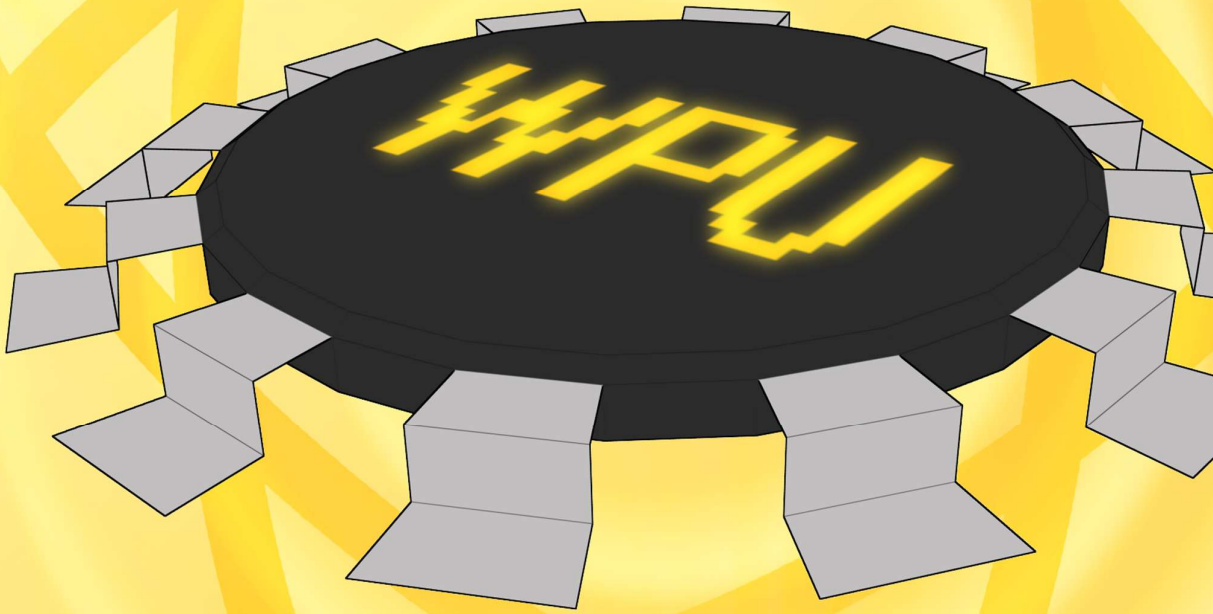
ATTOWPU 0.9 SPECIFICATION

Designed and written by:

Tomáš “Frooxius” Mariančík



Patreon archival release

https://patreon.com/frooxius

Contents

attoWPU 1.0 specification 1
Contents 2
Overview 5
Attoinstructions 6
Buses 7
Address bus 8-bit 0 – 7 7
Control bus 8-bit 8 – 15 7
Data bus 32-bit 16 – 47 8
Quick aJump bus 16-bit 48 – 63 9
Programming 10
Attoassembly (attoASM) 10
Integers 10
Floating point numbers 10
ASCII characters 11
Attoinstructions 11
Grouping attoinstructions 12
Attoinstruction syntax 12
Comments 12
Converting number to attoinstructions 12
Labels 13
Arbitrary data specification - data chunks 13
Including files 14
Symbol definition 15
Symbol redefinition 15
Local symbols and labels 15
Attocode organization 16
Customizable assembly (custASM) 16
Data units 17
Simple expressions 19
Instruction definition and usage 19
Comments 20
Symbols 21
Labels 21

