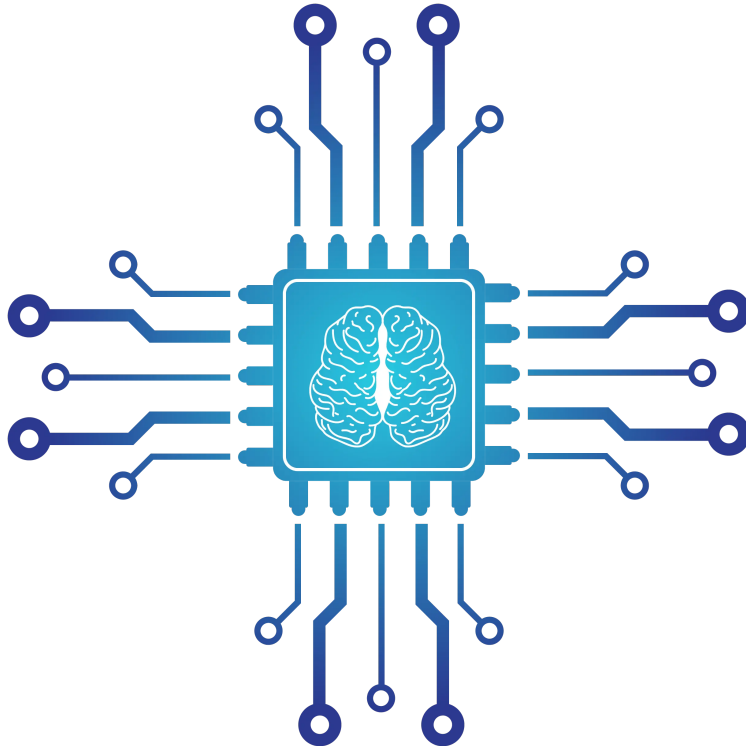




EE475L: Computer Architecture

Lab Manual

Author: xyz



Department of Electrical Engineering
University of Engineering and Technology, Lahore

Contents

1	Implementation and Testing of Functional Units of A Processor - Part 1	2
2	Implementation and Testing of Functional Units of A Processor - Part 2	5
3	Implementation of A Single Cycle Processor and UART	7
3.1	Single Cycle Processor	7
3.2	UART DataPath	8
4	RISCV GCC Toolchain and Testing the Single Cycle Processor	10
4.1	RISCV GCC Toolchain and Python Installation	10
4.2	Converting High Level Code to Machine Code	11
4.2.1	Assembly to Machine Code Conversion	11
4.2.2	C to Machine Code Conversion	12
4.2.3	Testing the Single Cycle Processor	13

Chapter 1

Implementation and Testing of Functional Units of A Processor - Part 1

1. Implement the ALU shown below in system verilog using QUESTA SIM software. Use the module name, input, output port names shown in the figure 1.1. The name of the module is shown in bold. The ALU should be able to perform AND, OR, ADD and SUBTRACT operations according to the **ALU operation** control signals given in figure 1.2. The output **Zero** should be 1 when subtraction of two operands give the result 0.

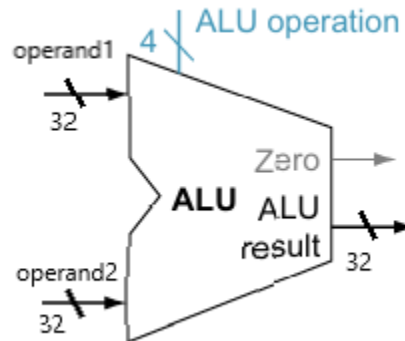


Figure 1.1: ALU

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Figure 1.2: ALU operation Signals

2. Implement the functional unit shown in figure 1.3 below which adds two 32-bit numbers. Remember to use the port names shown in the figure.

