

# Supplementary Material Part 2: Assembling and loading code for your NYU-6463-RV32I

Author: Hammond Pearce, 2021-10-19

In order to generate code to execute on your custom NYU-6463-RV32I processor you will need to get it into the binary format that the Verilog/VHDL tooling understands. This is actually the same process that would be taken in order to get your code to execute on a real-world processor as well. When the code is written in assembly, this process is known as *assembling*.

We break up this document into two sections. Part A will cover how you can generate your binary code file from your assembly code listing. Part B will then cover how you can get this binary code into the program.

We assume that you are generating binary code for the following simple listing:

```
/*  
 * Assembly 'reset handler' function upon startup.  
 */  
.global reset_handler  
.type reset_handler,@function  
reset_handler:  
    addi x1, x0, 1  
    addi x2, x0, 2  
loop:  
    add x1, x1, x2  
    j loop
```

*Listing 1: Simple program*

You may ignore the directives for now (the lines beginning with periods).

This simple program adds register 0 to value 1 and sets the result to register 1. As register 0 is hardwired to 0, this sets register 1 to the value  $0+1 = 1$ . Likewise, register 2 is set to value 2. The next instruction adds register 1 and 2 together and stores the result back in register 1. Finally, the final instruction makes this occur in a loop. The observed output should thus be:

*Register 1 = 1*

*Register 2 = 2*

*Register 1 = 3, 5, 7, ...*

Let's now look at how to make our processor run these instructions!

## Part A: Generating binary code

### 1. Hand assembly

The simplest but most labor intensive method for generating the binary code for the program in Listing 1 is to hand-assemble it. Using the supplementary project material Part 1, we can see the RISC-V encoding for each of the different instructions. We have used three instructions: addi, add, and j. Let us examine the instruction encoding for each.

For ADDI:

ADDI	imm[11:0]	rs1	000	rd	0010011
------	-----------	-----	-----	----	---------

- addi x1, x0, 1 ->
  - Immediate is “1” -> 000000000001,
  - rs1 is x0 -> “00000”,
  - “000”,
  - rd is “x1” -> “00001”,
  - “0010011”
    - -> 000000000001 00000 000 00001 0010011
    - -> 0000 0000 0001 0000 0000 0000 1001 0011 (spacing)
    - ->    0    0    1    0    0    0    9    3
    - -> 00 10 00 93
    - -> 00100093 - RISC-V ready!

We can likewise derive codes for the remaining instructions:

- addi x2, x0, 2
  - -> 0000000000010 00000 000 00010 0010011
  - -> 0000 0000 0010 0000 0000 0001 0001 0011
  - -> 00201213

ADD	0000000	rs2	rs1	000	rd	0110011
-----	---------	-----	-----	-----	----	---------

- add x1, x1, x2
  - -> 0000000 00010 00001 000 00001 0110011
  - -> 0000 0000 0010 0000 1000 0000 1011 0011
  - -> 002080B3

Now, you might notice that we have a “j” instruction. But the specifications do not have a “j”! This is known as a pseudoinstruction, and is a shorthand for a longer instruction. We can look up in the manual how “j” is implemented, along with the other pseudo-instructions, in Table 25.3 of the official specification: <https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf>

Here, it notes that “j” is implemented where “j offset” is equal to “jal x0, offset”

As such,

JAL	imm[20 10:1 11 19:12]	rd	1101111
-----	-----------------------	----	---------

- jal x0, -4
  - (We jump back by -4 so that we repeat the previous instruction infinitely)
  - Immediate = -4 = (32 bits 2's complement)
  - = 1111 1111 1111 1111 1111 1111 1100
  - This instruction has (a subset of) the bits in a curious order,
  - imm =
  - = 1 | 1111111110 | 1 | 11111111
  - Rd = 00000
    - 1 1111111110 1 11111111 00000 1101111
    - 1111 1111 1101 1111 1111 0000 0110 1111
    - FFDF06F

We now have the full program ready!

```
Program binary = 00100093 00201213 002080B3 FFDF06F
                  | addi   | | addi   | | add   | | j      |
```

In the specification, NYU-6463-RV32I says that it starts program memory at 0x01000000. So we would know to load these instructions from that address.

