Atlas 5900x
User Manual

Matthew Soltys and Joshua Hizgiaev

December 3rd, 2023

I pledge my honor that I have abided by the Stevens Honor System.

1 Group Responsibilities

1.1 Joshua

Joshua's responsibilities were designing the CPU micro-architecture, building the Micro-CPU circuit in Logisim, determining how registers will be supported, what instructions will be supported, and designing and implementing the machine code onto the CPU. Due to the fact that the CPU circuit and how the machine code was structured are closely related, it is only logical that these would be developed together. The process requiring staring at a textbook for hours trying to make the most efficient, practical, and simple design of the machine code whilst also learning the semantics of Logisim. Additionally, designing the machine code required comprehensive research on control signal and ALU design.

1.2 Matthew

Matthew's responsibilities were designing and implementing the small and efficient Atlas Assembly assembler in Python. This includes the format of the assembly language, implementing our comprehensive error checker, parsing instructions, and introducing support for `.atlas` file extensions. Developing the assembler in Python allowed to keep our Assembler clean, small, and easy to understand. Designing the format of the assembly language and implementing the assembler naturally go hand in hand, so it is essential they were streamlined together. Making the assembler required learning how Python handles file parsing, string, hex digit, and binary manipulation and making sure every instruction encoding was correctly 24 bits in length.

2 The Assembler

2.1 Assembler Details

The assembler, `assembler.py`, is written in Python. It specifically takes in files with `.atlas` file extensions, such as `test.atlas`, for example. This file extension signifies that a file is written in the Atlas Assembly Language.

The assembler parses the `.atlas` file it receives, and returns an image file to be used in the circuit that contains the assembly instructions converted to hexadecimal. The image file is not have a `.txt` extension, rather its just a general plain text file.

`assembler.py` contains comprehensive error checking, verifying that:

- The file name selected exists and is within the directory
- The file extension is `.atlas`
- All the instructions written are valid instructions (ADD, SUB, LDR, or STR)
- All registers used are valid registers (X0-X3)
- Any immediate numbers used are signed 8-bit integers
- Both ADD and SUB arguments are at least 0 if an integer is present
- Checks that all instructions begin with a ;

If any errors are found, ValueErrors are raised. The program will be terminated and the user will receive an error message.

2.2 Using the Assembler

Once you have a `.atlas` file to run the assembler with, place the `assembler.py` file in the same file directory as your created `.atlas` file. For later convenience, it would be best to have `project2.circ` inside this directory as well, but it is not required.

As it is a python file, there are several ways to run it:

1) Windows Users can open the directory in File Explorer and double left click on `assembler.py`. You will then be receive the following prompt:

Please input file name you want to use with the assembler:

Type in the full name of your `.atlas` file, ex: `test.atlas`. Press enter. You should now have a file named `instruction_memory` generated in the same directory you are in. This is the image file which will be loaded into the CPU.

2) Using an IDE, such as Visual Studio Code, you can select "Open Folder" and open the folder where `assembler.py` and your `.atlas` file reside. Select `assembler.py` and run it.You will then be receive the following prompt:

Please input file name you want to use with the assembler:

Type in the full name of your `.atlas` file, ex: `test.atlas`. Press enter. You should now have a file named `instruction_memory` generated in the same directory you are in. This is the image file which will be loaded into the CPU.

2.3 Loading to the Memory

After you have generated the image file, you can load into the `project2.circ` file.

Select *Simulate*, and then *Reset Simulation*

Select *Simulate*, and then make sure *Auto-Propagate* is activated

OPTIONAL: Select *Simulate*, and then *Auto-Tick Frequency*. Choose your preferred frequency. The program will be set by default to 1.0 Hz.

Right click on the RAM object near the left edge of the screen, below the CPU Title. Select *Load Image*. Find and select `instruction_memory`. Click Open.

Select *Simulate*, and then make sure *Auto-Tick Enabled* is activated

The program will now be running.

