CP**uwu**: User Manual

John Graham & Emma Hodor

Last revised: December 3, 2022

1 About CPuwu

CPuwu was designed as a project for CS 382 (Computer Architecture & Organization) at Stevens Institute of Technology. The goal was to create a CPU with a custom architecture, machine code, and assembly language similar to ARM (which we were studying in class). CPuwu has a basic, sequential implementation, and uses an assembly language that is capable of multiple arithmetic and bitwise operations.

CPuwu is made up of 8 general-purpose registers, each of which can store 8 bits (1 byte) of data. These registers are referred to with the prefix `u`, and are numbered 0-7 (i.e. `u1`). This is very similar to ARM, except for the fact that we have a quarter of the amount of registers.

The arithmetic logic unit (ALU) can handle 8 types of operations (4 arithmetic, 4 bitwise) on positive, 8-bit integers. (**Note**: there is no capability of handling overflows, and operations that result in overflows will have unexpected results). Multiplexors are responsible for choosing which registers we read from, if we use an immediate or not, and if we need to read from memory. The output of these multiplexors depends on the machine code of each instruction (which will be detailed later). When required by an instruction, the results of the ALU calculations or the memory read will be written back to the designated register. CPuwu also supports reading and writing data from memory.

CPuwu also supports simple conditional and unconditional jump instructions, which are useful for creating loops and iterative programs. A series of multiplexors determines how the Program Counter (PC) is updated (depending on if we are conditionally jumping, jumping in general to another address in the instruction memory, or just moving on to the next instruction).

The machine code for our assembly language (called "uwusembly") requires 32 bits of data, and as such, the Instruction Memory holds 32 bits of data at each address. CPuwu also supports loading and storing values to Data Memory, which can only hold 8 bits of data at each address (since all of our registers and operations use 8-bit numbers). Values can be printed to LED displays (both in hexadecimal and decimal forms) to be displayed to the user for one clock cycle.

1.1 John's Responsibilities

John handled creating the structure and format of the assembly language, writing the assembler program, and creating this Instruction Manual. Developing the assembler in Python heavily relied on what the format of the language was. Grouping these tasks together would streamline both the development the assembler. Creating the language was easy, but the assembler took a lot of debugging and testing to make sure it was accurately and consistently translating the instructions into the correct binary encoding.

1.2 Emma's Responsibilities

Emma's responsibilities included constructing the circuit in Logisim, as well as determining the specifics of the machine code. Since the machine code and the circuit are so closely related, it made sense for these tasks to be handled together. This took a lot of debugging (and also some Logisim research!) to perfect and fully develop. New bits for the machine code were added as new instructions and operations were added, and this took a lot of careful planning to organize which bits of the machine code related to what purpose.

1.3 Teamwork!

The above descriptions just give an overall view of how the work was divided. We frequently helped each other with debugging our respective programs/circuits. Additionally, we asked each other for input on stuff like the purpose of certain bits in the machine code, and how to format the assembly language, as these components would affect how we would each go about our individual responsibilities. Teamwork and coordination was key in making sure that CPuwu, the assembler, and everything inbetween functioned as expected.

2 The Language: uwusembley

uwusembley is an earier-to-read assembly language. The syntax is clearer and more straightforward as to what each instruction does. For example, a common instruction might look like this:

```
u1 add u0 1
```

While it might not be clear at first, reading uwusembley is very simple. The register listed as the first argument (`u1`) is the destination register. The resulting data of this instruction will be written back to register `u1`. The mnemonic `add` denotes the operation being performed in this instruction (which in this case is addition). The two arguments following `add` are the operands that will be added together (in this case, the value stored in `u0` will be added with `1`. In summary, this instruction will add the value of `add` with `1`, and then store the resulting sum in `u1`.

Simple, right? Once you get used to it, uwusembley becomes very easy to use. Generally, instructions follow this format:

```
<dst> <mnemonic> <arg1> <arg2>
```

There are a few instructions that deviate from this convention, mainly because they either take less arguments or have more complicated functions than simple arithmetic.

2.1 Instructions

uwusembly has a limited, simple instruction set. However, it does everything that a basic assembly language needs to do, including arithmetic, bitwise, and memory operations. Most instructions can take a mixture of registers of immediate (8-bit) values as operands, allowing for easier and clearer computations.

2.1.1 add

```
<dst> add <arg1> <arg2>
```

$$dst = arg1 + arg2$$

`add` returns the sum of the two arguments, and writes the value back to the register specified by `<dst>`. `<arg1>` and `<arg2>` can be either registers, immediate values, or any combination of the two.