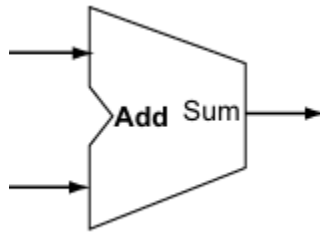


Figure 1.3: Branch Adder

3. Implement the functional unit shown in figure 1.5 which adds one operand to 4. Both the operands are 32-bit.

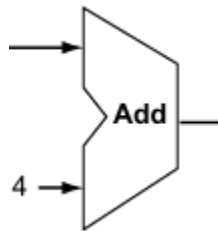


Figure 1.4: PC Adder

4. Implement the 2 to 1 mux shown in the figure below. The **MemtoReg** control signal selects one of the inputs to the mux.

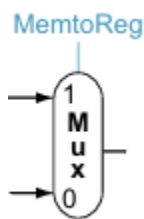


Figure 1.5: 2 to 1 Mux

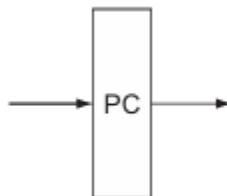
5. Write separate testbenches to test the three modules you created. Test for multiple, random inputs. Use the method `randomize()` to generate random inputs and a loop to generate multiple inputs.
6. Draw (in one space/file/area) and properly label all the above functional units using `draw.io` tool.

Chapter 2

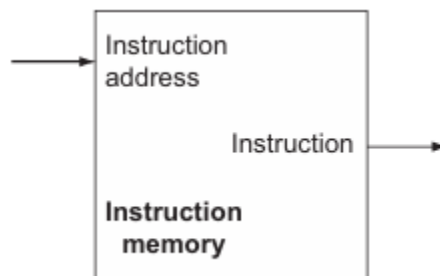
Implementation and Testing of Functional Units of A Processor - Part 2

In all the tasks below, draw and properly label the functional units in draw.io. Use a separate module for all the tasks. Write a randomized testbench to test the module. The testbench should use tasks. Read the textbook [Computer Organization and Design RiseV Edition](#) : section 2.5, section 2.7 and Section 4.3 (Pg. 261 - 269) for more details on the tasks given below. After reading, you should know what type of inputs go into the functional units and what type of outputs are expected to come out!

1. Write a code in system verilog to implement a 32-bit register having a active-high, asynchronous reset. This register is called program counter, also known as PC. Note: The inputs clock and reset are not shown in the figure.



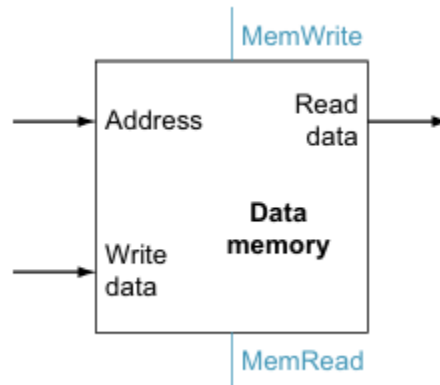
2. Write a code in system verilog to implement an instruction memory. It is a combinational circuit. Use `readmemh` command in the test bench to store data into the memory. Use `writememh` command in the testbench to verify the data stored inside the memory. Refer to lab notes for more details.



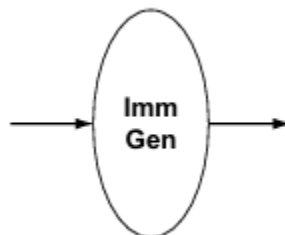
3. Write a code in system verilog to implement a register file. The register file should have 16 registers each of 32 bits. The write operation is edge-triggered. See the textbook for more details.



4. Write a code in system verilog to implement a 2kB, word-aligned, byte addressable data memory. Use `readmemh` command in the test bench to store data into the memory. Use `writememh` command in the testbench to verify the data stored inside the memory. Refer to lab notes for more details.



5. Write a code in system verilog to implement a immediate generator. Read Page 269, the elaboration section, for more details.



