
```

## Building an Application

Go to the riscYcompiler/riscYcompiler/

Open this on your editor of choice.  You can write code on main.c
There are some already made code samples on riscYcompiler/riscYcompiler/WorkingCodes/

For start copy the code form helloWorldNew.c file to the main.c on the root folder.
After that run make.
There should be a builds/ folder that contains all your builds. Go to the newest e.g. build0/program.bin

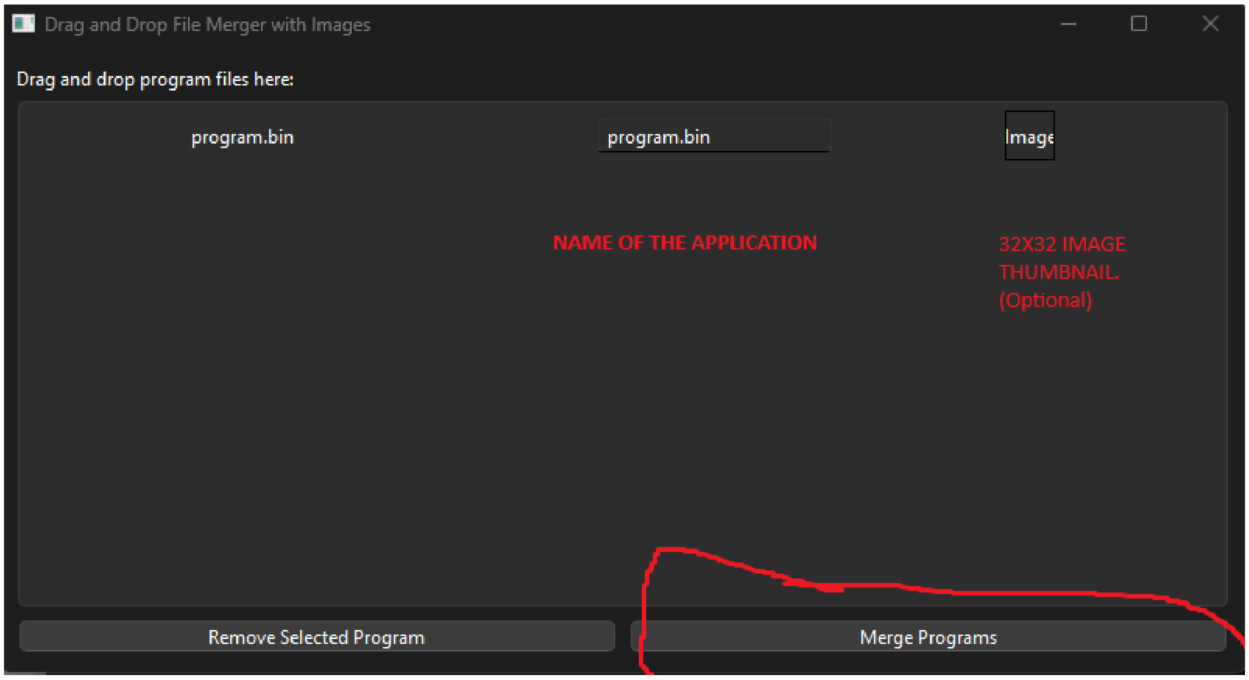
After you confirm the binary file has been generated, go to riscYcompiler/MergerV2/
Then run the merger.py python file.

*$python3 merger.py*

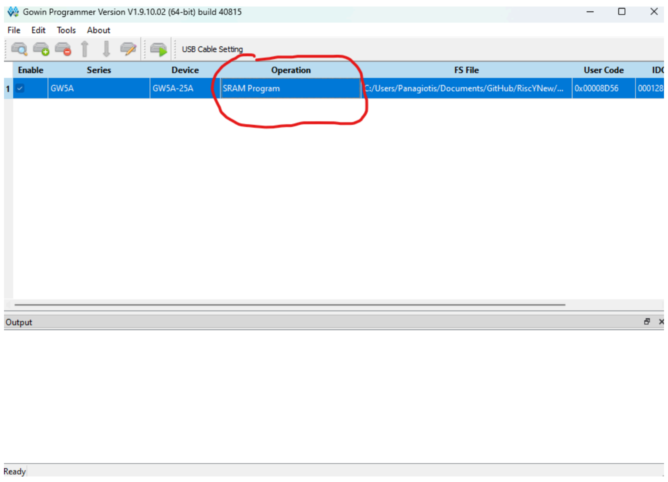
After that drag and drop the binary you built earlier and press merge.  You can add as many builds (applications) as you want preferably less than 4.