3 The CPU

The CPU design a simple one clock cycle design in which for every 1 clock cycle, 1 instruction within the instruction memory will be executed, every instruction within the memory is represented using hexadecimal digits generated from its binary encoding. The encoding is in total 24 bits. With 3 miscellaneous bits append to the far right of the encoding to make the number of encoding bits a multiple of 8.

3.1 Control Signals

There is a total of 7 control signals (7 bits) that the CPU must take into account for each instruction (in order from MSB to LSB):

1. *Reg2Loc*: This control signal will tell the CPU whether or not we choose to use the `Rm` value for our instruction, `1` for yes, `0` for no.
2. *ALUop*:

   This control signal will either signify the ALU of the CPU to perform subtraction (`1`) or addition (`0`).
3. *RegWrite*:

   This control signal signifies if the register will have data written into it, `1` for yes, `0` for no.
4. *ALUsrc*:

   This control signal signifies if the instruction is using `imm8`, `1` for yes, `0` for no.

5. *MemRead*:

   This control signal signifies whether the instruction requires reading from the memory, this will only be `1` if `LDR` instruction is used, `0` otherwise.

6. *MemWrite*:

   This control signal signifies whether the instruction requires writing to the memory, this will only be `1` if `STR` instruction is used, `0` otherwise.

7. *MemToReg*

   This control signal signifies whether the instruction requires writing data from memory to register, this will only be `1` if `LDR` instruction is used, `0` otherwise.

3.2 Registers & Supported Operations

There is a total of 4 general purpose registers available within the CPU labeled from X0, X1, X2, and X3. Each register can store 8 bits of data (1 byte of data). All registers represent the data in terms of hexadecimals.

The CPU's ALU supports only 2 arithmetic operations addition and subtraction. The main purpose for a small amount of arithmetic instructions is because the `LDR`, `STR` only require either addition or subtraction to function properly.

3.3 Memory & Program Counter

Our instruction memory size allows for 24 bits (3 bytes) of data for each instruction and up to 256 instructions are allowed per `.atlas` file. Our RAM can also hold 1 byte of data per address for 256 addresses as that is how much data is allowed within a register.

The program counter simply uses 8 bits (1 byte) of data representing the current memory address we are on for instruction execution. It is responsible for essentially indexing our instruction memory and tells the CPU what instruction to execute next. The program counter will be incremented by 1 per instruction using an adder and constant.

4 The Instructions

4.1 Machine Code

Each instruction contains 24 bits, of which the breakdown is:
The first (beginning with MSB) 7 bits representing the opcode
The next 8 bits representing the value of any entered Imm8
The next 2 bits representing the ID of Rn
The next 2 bits representing the ID of Rm
The next 2 bits representing the ID of Rd/Rt
The final (ending with LSB) 3 bits representing miscellaneous bits used to make the machine code 24 bits
Any factors that are irrelevant (such as if an immediate is not used in the instruction) will result in the part of the machine code representing that to be all 0s.

24-18	17-10	9-8	7-6	5-4	3-1
Opcode	Imm8	Rn	Rm	Rd/Rt	Misc

The breakdown of the opcode is as follows (leftmost bit representing MSB):

24	23	22	21	20	19	18
Reg2Loc	ALUop	RegWrite	ALUsrc	MemRead	MemWrite	MemToReg

4.2 ADD

With Registers: `;ADD;Rd;Rn;Rm`
With Immediates: `;ADD;Rd;Rn;Imm8`

Rd refers to the destination register
Rn refers to register used as source 1
Rm refers to register used as source 2
Imm8 refers to a signed 8-bit integer

The ADD instruction takes in two values as inputs, a register Rn and a signed 8-bit immediate Imm8/second register Rn, and puts the sum of the contents of the two inputs into the destination register Rd.

All instruction notation and register IDs must be full uppercase, Atlas Assembly is case sensitive.

Note: Any immediate used in the ADD instruction be >= 0

Example: `;ADD;X0;X1;X2` will add X1 and X2, and then put the result in X0. The binary for this exact instruction would be 101000000000000011000000.