2.1.2 sub

```
<dst> sub <arg1> <arg2>
```

$$dst = arg1 - arg2$$

`sub` returns the difference of the two arguments, and writes the value back to the register specified by `<dst>`. `<arg1>` and `<arg2>` can be either registers, immediate values, or any combination of the two.

2.1.3 mul

```
<dst> mul <arg1> <arg2>
```

$$dst = arg1 * arg2$$

`mul` returns the product of the two arguments, and writes the value back to the register specified by `<dst>`. `<arg1>` and `<arg2>` can be either registers, immediate values, or any combination of the two.

2.1.4 div

```
<dst> div <arg1> <arg2>
```

$$dst = \frac{arg1}{arg2}$$

`div` returns the quotient of the two arguments, and writes the value back to the register specified by `<dst>`. `<arg1>` and `<arg2>` can be either registers, immediate values, or any combination of the two.

2.1.5 and

```
<dst> and <arg1> <arg2>
```

$$dst = arg1 \& arg2$$

`and` returns the bitwise and (&) of the two arguments, and writes the value back to the register specified by `<dst>`. `<arg1>` and `<arg2>` can be either registers, immediate values, or any combination of the two.

2.1.6 or

```
<dst> or <arg1> <arg2>
```

$$dst = arg1 | arg2$$

`or` returns the bitwise or (|) of the two arguments, and writes the value back to the register specified by `<dst>`. `<arg1>` and `<arg2>` can be either registers, immediate values, or any combination of the two.

2.1.7 not

```
<dst> not <arg>
```

$$dst = \sim arg$$

`not` returns the bitwise negation ($\sim$) of the argument, and writes the value back to the register specified by `<dst>`. `<arg>` can be either a register or immediate value.

2.1.8 xor

```
<dst> xor <arg1> <arg2>
```

$$dst = arg1 \oplus arg2$$

`xor` returns the bitwise exclusive-or ($\oplus$) of the two arguments, and writes the value back to the register specified by `<dst>`. `<arg1>` and `<arg2>` can be either registers, immediate values, or any combination of the two.

2.1.9 load

```
<dst> load <adr>
```

`load` loads the data from an address in data memory to the register specified by `dst`. The memory address is specified by `adr`, and can either be the value from a register or an immediate value.

2.1.10 store

```
<src> store <adr>
```

`store` stores the data to an address in data memory from the register specified by `src`. The memory address is specified by `adr`, and can either be the value from a register or an immediate value.

2.1.11 move

```
<dst> move <arg>
```

$$dst = arg$$

`move` copies the data specified by `arg` to the register specified by `dst`. `arg` can either be the value from a register or an immediate value.

2.1.12 print

```
<src> print
```

`print` displays the value specified by `src` to hexadecimal and decimal digit LED displays. `src` can be either the value from a register or an immediate value. (**Note**: the display will only remain lit with the value of `src` for the clock cycle in which the `print` instruction is being called).

2.1.13 label

```
<name> label
```

`label` signifies that a label is being created at the current address in the instruction memory, with the label's name being specific by `name`. The assembler records this address and will use it in any jumping instruction that specifies it. (**Note**: labels can be defined anywhere in the `.text` segment of the program, even if it is defined after it is called in a jumping instruction).

2.1.14 jump

```
<name> jump
```

Jumps to the label in the program specified by `name`. This instruction is unconditional and performed automatically. `jump` will change the PC to the address of the label specified by `name`.

2.1.15 zero

```
<name> zero <arg>
```

Jumps to the label specified by `name` only if the value in `arg` is 0. `arg` must be a register, and cannot be an immediate value. If the value of `arg` is equal to zero, then the PC will be updated to jump to the label specified by `name`. Otherwise, nothing will occur, and the PC will move on to the next instruction.

2.2 Additional Syntax Notes

- To insert a comment on a line, insert two slashes (`//<comment goes here>`) (**Note**: all text after the slashes will not be included in the program)
- Each instruction should **have its own line**
- Extra whitespace (spaces on a line, empty lines between instructions, etc.) does not matter. So long as arguments are written in the correct order and are sufficiently spaced, the assembler will work properly
- Instructions are **case-sensitive**
- Labels can be defined anywhere in the `.text` segment of your `program.txt` file, even if they are defined after the line in which it is used in a jumping instruction
- When a label is created, it will take up its own address space in the instruction memory. However, it will be represented as a blank instruction which will not perform any actions. The instructions following it will perform their actions as usual

2.3 Writing a Program

Writing a program in uwusembly is easy, however a couple of rules must be followed precisely in order to ensure that the program can be assembled correctly. In this section, the steps necessary to create a program for uwusembly will be outlined.