IF you want to add a thumbnail you can drag and drop a 16x16 image. There as some sample images already on the merger folder that you can drag and drop. Make sure the image is dropped on the Image square and not the whole window.

This should generate a **merged_programs.bin** file on the same folder.


Drag and Drop File Merger with Images
Drag and drop program files here:
program.bin
program.bin
Image
NAME OF THE APPLICATION
32X32 IMAGE THUMBNAIL. (Optional)
Remove Selected Program
Merge Programs

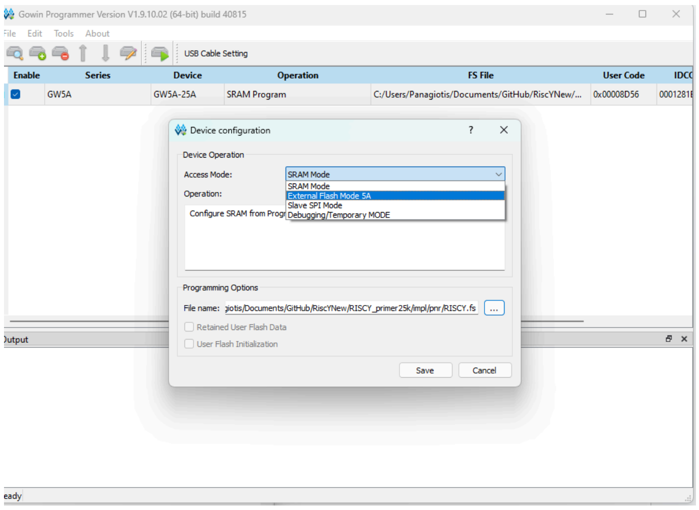
After that go to the Gowin IDE, Open the programmer and go to


Gowin Programmer Version V1.9.10.02 (64-bit) build 40815

| Enable | Series | Device | Operation | FS File | User Code | IDC |
|---|---|---|---|---|---|---|
| ✓ | GW5A | GW5A-25A | SRAM Program | C:/Users/Panagiotis/Documents/GitHub/RiscYNew/... | 0x00008D56 | 000128 |

Output

Ready


Gowin Programmer Version V1.9.10.02 (64-bit) build 40815

| Enable | Series | Device | Operation | FS File | User Code | IDCC |
|---|---|---|---|---|---|---|
| ✓ | GW5A | GW5A-25A | SRAM Program | C:/Users/Panagiotis/Documents/GitHub/RiscYNew/... | 0x00008D56 | 0001281E |

Device configuration

Device Operation

Access Mode: SRAM Mode

SRAM Mode
External Flash Mode 5A