Overriding argument value 22
Including files 22
Assembly process settings 22
Simulation and usage 23
Simulation 23
Usage 23
Units reference 25
Clock (---) 25
Attocore (---) 25
aPC write (accessed directly through Quick aJump) 25
aPC (0x00) 26
Internal registers 26
Command codes 2 valid bits 26
Attocode memory (0x01) 26
Internal registers 26
Command codes 4 valid bits 27
TEMP register (0x02) 27
Internal registers 27
Command codes 4 valid bits 28
Register memory (0x03) 28
Internal registers 28
Command codes 5 valid bits 29
ALU (0x04) 30
Internal registers 30
Command codes 6 valid bits 30
OUT register (0x05) 31
Internal registers 31
Command codes 1 valid bit 31
FPU (0x06) 32
Internal registers 32
Command codes 5 valid bits 32
Memory controller A (0x07) 32
Internal registers 33
Command codes 5 valid bits 33

Memory controller B (0x08) 34
SmallQueue (0x09) 34
Internal registers 35
Command codes 5 valid bits 35
LED control (0x0A) 36
Internal registers 36
Command codes (2 valid bits) 36
Text display controller (0x0B) 36
Internal registers 36
Command codes (4 valid bits) 37
LCD Display Controller (0x0C) 37
Internal registers 37
Command codes (5 valid bits) 38
Input Controller (0x0D) 38
Numeric keyboard layout 39
Reading several scan codes 39
Internal registers 39
Command codes (4 valid bits) 39
Timer controller (0x0E) 40
Internal registers 40
Speaker Output (0x0F) 40
Internal registers 41
Command codes (0 valid bits) 41
Glossary 42
Authors 44
attoWPU specification, design and programming 44
Tomáš “Frooxius” Mariančík 44

OVERVIEW

AttoWPU is experimental processor architecture from the WPU (Weird Processing Unit) series with accompanying programming languages, which tries different unusual approach the assembly programming and programming in general for various purposes, including education, curiosity, fun and even an artistic intent.

This experimental **big endian** processing unit allows programmer to design processor's function himself, using special **attoassembly language**. Processor itself doesn't have any function at all or direct way to control all its parts using conventional instructions, because there aren't any built in opcodes for a program. There are however three opcodes for the **attocode**, allowing to change status of one 64 wires, that is, set it to logical zero, logical one or invert the current status.

64 wires connected directly to the processor's core are divided into four buses: address bus, control bus, data bus and **Quick aJump** bus. These buses allow information exchange between various logical blocks/units connected to these buses and control these units by changing values on the buses using three **attoinstructions**.

The base of the processor is the **attocore**. During each attocycle, this logical part will read current instruction from attocode memory, decode instruction and bit number from it and change value of specified bit accordingly. Then it will instruct **atto Program Counter (aPC)** to increment by one, thus, moving to the next attoinstruction. This simple basic process repeats until the processor is stopped.

Programmer can create processor's function by designing the program in the attocode program memory: by specifically adjusting values of individual wires of the buses, he can control other units in the processor to his needs, usually by creating attocode that will process actual program created in normal assembly language, stored in **program+data memory**: the attocode will decode instruction opcode (designed by programmer) from the program+data memory and then control processor's units to execute appropriate function.

Note: this documentation specifies architecture only from programmer's viewpoint; implementation-specific details are not specified.

ATTOINSTRUCTIONS

Attoinstructions are used to control logical values of individual wires and thus to change binary values of buses. Each attoinstruction changes status of only one bit at the time. There are three attoinstructions available (four in some implementations):

- Rise (sets to logical zero)
- Fall (sets to logical one)
- Invert (inverts current value)
- Halt (stops the processor, optional)

Instruction must be combined with a wire number it changes: each wire has its unique number (address). There are 64 wires, so 6 bits are needed to address them correctly. 2 bits are needed to encode the instruction code. Thus, each single instruction has size of 8 bits (one byte) and with following format:

IH	IL	WN5	WN4	WN3	WN2	WN1	WN0
0	1	2	3	4	5	6	7

IL – instruction low bit
IH – instruction high bit

IH	IL	Instruction
0	0	Fall
0	1	Raise
1	0	Invert
1	1	HALT/undefined (see below)

WN0 to WN5 – Wire Number (from 0 to 63 dec)

There's a special HALT instruction: 0xFF
HALT causes the processor to stop and do not process any other attoinstruction unless it's reset. This is implemented mainly in the simulator, because it will allow doing certain tasks, which are described later, physical version of the processor can however just ignore this instruction, as it probably won't be much relevant.

