Implementation of Superscalar Processor Design | Self Project

GUNANJAN REDDY K

23M1148 EE7

ELECTRICAL ENGINEERING

IIT BOMBAY

https://github.com/GUNANJAN/Superscalar_Processor_Design-.git

---

The datapath consists of Two 16 bit ALUs, One 16 bit priority encoder that gives a 16 bit output and 3bit register address. There are two Sign Extenders SE6 and SE9 for 6 and 9 bit inputs giving 16 bit outputs. There are two left bit shifters Lshifter7 and Lshifter1 which respectively shift the input by 7 and 1 bit (s) to the left appending 0s to the right giving a 16 bit output. There are four temporary registers TA, TB, TC, TD, where TA, TB and TC are 16bit and TD is 3bit. The instructions and information regarding the 27 states of the FSM can be found in the file FSM States.pdf and the state transition flows of the Finite State Machine can be found in the file State Transition Diagram.pdf. The control decode logic for each instruction can be found in the file Decoding Logic.pdf.

The control words for each of the states can be found in the file Control Words.pdf The project folder also has attached the hardware descriptions of the various components in VHDL. The toplevel entity contains the Datapath and the FSM. GHDL and GTKWave were used to test and debug all the various instructions. Screenshots of the wave viewer(GTKWave) are placed in the waveforms folder. The instructions listed in waves.txt were run by placing them in memory and the waveforms were captured.

Instruction Set Architecture

The **IITB-RISC-22** has a **Turing Complete ISA** and the **17** instructions supported by the **IITB-RISC-22**, their assembly notation and encoding can be found in the Problem Statement.

We noticed that LHI has the same encoding as ADI hence we decided to assign LHI an opcode of 0011

Software Requirements

- GHDL
- GTKWave
- Python 3.x

Assembler

An assembler for the **IITB-RISC-22** was designed in Python to convert any input program stored as .asm into a sequence of machine level 16 bit word instructions stored in source.bin . The source code for it can be found in ./assembler.py. The assembler also provides support for both **inline** and **out of line comments** for documentation to be present in the .asm file.

To assemble the code for a file called code.asm in the same directory as assembler.py can be done in the following way.

python assembler.py code

Bootloader

An software emulated bootloader for the **IITB-RISC-22** was designed in Python to dump the binary file into the memory of the **IITB-RISC-22**. The source code for it can be found in ./bootloader.py . It takes as input the binary file source.bin and loads the instructions into the file ./Memory.vhdl.

It also has an additional feature to initialize values into the memory locations 61 and 62 of the memory before the start of processor execution. This functionality was introduced to allow initializing the two numbers to be multiplied in the benchmark code, which can be found later below.

To load the binary file source.bin into memory without initializing the values of memory location 61 and 62 do the following