Slave SPI Mode
Debugging/Temporary MODE

Operation:

Configure SRAM from Progr

Programming Options

File name: giotis/Documents/GitHub/RiscYNew/RISCY_primer25k/impl/pnr/RISCY.fs

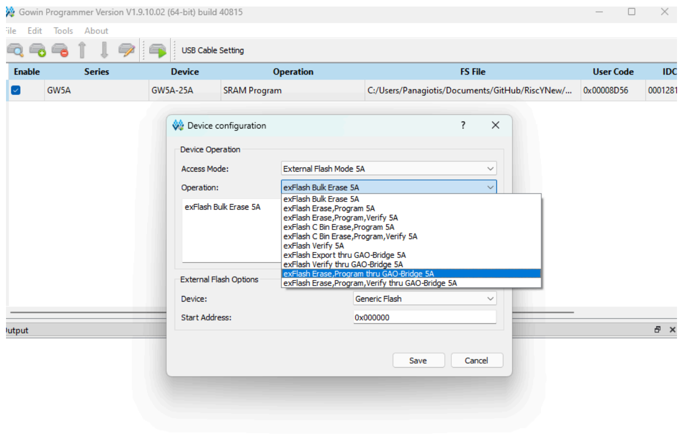
Retained User Flash Data

User Flash Initialization

Save Cancel

Output

eady



| Enable | Series | Device | Operation | FS File | User Code | IDC |
|---|---|---|---|---|---|---|
| ✓ | GW5A | GW5A-25A | SRAM Program | C:/Users/Panagiotis/Documents/GitHub/RiscYNew/... | 0x00008D56 | 0001281 |

Device Operation

Access Mode: External Flash Mode 5A

Operation: exFlash Bulk Erase 5A

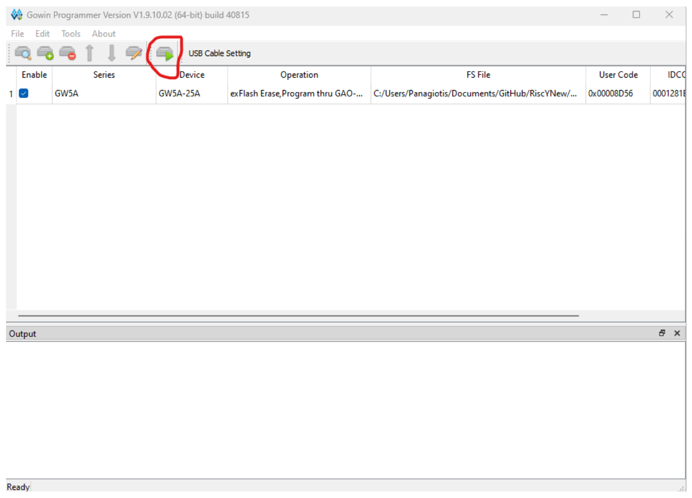
- exFlash Bulk Erase 5A
- exFlash Erase,Program 5A
- exFlash Erase,Program,Verify 5A
- exFlash C Bin Erase,Program 5A
- exFlash C Bin Erase,Program,Verify 5A
- exFlash Verify 5A
- exFlash Export thru GAO-Bridge 5A
- exFlash Verify thru GAO-Bridge 5A
- exFlash Erase,Program thru GAO-Bridge 5A
- exFlash Erase,Program,Verify thru GAO-Bridge 5A

exFlash Bulk Erase 5A

External Flash Options

Device: Generic Flash

Start Address: 0x000000

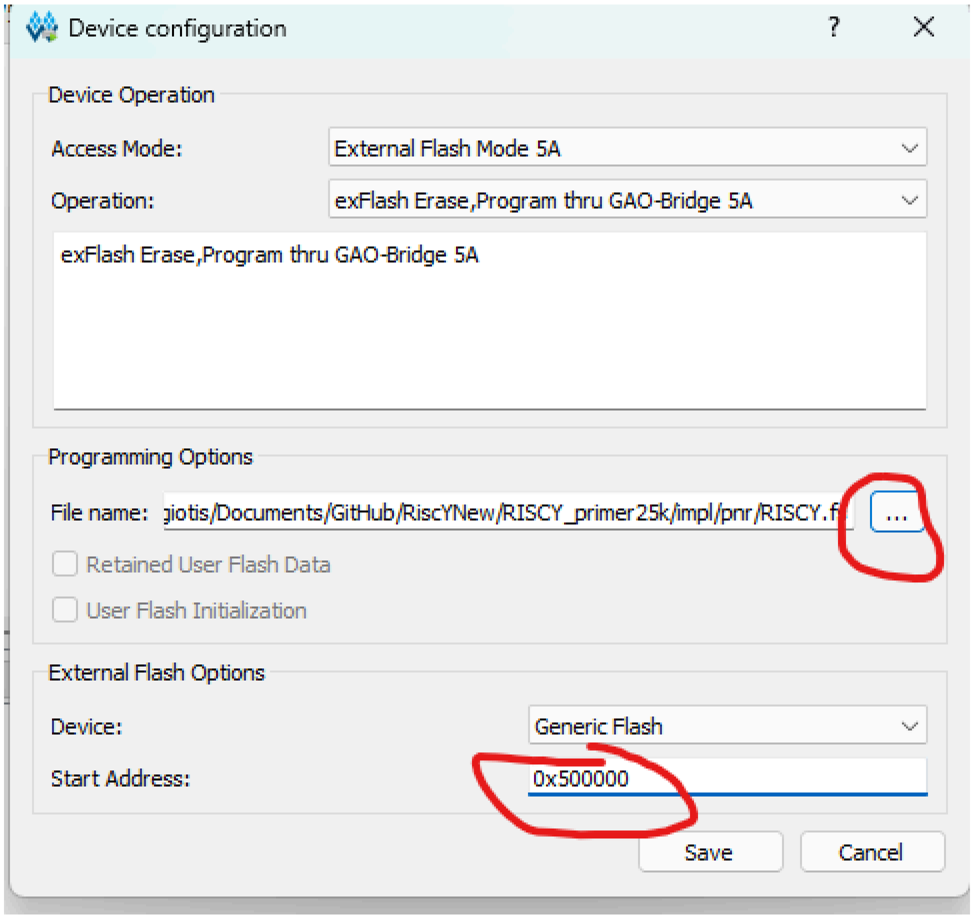


|  | Enable | Series | Device | Operation | FS File | User Code | IDCC |
|---|---|---|---|---|---|---|---|