Chapter 3

Implementation of A Single Cycle Processor and UART

3.1 Single Cycle Processor

In the first part of this lab, you are required to code the datapath as well as the control part for a single-cycle processor shown in figure 3.1 to execute the `add`, `sub`, `and`, `or`, `lw`, `sw`, `beq` and `addi` instructions. The code for functional units of the datapath was written in the last two labs. In this lab you have to just connect the functional units like PC, Register File etc. and implement the control part.

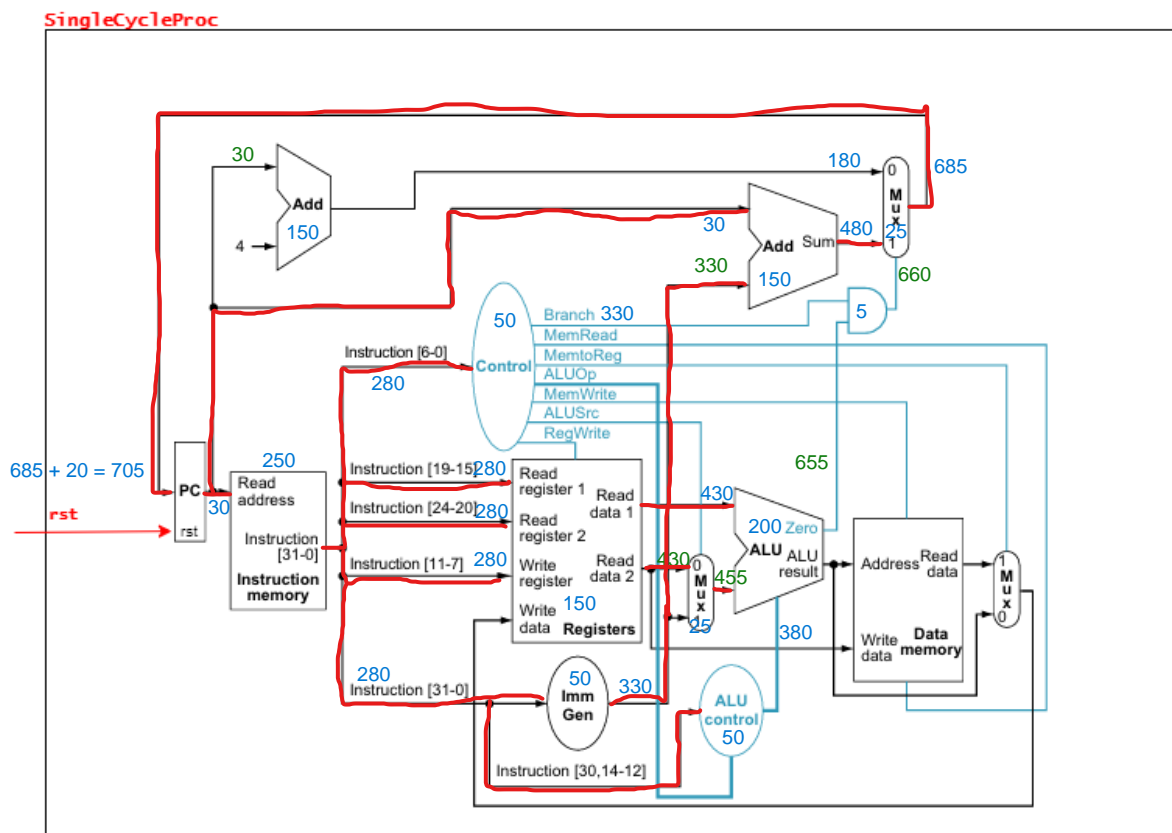


Figure 3.1: Datapath of A Single Cycle Processor

1. Draw the datapath shown in figure 3.1 using draw.io

2. Write code for the Control and ALU Control functional units. For more details see table 4.12 on page 270 and figure 4.26 on page 281 of the [textbook](#).
3. In a separate module named `SingleCycleProc`, instantiate all the required modules and make connections between the datapath elements or functional units shown in figure.