## 2. Automatic assembly using gcc

Alternatively, instead of hand-compiling, we can also generate the program using gcc. This will require you to install the complete riscv toolchain. To do this, follow the following instructions at your terminal:

(Inside an Ubuntu Linux 20.04 installation - e.g. a VM or local copy of Linux)

Install prereqs:

1. `sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev git`

Make build directory

2. `cd ~`
3. `mkdir riscv && cd riscv`
4. `git clone https://github.com/riscv/riscv-gnu-toolchain`
5. `cd riscv-gnu-toolchain`

Make install directory

6. `sudo mkdir /opt/riscv`
7. `sudo chmod 777 /opt/riscv`

Add install directory to your PATH

```
8. echo "export PATH=\$PATH:/opt/riscv/bin" >> ~/.profile
9. source ~/.profile
```

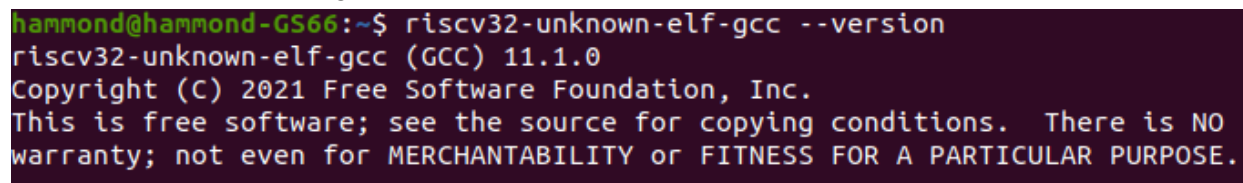
Configure and run the build (**Important: Make sure you copy these flags exactly!**)

```
10. ./configure --prefix=/opt/riscv --with-arch=rv32i
    --with-abi=ilp32
11. make
12. sudo ldconfig
```

Test the build

```
13. riscv32-unknown-elf-gcc --version
```

You should see something like this:



```
hammond@hammond-GS66:~$ riscv32-unknown-elf-gcc --version
riscv32-unknown-elf-gcc (GCC) 11.1.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Figure 1: Testing the compiler by getting the version

Your installation is now working! You will not need to repeat these steps unless you uninstall the toolchain.

You will now need to download a “template” for the assembly project that you’ll be making. I have provided such a template on brightspace. Download and unzip the folder ‘standalone\_asm.zip’. You will see it contains a “main.s” which is largely empty, a Makefile, and a special *linker* file called “nyu-6463-rv32i.ld”




Name	Size
 main.s	294 bytes
 Makefile	2.3 kB
 nyu-6463-rv32i.ld	1.2 kB

Figure 2: The assembly project template

The linker file is used to *place* your variables, instructions etc at the correct memory addresses. This includes for instance the memory address 0x01000000 that we mentioned earlier. To function, the assembler relies on the special assembler directives in Listing 1 (the lines beginning with periods). Essentially we ask in the linker script to start the code beginning with “reset\_handler” at 0x01000000.

To generate our program, we enter our assembly instructions in main.s. Then, run “make” on the command line. You should see something like this:

```
hammond@hammond-G566:~/riscv/programs/nyu-6463-rv32i-program/standalone_asm$ make
riscv32-unknown-elf-gcc -x assembler-with-cpp -c -O0 -Wall -fmessage-length=0 -march=rv32i -mabi=ilp32
-mcmodel=medlow -I./ main.s -o main.o
riscv32-unknown-elf-gcc main.o -Wall -Wl,--no-relax -Wl,--gc-sections -nostdlib -nostartfiles -lc -lgc
c --specs=nosys.specs -march=rv32i -mabi=ilp32 -mcmodel=medlow -T../common/nyu-6463-rv32i.ld -o main.e
lf
riscv32-unknown-elf-objcopy -S -O binary main.elf main.bin
riscv32-unknown-elf-size main.elf
   text    data     bss     dec      hex filename
   16       0    1024    1040     410 main.elf
riscv32-unknown-elf-objcopy -S -O ihex main.elf main.hex
hexdump -ve '1/4 "%.8x\n"' main.bin > main.mem
riscv32-unknown-elf-objdump --syms -d main.elf > main.dump
```

*Figure 3: Running ‘make’ on the assembly project*

As you can see, the assembly file is assembled into several final files, including a “main.elf” and a “main.bin”.

The Makefile also generates a memory file ready for use with our FPGA, called “main.mem”.

This text file is formatted to make it easier to use with our programming code! Let’s see what it looks like:

```
00100093
00200113
002080b3
ffdf06f
```

*Listing A.1: main.mem*

Looks just like the binary string we produced at the end of Option 1!

Whichever method you used, we’re now ready to try and load the binary code into the processor. Let’s do this in Part B.

## Part B: Loading binary code

This part of the guide assumes that you have constructed your instruction memory layout according to the following:

- (a) It is 32 bits wide and is word addressed.
- (b) The instruction memory array goes from index 0 up to ROM\_LENGTH\_WORDS - 1.

E.g. (Verilog)