| 1 | ✓ | GW5A | GW5A-25A | exFlash Erase,Program thru GAO-... | C:/Users/Panagiotis/Documents/GitHub/RiscYNew/... | 0x00008D56 | 0001281E |

After that if you are using the Tang Primer 25k or the Tang Nano 20k change the start address to 0x500000.

Under the FIle name change this to the location of the **merged_programs.bin** file. If you are on the Tang Primer 25k there is a bug and you will need to use the **output.fs** that was also generated from the python script.

After that press the send bit stream as previously and it should work.


Device configuration

Device Operation

Access Mode: External Flash Mode 5A

Operation: exFlash Erase,Program thru GAO-Bridge 5A

exFlash Erase,Program thru GAO-Bridge 5A

Programming Options

File name: giotis/Documents/GitHub/RiscYNew/RISCY_primer25k/impl/pnr/RISCY.f

Retained User Flash Data

User Flash Initialization

External Flash Options

Device: Generic Flash

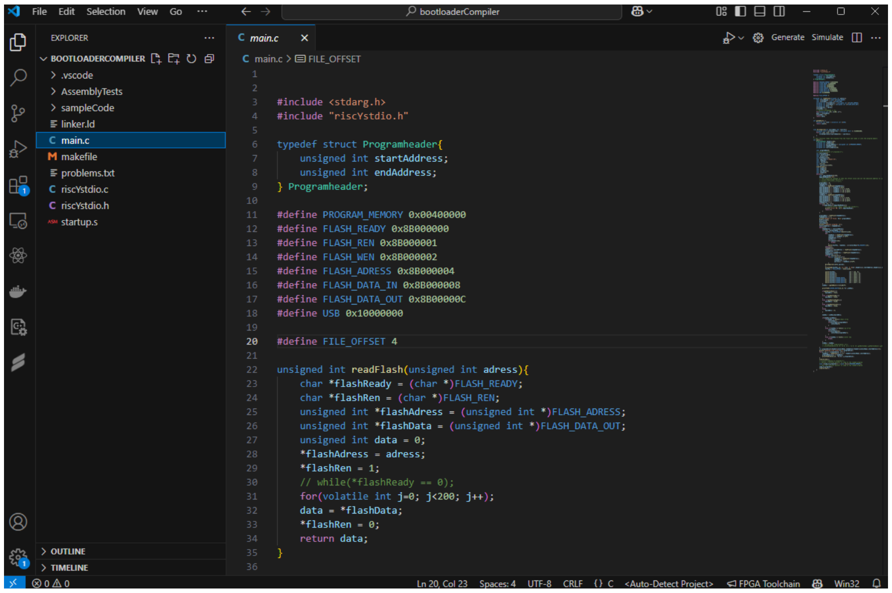
Start Address: 0x500000

## Bootloader Changes

The bootloader code is the code that is accessed by the cpu when it is initialized. Unlike to program code, the bootloader can only house 4KB of instructions/data. This is why you should be very careful on what you add since more likely than not, the bootloader will crash.