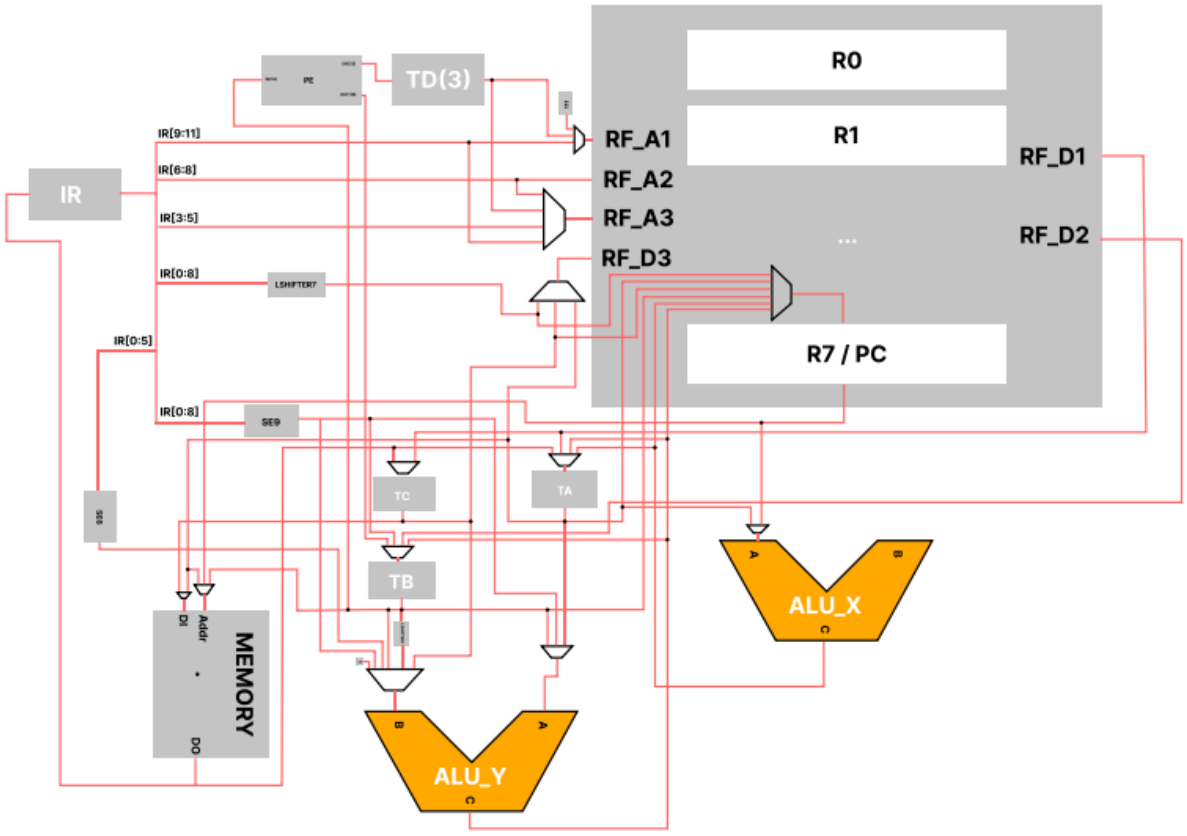
python bootloader.py code

To load the binary file source.bin into memory, initializing the values of memory location 61 and 62 to m1 and m2, where m1, m2∈[0,65535] are decimal numbers do the following

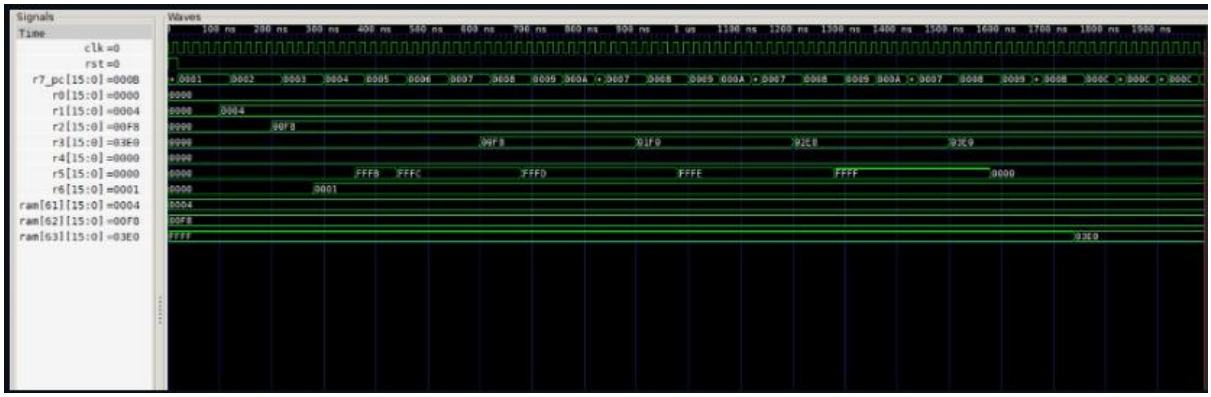
python bootloader.py m1 m2

Preliminary Testing of Instructions

- The instructions listed in tested.txt were run by placing them in memory and the results seen were as expected by the ISA and design specifications.


R0
R1
RF_A1
RF_A2
RF_A3
RF_D3
RF_D1
RF_D2
R7 / PC
TD(3)
IR
IR[9:11]
IR[6:8]
IR[3:5]
IR[0:8]
IR[0:5]
LSHIFTER7
SE9
TC
TA
TB
ALU_X
ALU_Y
MEMORY
Addr
DI
DO

Results:


Signals
Waves
clk =0
rst =0
r7_pc[15:0] =000B
r0[15:0] =0000
r1[15:0] =0004
r2[15:0] =00F8
r3[15:0] =03E0
r4[15:0] =0000
r5[15:0] =0000
r6[15:0] =0001
ram[61][15:0] =0004
ram[62][15:0] =00F8
ram[63][15:0] =03E0