BUSES

The buses are the most important part of this processor, because all other parts are connected to one or more buses in parallel, so they can receive and output data. There are four buses, each one having different number of wires for data exchange. The number of wires (bus width) determines how much data can be transferred at once and how many combinations are possible.

Address bus 8-bit 0 – 7

Because there are various units in the processor, it's needed to select one of them prior to sending command codes to the unit. This is done via address bus. All units are connected to this bus and each one of them has unique 8 bit address. When the address on the address bus matches the address of the unit, it becomes active and starts receiving commands from the control bus. Other not currently addressed units ignore commands from the control bus. Because this bus is 8 bit, it's possible for up to 256 units to b implemented, which may serve as a reserve for future versions, because current one doesn't utilize even quarter of available addresses.

A7	A6	A5	A4	A3	A2	A1	A0
0	1	2	3	4	5	6	7

A0 – A7 address bits

Value after reset: 0x00

Control bus 8-bit 8 – 15

Once unit is addressed, it can receive commands from the control bus, which tell it what to do: for example, command may tell unit to read value from the data bus and store it in itself or output its value to the data bus, start and stop timer, fill the OUT register with a result of some calculation and so on.

Although the bus is 8 bit, the maximum number of direct commands for one unit is 128. This is because the least significant bit (wire 15) is used to indicate when to execute the command from the control bus: programmer first prepares 7-bit command code on the control bus and then changes the value on wire 15 to logical 1 and then back to 0. Unit ignores commands on the control bus until it registers change from 0 to 1 in the least significant bit. Only when change is detected the command is executed. This is needed because the attocore changes only one bit at the cycle, so it needs 7 cycles to change all 7 bits. If the unit didn't wait for the least significant bit to change, it would start executing different commands in each attocycle before the desired command would be completely prepared on the bus.

As has been said, each unit ignores the control bus completely when its address doesn't match the address at the address bus. This is done by ANDing all bits of the control bus with logical 1 in case the unit address matches the address at address but and with logical 0 in case it doesn't match. If programmer leaves the execution bit in the control bus set to 1 and changes value at the address bus, the unit at new address will execute the command immediately, because it will register change from 0 to 1, although the bit didn't change at the moment on the control bus.

The least significant bit used to execute command is called **execution bit**. Each unit has its own set of commands. It can choose to ignore some bits of the bus (this is then specified at the unit documentation page by number of valid bits).

CC6	CC5	CC4	CC3	CC2	CC1	CC0	EB
8	9	10	11	12	13	14	15

CC0 – CC6 control code (7bit, max 128 codes)
EB execution bit

Value after reset: 0x00

Data bus 32-bit 16 – 47

This is the largest bus in the processor and allows exchange of various data in arbitrary binary form (depending on the unit). Because of 32-bit width, it can transfer also standard values like 32-bit integer, single-precision float value or even a 32-bit address, allowing addressing up to 4 GB directly (that is, without need to transfer two or more parts of the address), although this much memory won't be usually used or even implemented.

Units can choose to ignore one or more bits of this bus if they're not needed (even changing the number of ignored bits depending on the command), this is then specified in the documentation of the unit. This allows programmer to leave out part of the code, which would clear all 32-bits to zero, before letting the unit to read the value.

Unlike control and address bus, units can also output data to this bus, where they can be read by some other unit. This allows exchange of data between two (or more in some cases) units connected to this bus. Some units may only read data and some only write, depending on what they're used for. Most units will however need both to write and read data in order to function.