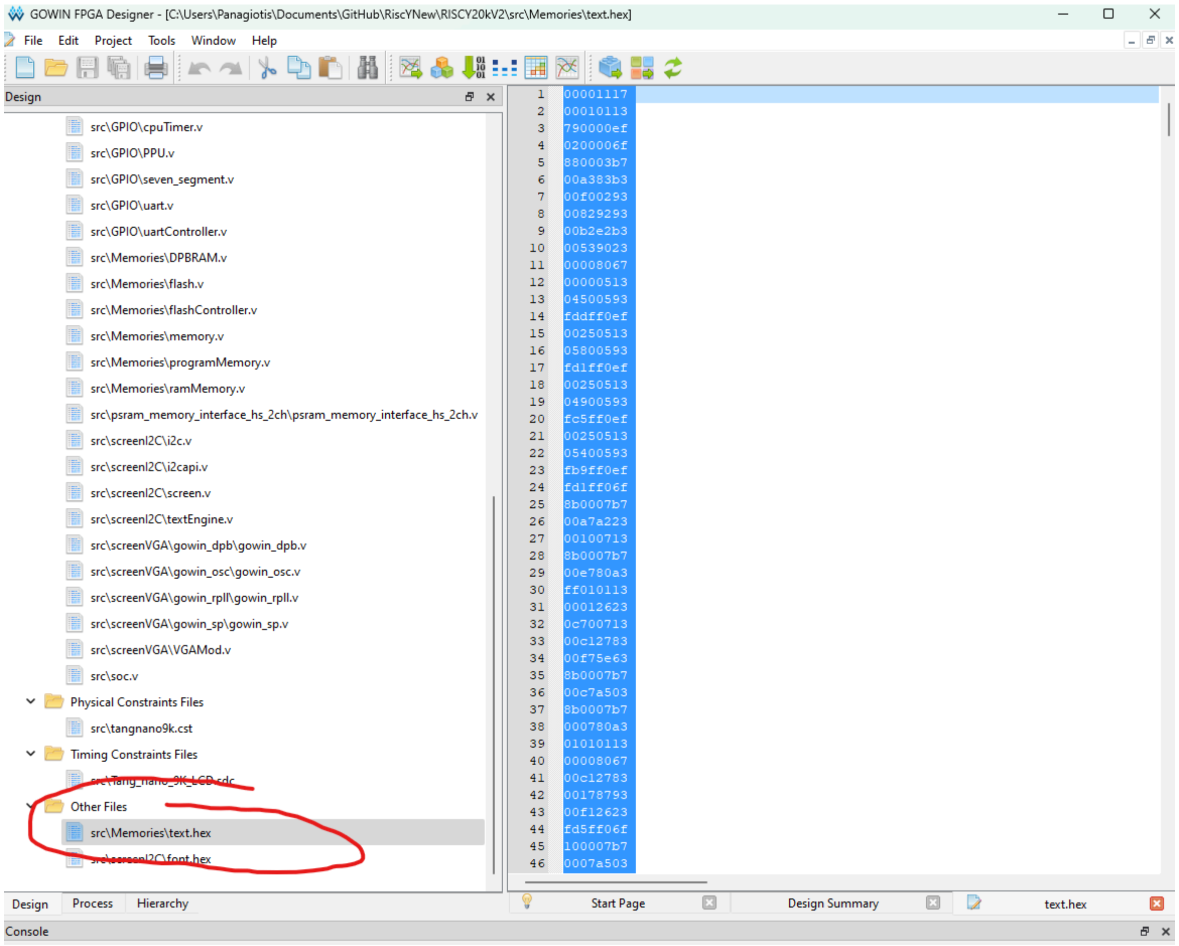
To change the code of the bootloader go to riscYcompiler/bootloaderCompiler/

Open this in VS code or your editor of choice. The main.c is the file that is compiled and is the first program running on the CPU.