4. Modify the control and datapath so that the processor could execute the `addi` instruction. Refer to RiscV [cheat sheet](#) to determine the datapath and control for `addi`.
5. Create a testbench and connect it to the `SingleCycleProc` instance as shown in figure. The testbench and `SingleCycleProc` must be instantiated in the module named `top` as shown in figure 3.2. The `rst` signal asynchronously resets the PC you coded in the last lab i.e. the output of the PC should be zero upon reset. In the next lab, you will write the complete testbench to test the processor.

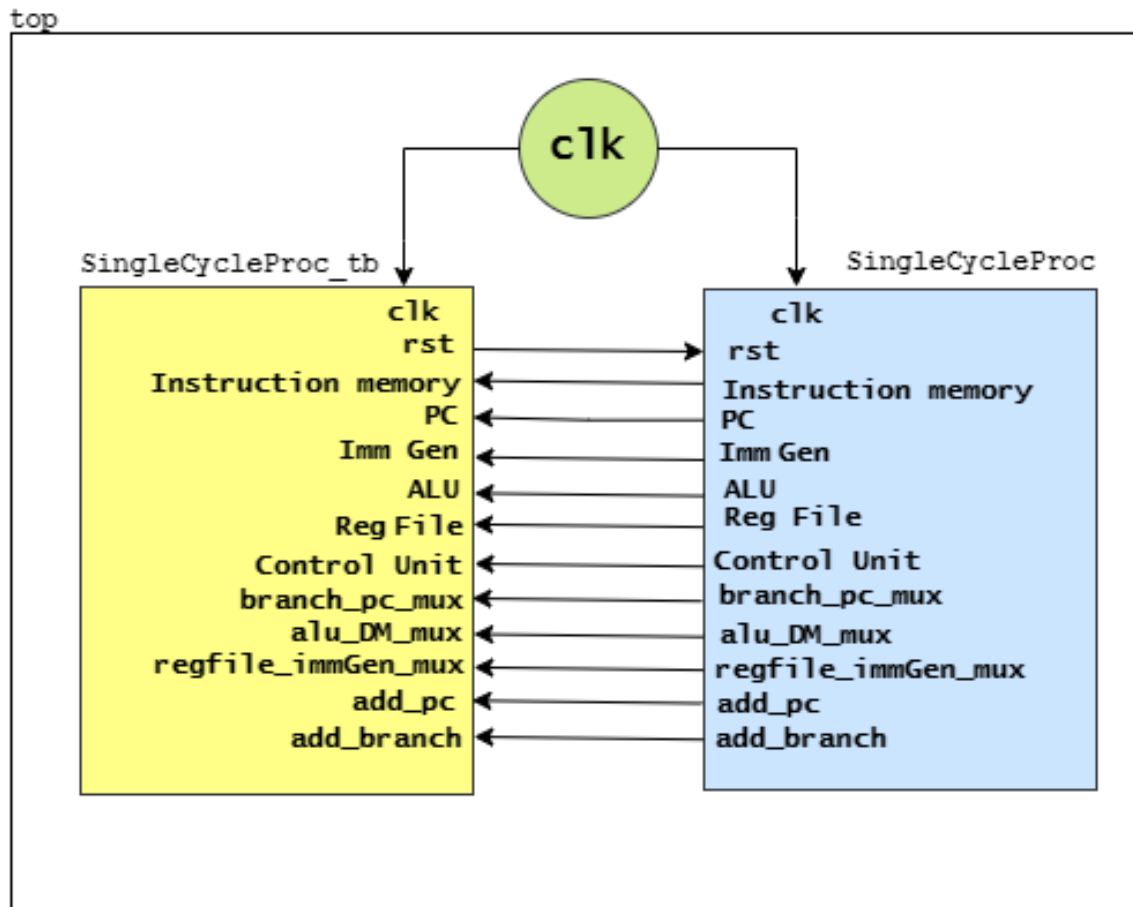


Figure 3.2: The Top Module

3.2 UART DataPath

In the second part of this lab, you will create a datapath to implement the UART protocol for transmitter and the receiver. To figure out the datapath, read [theory document](#) and the [datasheet](#) of the `tm4c123` microcontroller. Write system verilog code to implement the datapath. Datapath should include the following registers.