1. Create a standalone directory for your program. The name doesn't matter, but make sure you are able to locate the directory.
2. Create a new text file titled `program.txt`. (**Note**: the text file **MUST** be named exactly as shown, otherwise the assembler won't be able to properly recognize it)
3. In the first line of the text file, write only `.text` to signify the start of the program
4. In the following lines, begin writing instructions, with each instruction being on its own line
5. **OPTIONAL**: Write `.data` on its own line, and in the following lines, write 8-bit immediate numbers to load into data memory. The first number will be stored at address `0x00`, then `0x01`... and so on
6. Save the file

3 Assembling & Running your Program

The assembler for the uwusembly language is written in Python. Python was chosen because of its readability and relative simplicity to other languages. Additionally, file handling in Python makes it easy to read from text files and write the image files that we will later load into memory.

Assembling your program can be as simple as clicking a button. The assembler automatically reads the lines from the `program.txt` file, and translates each instruction into its binary encoding (which will be explained later in this section). These encodings are contained in "image" files, which can be loaded into the memory of the CPuwu curcuit.

3.1 Machine Code of CPuwu

The encodings of each instruction differ slightly based on if we use registers or immediates as arguments. This is because the same range of bits are used to designate registers and immediates.

2 bits at the end (bits 0-1) are not used for anything, and are simply filler so that the machine code can be 32 bits long. In the tables below, assume that any bit ranges not listed have a value of 0 by default.

3.1.1 Using Registers

Bits	**31**	**30-28**	**27**	**26-24**	**18**	**17-15**	**9-7**	**6**	**5**	**4**	**3**	**2**
Signal	`Write`	`Dst`	`Imm1`	`Src1`	`Imm2`	`Src2`	`Oper`	`MemR`	`MemW`	`Print`	`Jump`	`Zero`

When using registers, bits 26-24 will denote which one of the 8 registers we will be reading from (same applies to bits 17-15). When using registers for either one of the sources, the corresponding `Imm1` or `Imm2` control signals will be disabled, signifying that we will be reading values from registers.

3.1.2 Using Immediates

Bits	**31**	**30-28**	**27**	**26-19**	**18**	**17-10**	**9-7**	**6**	**5**	**4**	**3**	**2**
Signal	`Write`	`Dst`	`Imm1`	`Num1`	`Imm2`	`Num2`	`Oper`	`MemR`	`MemW`	`Print`	`Jump`	`Zero`

When using immediates, bits 26-19 will denote which 8-bit immediate number we will be using (same applies to bits 17-10). When using immediates for either one of the sources, the corresponding `Imm1` or `Imm2` control signals will be enabled, signifying that we will be using immediate values for instructions.

3.1.3 Control Signals

The variables listed in the above tables are control signals, and, when enabled, will denote that a certain function or operation is to be performed by the CPuwu. All of the different combinations of these control signals make up the instructions of uwusembly. Below is a definition of all control signals:

- `Write`: when enabled, it signals that a register will be written to
- `Dst`: the 3 bits that denote the register that will be written to if `Write` is enabled
- `Imm1`: when enabled, it signals that we will be using an immediate value as our first operand. Otherwise, we use the value from a register
- `Src1`: the 3 bits that denote the first register to read from, if `Imm1` is disabled
- `Num1`: the 8 bits that represent the first immediate value to be used, if `Imm1` is enabled
- `Src2`: the 3 bits that denote the second register to read from, if `Imm2` is disabled
- `Num2`: the 8 bits that represent the second immediate value to be used, if `Imm2` is enabled
- `Oper`: the 3 bits that represent which arithmetic/bitwise operation we will be performing
- `MemR`: when enabled, it signals that we are reading a value from data memory
- `MemW`: when enabled, it signals that we are writing a value to data memory
- `Print`: when enabled, it signals that a value should be printed to the display
- `Jump`: when enabled, it signals that we will be jumping to another instruction memory address instead of using the next PC value (**Note**: this can be altered by the following `Zero` signal)
- `Zero`: when enabled, it signals that we should use the result of the comparison to 0 to determine whether or not to jump (**Note**: if the comparison to 0 results in a value of 1, then the `Jump` signal will be set to 1. Otherwise, it will be 0)