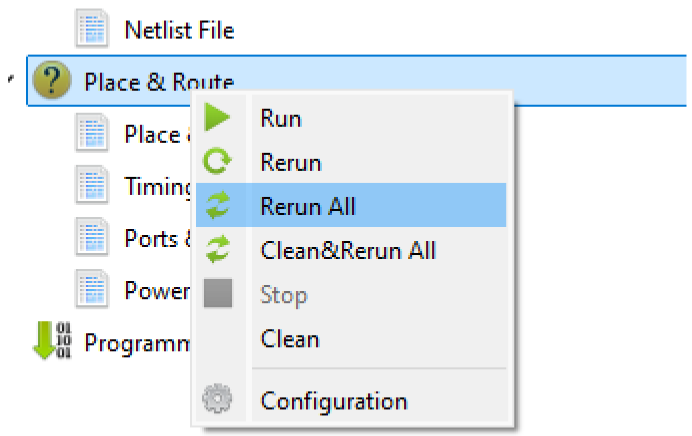
```c


#include <stdarg.h>
#include "riscYstdio.h"

typedef struct Programheader{
    unsigned int startAddress;
    unsigned int endAddress;
} Programheader;

#define PROGRAM_MEMORY 0x00400000
#define FLASH_READY 0x8B000000
#define FLASH_REN 0x8B000001
#define FLASH_WEN 0x8B000002
#define FLASH_ADRESS 0x8B000004
#define FLASH_DATA_IN 0x8B000008
#define FLASH_DATA_OUT 0x8B00000C
#define USB 0x10000000

#define FILE_OFFSET 4

unsigned int readFlash(unsigned int adress){
    char *flashReady = (char *)FLASH_READY;
    char *flashRen = (char *)FLASH_REN;
    unsigned int *flashAdress = (unsigned int *)FLASH_ADRESS;
    unsigned int *flashData = (unsigned int *)FLASH_DATA_OUT;
    unsigned int data = 0;
    *flashAdress = adress;
    *flashRen = 1;
    // while(*flashReady == 0);
    for(volatile int j=0; j<200; j++);
    data = *flashData;
    *flashRen = 0;
    return data;
}

```

After changing the code, you can go to builds/buildX/program.hex (where X is the latest build number). After that you can copy the hex code from the program.hex and paste it under the text.hex in the bottom of the source files in GowinEDA



```
00001117
00010113
790000ef
0200006f
880003b7
00a383b3
00f00293
00829293