```
reg [31:0] rom_words [0:`ROM_LENGTH_WORDS-1];
...
wire [31:0] addr_word = (addr[`ROM_ADDR_BITS-1:0]>>2);

always @(posedge clk) begin
    instr_out <= rom_words[addr_word];
end
```

*Listing B.1: Example Memory Code (Verilog)*

E.g. (VHDL)

```
type instr_rom is
    array(0 to ROM_LENGTH_WORDS-1)
    of std_logic_vector(32 downto 0);
signal rom_words: instr_rom;
signal addr_word: std_logic_vector(31 downto 0) := x"00000000";
...

addr_word(ROM_ADDR_BITS-3 downto 0) <=
    addr(ROM_ADDR_BITS-1 downto 2);

process(clk)
begin
    if rising_edge(clk) then
        instr_out <=
            rom_words(to_integer(unsigned(addr_word)));
    end if;
end process;
```

*Listing B.2: Example Memory Code (VHDL)*

Let's now look at how to load the code into our ROMs.

## 1. Manually

The first option involves creating ROMs as you previously have in Verilog/VHDL code. For instance, the original SKEY block in the “Simple Function” RC5 was a ROM.

E.g. (Verilog) - To load the instructions you can simply specify them in the initial block.

```
initial begin
    rom[0] <= 32'h00100093; //addi x1, x0, 1
    rom[1] <= 32'h00200113; //addi x2, x0, 2
    rom[2] <= 32'h002080b3; //loop: add x1, x1, x2
    rom[3] <= 32'hffdf06f; //j loop
end
```

*Listing B.3: Manual Instantiation of Memory Contents (Verilog)*

E.g. (VHDL) - Here we do not specify the addresses, VHDL works it out on its own.