3.1.4 Binary Encodings of Instructions

Below is a general overview of how each instruction is encoded in its 32-bit binary format. This section serves to show which control signals are enabled with each instruction. *As such, we will not include* `Dst`, `Src`, *or* `Num` *signals, as they will always vary with each instruction.*

Bits	**31**	**27**	**18**	**9-7**	**6**	**5**	**4**	**3**	**2**
Signal	`Write`	`Imm1`	`Imm2`	`Oper`	`MemR`	`MemW`	`Print`	`Jump`	`Zero`
add	1	0/1*	0/1*	000	0	0	0	0	0
sub	1	0/1*	0/1*	001	0	0	0	0	0
mul	1	0/1*	0/1*	010	0	0	0	0	0
div	1	0/1*	0/1*	011	0	0	0	0	0
and	1	0/1*	0/1*	100	0	0	0	0	0
or	1	0/1*	0/1*	101	0	0	0	0	0
not	1	0/1*	0/1*	110	0	0	0	0	0
xor	1	0/1*	0/1*	111	0	0	0	0	0
load	1	1	N/A	N/A	1	0	0	0	0
store	0	1	N/A	N/A	0	1	0	0	0
move	1	0/1*	1	N/A	0	0	0	0	0
label	0	0	N/A	N/A	0	0	0	0	0
print	0	0/1*	N/A	N/A	0	0	1	0	0
jump	0	1	N/A	N/A	0	0	0	1	0
zero	0	1	N/A	N/A	0	0	0	1**	1

- ***For these instructions, the signal will be enabled depending on whether an immediate is supplied as an argument
- ****A value of 1 is passed from the assembler, but this instruction will not always result in a jump, as it is conditional
- For all entries marked with **N/A**, the value of those bits have no bearing on the result of the instruction. **They are marked as 0 by the assembler**

Example: u1 add u0 2

Bits	**31**	**30-28**	**27**	**26-24**	**18**	**17-10**	**9-7**	**6**	**5**	**4**	**3**	**2**
Signal	`Write`	`Dst`	`Imm1`	`Src1`	`Imm2`	`Num2`	`Oper`	`MemR`	`MemW`	`Print`	`Jump`	`Zero`
`add`	1	001	0	000	1	00000001	000	0	0	0	0	0

3.2 Using the Assembler

The assembler program uses the rules described above to translate instructions into their binary encodings. It goes through each line of the program, and then writes the binary encoding of that line into a properly-formatted image file to be loaded into the Logisim circuit. Here is how to use the assembler:

1. Place `uwusembler.py` in the **same directory** as `program.txt`
2. Run the assembler program. There are multiple ways to do this, depending on the platform you're using:

   Visual Studio Code: Open the directory folder in VS Code, and open `uwusembler.py`. Then run the script

   IDLE: Open `uwusembler.py` in IDLE, and run the script in the shell.

   Windows: Open a terminal window, and enter the current directory of your project. Then type the following command: `py uwusembler.py`

You should now have a plain file titled `instr` (and also `data`, if you included a `.data` section in your program). This file contains the binary encodings of the instructions that can be loaded into the instruction memory of CPuwu.

3.3 Running the Program in CPuwu

Now that we have the proper image files, we can run the program in the Logisim circuit and see the results of the program. After opening the `cpuwu.circ` file, follow these steps to ensure the program runs correctly:

1. Make sure `Simulate > Auto-Propagate` is enabled
2. Reset the simulation (`Simulate > Reset Simulation`)
3. Right-click on the Instruction Memory object (on the left side of the circuit), select `Load Image...`, and select the `instr` file
4. **OPTIONAL**: If you have a `.data` section of your program, load the `data` file into the Data Memory object (on the right side of the circuit), using the same method above
5. Set your simulation speed with `Simulate > Auto-Tick Frequency`, and then select your prefered clock rate (*we reccomend **0.5 Hz** for small programs, so that you can easily see the results of each instruction*)
6. Enable `Simulate > Auto-Tick Enabled`, and now your program will start running!

Enjoy CPuwu!

We hope that you enjoy grading CPuwu and using uwusembly to test your own programs! While it is rather barebones, we had a lot of fun creating the Logisim circuit, the assembly language, and the assembler program. (This was also a great opportunity to strengthen our Logisim, Python, and LaTeX skills!).

I pledge my honor that I have abided by the Stevens Honor System.
-John Graham & Emma Hodor