00b2e2b3
00539023
00008067
00000513
04500593
fddff0ef
00250513
05800593
fd1ff0ef
00250513
04900593
fc5ff0ef
00250513
05400593
fb9ff0ef
fd1ff06f
8b0007b7
00a7a223
00100713
8b0007b7
00e780a3
ff010113
00012623
0c700713
00c12783
00f75e63
8b0007b7
00c7a503
8b0007b7
000780a3
01010113
00008067
00c12783
00178793
00f12623
fd5ff06f
100007b7
0007a503
```

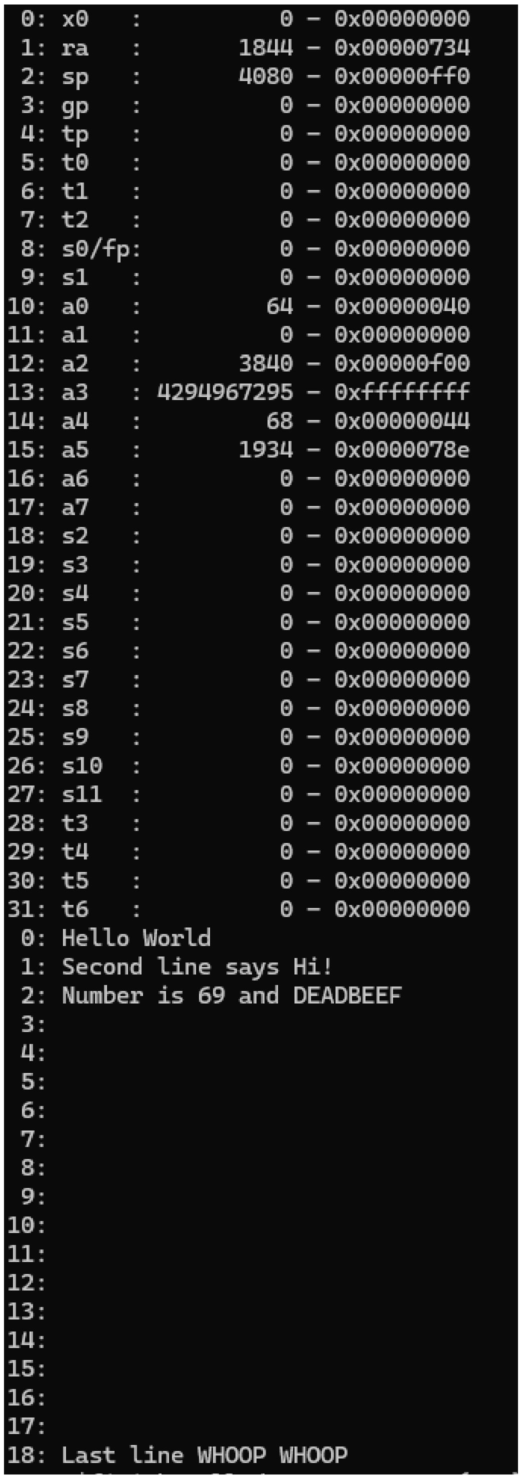
After that right click the Place & Route and click Rerun All



After that just flash the bitstream and your new bootloader code should show up when resetting the FPGA.

## Testbench.

For testing your code, you can use a testbench that runs the assembly code of your liking from the bootloader memory.
To do this navigate to RISCY_primer25k/src/ and open that folder in Code.
After that you can copy paste your assembly hex file that has been compiled with the bootloaderCompiler to includes/testbenchtext.hex and do any changes to the verilog files you want.
Then go to ./Testbench/ and run the script “runTB.bash” if you are on linux/macOS or “testbench.bat” for Windows. That should produce a gtkwave file and a debug output of the runtime.
If your code prints anything in the screen you will be able to see the final output on the terminal like this example.