The binary encoding of ADD is:

Bit	24	23	22	21	20	19	18
Signal	Reg2Loc	ALUop	RegWrite	ALUsrc	MemRead	MemWrite	MemToReg
Value	1	0	1	0/1*	0	0	0

* ALUsrc will be 1 if an Imm8 is chosen, and 0 otherwise

4.3 SUB

With Registers: `;SUB;Rd;Rn;Rm`
With Immediates: `;SUB;Rd;Rn;Imm8`

Rd refers to the destination register
Rn refers to register used as source 1
Rm refers to register used as source 2
Imm8 refers to a signed 8-bit integer

The SUB instruction takes in two values as inputs, a register Rn and a signed 8-bit immediate Imm8/second register Rm, and puts the difference of Rn and Rm/Imm8 (Rn - Rm/Imm8) of the contents of the two inputs into the destination register Rd.

All instruction notation and register IDs must be full uppercase, Atlas Assembly is case sensitive.

Note: Any immediate used in the SUB instruction be >= 0

Example: `;SUB;X0;X1;X2` will subtract X2 from X1, and then put the result in X0. The binary for this exact instruction would be 111000000000000011000000.

The binary encoding of SUB is:

Bit	24	23	22	21	20	19	18
Signal	Reg2Loc	ALUop	RegWrite	ALUsrc	MemRead	MemWrite	MemToReg
Value	1	1	1	0/1*	0	0	0

* ALUsrc will be 1 if an Imm8 is chosen, and 0 otherwise

4.4 LDR

With Immediates: `;LDR;Rt;Rn;Imm8`

Rt refers to the target register
Rn refers to register used as source 1
Imm8 refers to a signed 8-bit integer

The LDR instruction takes in two values as inputs, a register Rn and a signed 8-bit immediate Imm8. It accesses Rn at an offset of Imm8, and loads the value at the address into Rt, the target register. Immediates in this case do not have to be >= 0, they can be negative integers.

All instruction notation and register IDs must be full uppercase, Atlas Assembly is case sensitive.

Note: If you do not want any offset from the specified Rn register, you must enter Imm8 as "0". You cannot leave the field blank in the instruction.

2nd Note: It is recommended for LDR to be used only after STR has been used, otherwise LDR will zero out a register.

Example: `;LDR;X0;X1;1` will load the value that is in X1 with an offset of 1 into X0. The binary for this exact instruction would be 00111010000000101000000.

The binary encoding of LDR is:

Bit	24	23	22	21	20	19	18
Signal	Reg2Loc	ALUop	RegWrite	ALUsrc	MemRead	MemWrite	MemToReg
Value	0	0/1*	1	1	1	0	1

* ALUop will be 0 if the Imm8 chosen is >= 0, and 1 otherwise

4.5 STR

With Immediates: `;STR;Rt;Rn;Imm8`

Rt refers to the target register
Rn refers to register used as source 1
Imm8 refers to a signed 8-bit integer

The STR instruction takes in two values as inputs, a register Rn and a signed 8-bit immediate Imm8. It accesses Rn at an offset of Imm8, and stores the value that is currently in Rt into that address. Immediates in this case do not have to be >= 0, they can be negative integers.

All instruction notation and register IDs must be full uppercase, Atlas Assembly is case sensitive.

Note: If you do not want any offset from the specified Rn register, you must enter Imm8 as "0". You cannot leave the field blank in the instruction.

Example: `;STR;X0;X1;1` will store the value that is in X0 into X1 with an offset of 1. The binary for this exact instruction would be 000101000000001010000000.

The binary encoding of STR is:

Bit	24	23	22	21	20	19	18
Signal	Reg2Loc	ALUop	RegWrite	ALUsrc	MemRead	MemWrite	MemToReg
Value	0	0/1*	0	1	0	1	0

* ALUop will be 0 if the Imm8 chosen is >= 0, and 1 otherwise