```
signal rom: instr_rom := (x"00100093", x"00200113", x"002080b3",
x"hffdf06f", others => (others => 'X'));
```

*Listing B.4: Manual Instantiation of Memory Contents (VHDL)*

## 2. Using readmemh (Verilog)

Alternatively, once you have generated your program memory file (either by hand or using the assembler), you can load it in as a resource similarly to how we have loaded traces for testbenches previously.

The steps are as follows. Firstly, you can copy and paste the contents into a new text file in Xilinx Vivado which you save in your project directory as main.mem, or you can use a program like FileZilla to upload the file.

Once the file exists, include this file in your Verilog project sources by right clicking on the Design Sources, selecting [Add Sources...], [Next>], [Add Files], and selecting the mem file as shown:

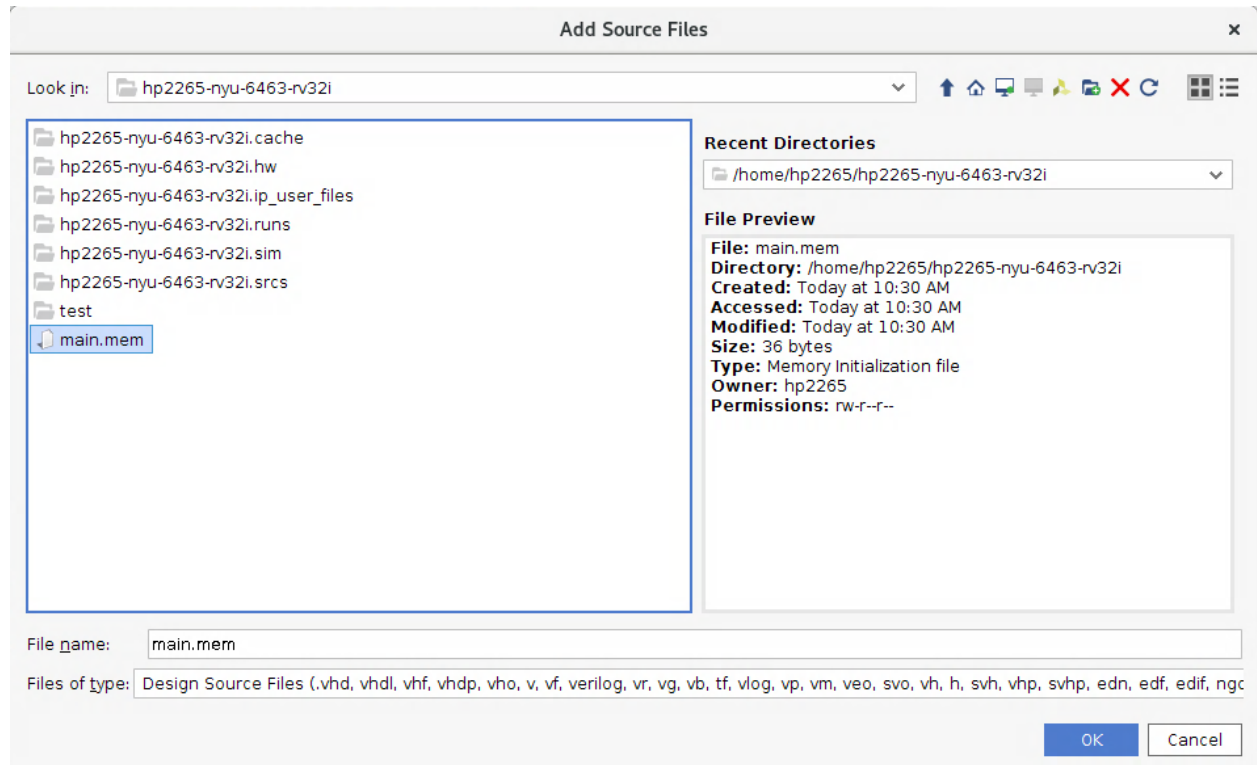


Figure 4: Importing the memory file

Remember, this is the same steps used when adding a text file with test traces for testbenches (I showed how to do this in my Oct 13 Office Hour video recording), except it is of type “memory file” and we add it to Design Sources and not Simulation Sources.

If you have done the above steps correctly, it should appear under BOTH your Design Sources and Simulation Sources as a “memory file”, as shown:



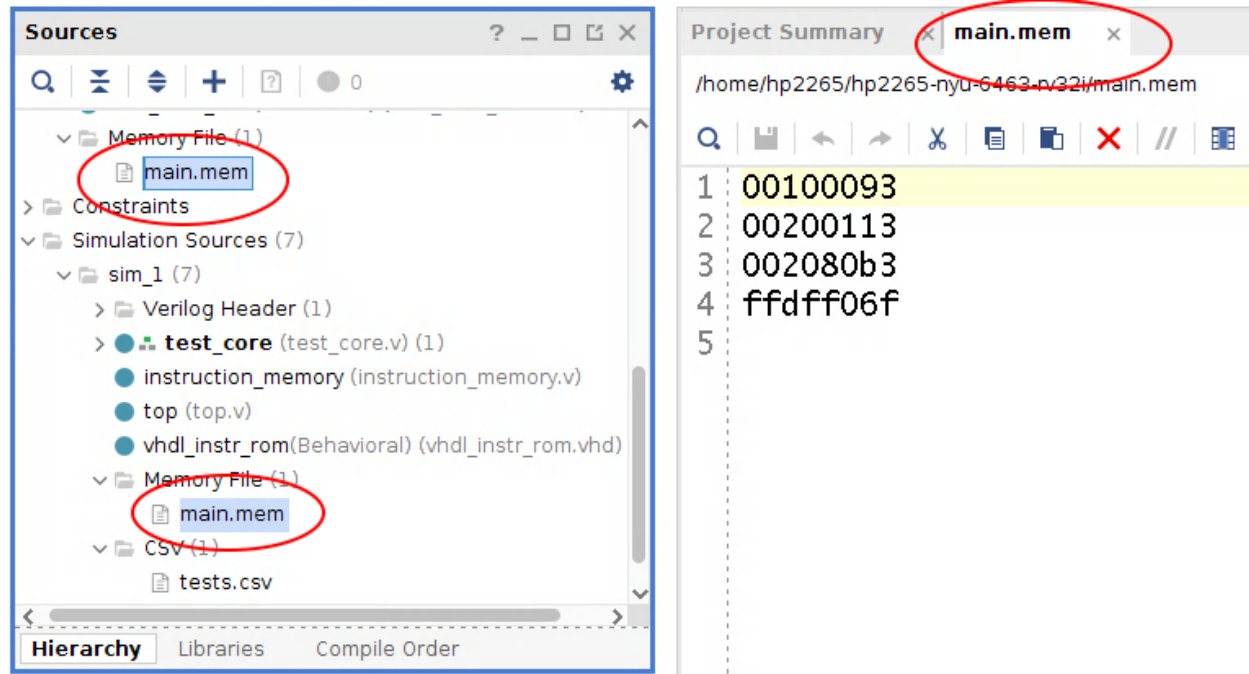


Figure 5: The memory file, successfully imported.

Now we can use the special \$readmemh function in Verilog:

```
reg [31:0] rom_words [0:`ROM_LENGTH_WORDS-1];
initial begin
    $readmemh("main.mem", rom_words);
end
```

Listing 8: \$readmemh() in Verilog

If we now simulate the instruction memory, we should be able to see that the memory is loaded correctly:

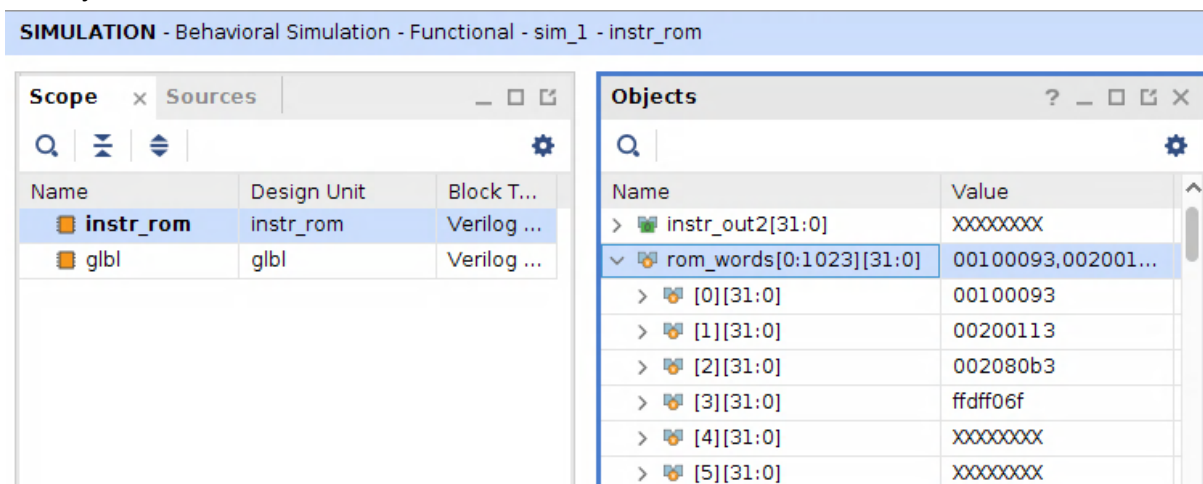


Figure 6: Successful simulation of the instruction memory

Looks good - note that the contents beyond our memory file are XX due to their absence from the file!

### 3. Using textio (VHDL)

This one is a little trickier. VHDL doesn't have an easy function like readmemh, so we need to load in the memory files line by line (as we do when there is a testbench using textio in VHDL). We can do this in synthesizable code by using a *function*. We haven't taught Functions in this course, but you can consider them as special blocks of code, more akin to software than hardware. In a sense they can be thought of as your own custom extensions to VHDL.

In this case, we define a function which can read the "main.mem" file from Listing 3. You first need to add this file to your project using the steps from the beginning of Option 2, Figures 4 and 5.

Now we will write the function. You should recognize the textio syntax from the testbenches you have designed.

Insert the following code (from Listing 4) in your architecture declaration area **with your instr\_rom type declaration**. It might be wise to also consider moving these to a custom package which you import, which would keep your project tidier and also allow you a place to store constants.

```
type instr_rom is array(0 to ROM_LENGTH_WORDS-1) of
std_logic_vector(31 downto 0);
-- Read a *.hex file
impure function instr_rom_readfile(FileName : STRING) return
instr_rom is
    file FileHandle      : TEXT open READ_MODE is FileName;
    variable CurrentLine : LINE;
    variable CurrentWord  : std_logic_vector(31 downto 0);
    variable Result       : instr_rom := (others => (others => 'X'));

begin
    for i in 0 to ROM_LENGTH_WORDS - 1 loop
        exit when endfile(FileHandle);

        readline(FileHandle, CurrentLine);
        hread(CurrentLine, CurrentWord);
        Result(i) := CurrentWord;
    end loop;

    return Result;
end function;
```

*Listing 9: VHDL custom instr\_rom\_readfile() function*

You will also need to include the following packages in the file that defines the above function:

```
use ieee.std_logic_1164.all;  
use ieee.numeric_std.all;  
use std.textio.all;  
use ieee.std_logic_textio.all;
```

*Listing 10: VHDL custom instr\_rom\_readfile() function required packages*

Now, where you declare the rom signal, use the function to define the default value:

```
signal rom: instr_rom := instr_rom_readfile("main.mem");
```

*Listing 11: Utilizing the VHDL custom instr\_rom\_readfile() function*

That's it! You're good to go! You should now be able to see the memory contents when you perform a simulation of your instruction memory, as per the screenshot in Figure 6.

Finally, for more details on inferring ROMs, you can see the following documentation by Xilinx:  
[https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2016\\_4/ug901-vivado-synthesis.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug901-vivado-synthesis.pdf)