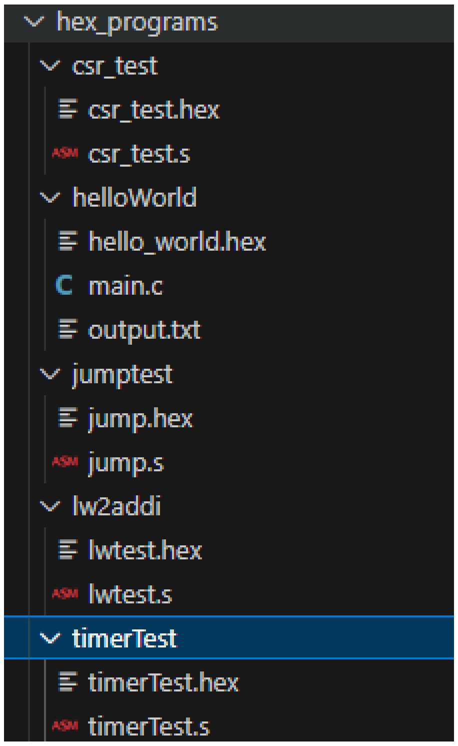


```
 0: x0   :          0 - 0x00000000
 1: ra   :       1844 - 0x00000734
 2: sp   :       4080 - 0x00000ff0
 3: gp   :          0 - 0x00000000
 4: tp   :          0 - 0x00000000
 5: t0   :          0 - 0x00000000
 6: t1   :          0 - 0x00000000
 7: t2   :          0 - 0x00000000
 8: s0/fp:          0 - 0x00000000
 9: s1   :          0 - 0x00000000
10: a0   :         64 - 0x00000040
11: a1   :          0 - 0x00000000
12: a2   :       3840 - 0x00000f00
13: a3   : 4294967295 - 0xffffffff
14: a4   :         68 - 0x00000044
15: a5   :       1934 - 0x0000078e
16: a6   :          0 - 0x00000000
17: a7   :          0 - 0x00000000
18: s2   :          0 - 0x00000000
19: s3   :          0 - 0x00000000
20: s4   :          0 - 0x00000000
21: s5   :          0 - 0x00000000
22: s6   :          0 - 0x00000000
23: s7   :          0 - 0x00000000
24: s8   :          0 - 0x00000000
25: s9   :          0 - 0x00000000
26: s10  :          0 - 0x00000000
27: s11  :          0 - 0x00000000
28: t3   :          0 - 0x00000000
29: t4   :          0 - 0x00000000
30: t5   :          0 - 0x00000000
31: t6   :          0 - 0x00000000
 0: Hello World
 1: Second line says Hi!
 2: Number is 69 and DEADBEEF
 3:
 4:
 5:
 6:
 7:
 8:
 9:
10:
11:
12:
13:
14:
15:
16:
17:
18: Last line WHOOP WHOOP
```

Please notice that you can also see the registers on their final state and also the lines printed on screen. Usually if the above Hello World program runs ok in the testbench, chances are that it's going to run correctly on the FPGA as well.

There is a folder with some sample tests that you can use to verify the correctness of the cpu with ranging complexity of code.

You can find them on Testbench/hex_programs along with their corresponding hex code and assembly/C



```
hex_programs
  csr_test
    csr_test.hex
    csr_test.s
  helloWorld
    hello_world.hex
    main.c
    output.txt
  jumptest
    jump.hex
    jump.s
  lw2addi
    lwtest.hex
    lwtest.s
  timerTest
    timerTest.hex
    timerTest.s
```

## To Dos

1) **DRAM controller.** A controller for giving basic instruction and data to the cpu via the DDR3 of the Tang Nano 20k(64Mbit) and Tang Primer 25k(512Mbit) FPGA boards.
2) **Support For Better Verification.** A better workflow for finding architecture errors due to changes. Can also be used along with the development of co simulation.
3) **Operating System.** A small operating system like ZephyrOS could be bootstraped to work for our cpu. This would also the user privilege mode (U-Mode) to be implemented in the architecture of the CPU. Along with that a module to check if the U-Mode has access to that part of the memory via a memory table would also be useful.
4) **Adding the Performance Monitor CSR Registers.** We can also add performance monitors to monitor cache hits/misses, branch flushes, loads/stores etc.
5) **Increase single core performance**
    a) **Branch Predictors:** we can increase the single core performance with the help of branch predictors that would minimize the stalls when doing for loops.
    b) **Out of order execution:** Implementing an out of order execution algorithm could vastly increase the performance of the core by eliminating a lot of the stalls.
6) **Virtual Memory.** This goes along with the Operating System section but it is worth mentioning as it's own part.
7) **Peripherals:**
    a) **SD card file reading**

    b) **Memory Mapped GPIO**
    c) **I2C controller**

8) **Multicore.** Adding more cores to the soc could increase the performance and be worthwhile for educational purposes. This can either be done with a master/slave core configuration where a master core initiates the other cores and manages them, or a more typical approach where all the cores are managed by the operating system.