1. UART-DATA
2. UART-CTL for:

- (a) Enabling Tx and Rx
 - (b) Setting number of stop bits (0 for one stop bit, 1 for two stop bits)
 - (c) Parity choice (0 for even, 1 for odd)
3. UART-BAUD (only the integer part)
 4. UART-STATUS for monitoring errors.

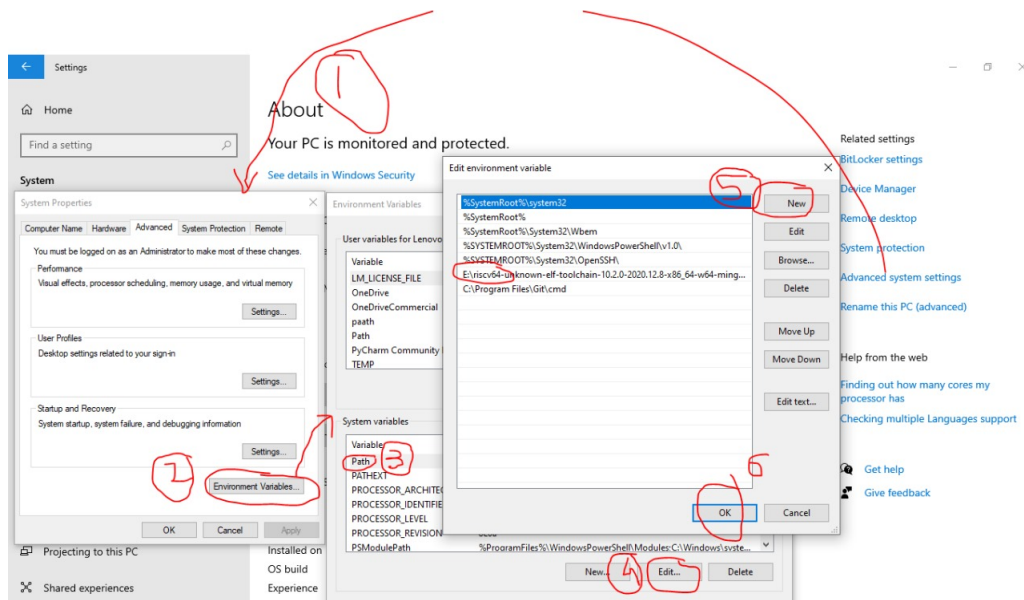
Chapter 4

RISCV GCC Toolchain and Testing the Single Cycle Processor

In this lab, you will learn to use a tool called GNU Compiler Collection (GCC) suite that converts C or Assembly code to machine code. You will also test the single cycle processor you created in the last lab. Follow the steps below to get started (for Windows OS only):

4.1 RISCV GCC Toolchain and Python Installation

1. Download the toolchain using this [link](#).
2. Extract the zip file to any of the local drives on your computer.
3. Add the path of the bin folder inside the extracted folder to the system variable called Path by right clicking **This PC** → **Properties** → **Advanced System Settings** → **Environment Variables** → click **Path** in system variables box → **edit** → **New** → paste the bin folder path and click **OK**.



4. Open the command prompt and run the command `riscv64-unknown-elf-gcc --help`
5. If you get the following output it means you installed the suite correctly.

```
Usage: riscv64-unknown-elf-gcc [options] file...
Options:
  -pass-exit-codes      Exit with highest error code from a phase.
  --help                Display this information.
  --target-help          Display target specific command line options.
```

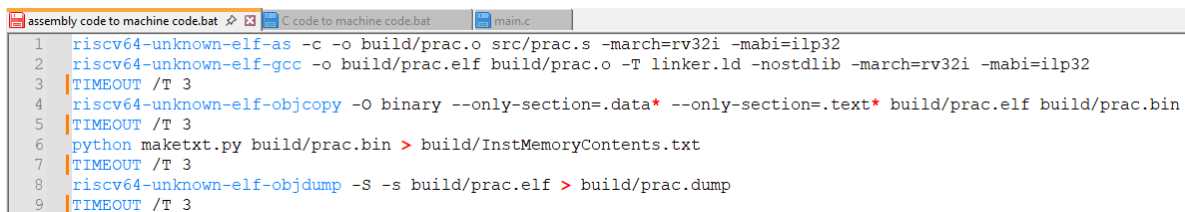
6. Install latest version of Python using this [link](#). Watch this [video](#) for more clarity.

4.2 Converting High Level Code to Machine Code

The following steps will enable you to convert a code written in Assembly Language to the machine code.

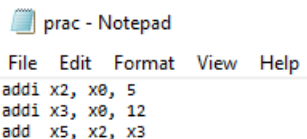
4.2.1 Assembly to Machine Code Conversion

1. All the required files for this task are available [here](#)
2. In any local drive, create a folder named **xyz** and then create two sub folders in it and name them **src** and **build** respectively.
3. Place the file **linker.ld** (see step 1) inside the folder **xyz** but outside the subfolders.
4. Place the file **startup.s** inside the **src** folder.
5. Create a new text file and type the following. Take care of the spaces.



```
1 riscv64-unknown-elf-as -c -o build/prac.o src/prac.s -march=rv32i -mabi=ilp32
2 riscv64-unknown-elf-gcc -o build/prac.elf build/prac.o -T linker.ld -nostdlib -march=rv32i -mabi=ilp32
3 TIMEOUT /T 3
4 riscv64-unknown-elf-objcopy -O binary --only-section=.data* --only-section=.text* build/prac.elf build/prac.bin
5 TIMEOUT /T 3
6 python maketxt.py build/prac.bin > build/InstMemoryContents.txt
7 TIMEOUT /T 3
8 riscv64-unknown-elf-objdump -S -s build/prac.elf > build/prac.dump
9 TIMEOUT /T 3
```

6. Save the file by the name **AsmToMachine.bat** in **xyz** folder outside the subfolders.
7. Create a new text file (notepad) and write the following Assembly code.



```
File Edit Format View Help
addi x2, x0, 5
addi x3, x0, 12
add x5, x2, x3
```

8. Save the file by the name **prac.s** and place it in the **src** folder.
9. Double click **AsmToMachine.bat** file. If any error, resolve it.
10. Go to **build** folder. Open the file **prac.dmp** and **prac.txt** to see the machine code of the assembly code you wrote.

```

prac - Notepad
File Edit Format View Help

build/prac.elf:      file format elf32-littleriscv

Contents of section .text:
 1000 13015000 9301c000 b3023100      ..P.....1.
Contents of section .riscv.attributes:
 0000 41190000 00726973 63760001 0f000000  A....riscv.....
 0010 05727633 32693270 3000      .rv32i2p0.

Disassembly of section .text:

00001000 <.text>:
   1000:      00500113          li      sp,5
   1004:      00c00193          li      gp,12
   1008:      003102b3          add     t0,sp,gp

```

Figure 4.1: The prac.dmp File

The following steps will enable you to convert a code written in C Language to the machine code.

4.2.2 C to Machine Code Conversion

1. Create a new text file and type the following. Take care of the spaces.
2. Save the file by the name CToMachine.bat in xyz folder outside the subfolders.
3. Create a new text file (notepad) and write the following C code.

```

main - Notepad
File Edit Format View Help

int main(void) {
    int f, g, h, i, j;
    g = 3;
    h = 4;
    i = 5;
    j = 6;
    f = (i + j) - (g + h);
}

```

4. Save the file by the name main.c and place it in the src folder. Ensure startup.s file is there otherwise the code will not compile.
5. Double click CToMachine.bat file. If any error, resolve it.
6. Go to build folder. Open the file main.dump and main.txt to see the machine code of the assembly code you wrote.

```

main - Notepad
File Edit Format View Help
Contents of section .vector_table:
1000 17f1ff7f 13010140 ef008000 6f000000 .....@....0...
Contents of section .text:
1010 130101fd 23268102 13040103 93073000 ....#&.....0.
1020 2326f4fe 93074000 2324f4fe 93075000 #&....@.#$....P.
1030 2322f4fe 93076000 2320f4fe 032744fe #"....'.# ...'D.
1040 832704fe 3307f700 8326c4fe 832784fe .'.3....&....'..
1050 b387f600 b307f740 232ef4fc 93070000 .....@#.....
1060 13850700 0324c102 13010103 67800000 .....$......g...
1070 6ff01ff9                                     o...
Contents of section .comment:
0000 4743433a 20285369 46697665 20474343 GCC: (SiFive GCC
0010 2d4d6574 616c2031 302e322e 302d3230 -Metal 10.2.0-20
0020 32302e31 322e3829 2031302e 322e3000 20.12.8) 10.2.0.
Contents of section .riscv.attributes:
0000 411b0000 00726973 63760001 11000000 A....riscv.....
0010 04100572 76333269 32703000 ...rv32i2p0.

Disassembly of section .text:

00001010 <main>:
1010:      fd010113          addi    sp,sp,-48
1014:      02812623          sw      s0,44(sp)
1018:      03010413          addi    s0,sp,48
101c:      00300793          li      a5,3
1020:      fef42623          sw      a5,-20(s0)
1024:      00400793          li      a5,4
1028:      fef42423          sw      a5,-24(s0)
102c:      00500793          li      a5,5
1030:      fef42223          sw      a5,-28(s0)
1034:      00600793          li      a5,6
1038:      fef42023          sw      a5,-32(s0)
103c:      fe442703          lw      a4,-28(s0)
1040:      fe042783          lw      a5,-32(s0)
1044:      00f70733          add     a4,a4,a5
1048:      fec42683          lw      a3,-20(s0)
104c:      fe842783          lw      a5,-24(s0)
1050:      00f687b3          add     a5,a3,a5
1054:      40f707b3          sub     a5,a4,a5
1058:      fcf42e23          sw      a5,-36(s0)
105c:      00000793          li      a5,0
1060:      00078513          mv      a0,a5
1064:      02c12403          lw      s0,44(sp)
1068:      03010113          addi    sp,sp,48
106c:      00008067          ret

00001070 <_start>:
1070:      f91ff06f          j        1000 <reset_handler>

```

Figure 4.2: The main.dmp File

4.2.3 Testing the Single Cycle Processor

Now you will be testing the single cycle processor you created in the last lab. Write a test bench module that loads the following instructions into the instruction memory.

```

prac - Notepad
File Edit Format View Help
addi x2, x0, 5
addi x3, x0, 12
addi x7, x3, -9
or x4, x7, x2
and x5, x3, x4
add x5, x5, x4
beq x5, x7, end
sub x4, x4, x2
lw x1, 16(x0)
end: sw x2, 0x20(x3)
done: beq x2, x2, done

```

Test each instruction by comparing the expected outputs of EACH functional block with the actual outputs. See figure 3.2 for more details on the testbench module. Use the following table template (for every instruction) to write your test results.

Functional Unit	Expected Output (in Hex)	Actual Output (in Hex)
PC		
Instruction Memory		