If some unit is going to output data to the bus, the attocore must set all bits to 1 (or at least these that will be used). Logical zero value behaves like ground in electrical circuit – it will ground any voltage source and cause the voltage to be zero in all parts of the bus (in all units), causing all units to read logical 0 even if one or more units are outputting 1. This basically behaves like bitwise AND operation, so all units which are not currently used need to output 1 too (or change their output to high impedance mode), so the communicating unit can output logical 1 without being grounded by other units. Usually, units have command to change their output into high impedance mode, when they're not used.

D32	D30	D29	D28	…	D03	D02	D01	D00
16	17	18	19	…	44	45	46	47

D32 with signed int operations, this is a sign bit, otherwise it's not special anyhow

Value after reset: 0x00000000

Quick aJump bus 16-bit 48 – 63

This is special bus and it's used only by one part of the processor: atto program counter and attocode program memory. This bus allows making quick unconditional local jump in the attocode program, without changing the state of other buses. Although the bus is 16-bit, address can be only 15-bit, because the least significant bit is used similarly as the execution bit on the control bus. The only unit connected to the Quick aJump bus is the aPC write: this unit waits for the least significant bit on this bus to change to 1 from 0. Once it changes, it writes 15bit address to the aPC at once (instead of bit by bit) causing it to jump immediately. Least significant bit is called **jump bit**.

This allows fast jump in 32kB block of code, but the attocode program memory is 1 MB for example, which means that it's not possible to do long jump this way, only jump within a 32 kB block. There are 32 blocks and to jump between them, all other three buses need to be used and modified in order to make a long jump. Quick aJump bus allows making quick jumps whenever possible thus reducing program size and increasing processing speed.

AAE	AAD	AAC	AAB	AAA	AA9	AA8	AA7	AA6	AA5	AA4	AA3	AA2	AA1	AA0	JB
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

AA0 – AAE 15 bit attocode address
JB jump bit

Value after reset: 0x00000000

PROGRAMMING

Two languages are associated with the AttoWPU. Most important is the attoassembly language, attoASM for short, which is used to create the attocode, which controls the execution units. It's possible to write actuall programs in this language too, but it's recommened to use it for specification of processor's function for the program in the program and data memory - decoding instructions and instructing execution units to do various tasks.

To create the actuall program, that will be executed by the attocode, programmer can use the custom assembly programming language, custASM for short. This language is optional and isn't required, but it simplifies creation of programmer's own instructions, because it allows to define own instruction mnemonics and corresponding opcodes easily, thus removing the need to use external/own tools.

Attoassembly (attoASM)

To create an attocode program the **attoassembly** language is used (**attoASM** for short), and then assembled with **attoassembler**, producing machine code (attocode) which can be directly executed by the attoWPU's attocore.

Integers

To specify an integer number, four numeric bases can be used: binary, octal, decimal and hexadecimal. This is done by appending a symbol after the number. This is B for binary, O for octal, D for decimal (this is optional, as decimal is default) and H for hexadecimal. It's also possible to prepend minus sign to produce negative value.

Integer is a 32bit value, whether it's signed or not is determined automatically: for positive values smaller or equal to the maximum value of the 32bit signed integer (2 147 483 647), it's not required to determine type. For values larger than this value, integer is automatically stored as unsigned, for negative values, it's always signed. Signed integers are stored as two's complement.

Integers are used for specifying a start bit and conversion of a numeric value to a series of attoinstructions (see details below). If a hexadecimal integer starts with symbols A – F, then 0 must be prepended. It's also possible to use simple math using plus and minus sign: addition or subtraction with several numbers (and symbols) together, resulting in one final integer number, which is processed as usual.

Syntax:
```
<number>[base]
```

Example:
```
30923               // integer (can be both signed and unsigned)
-2440               // signed integer specified in octal
0C5809H             // hexadecimal integer (can be both signed and unsigned)
-1010111111B        // binary signed integer
3879330290          // unsigned integer (exceeds range of the signed 32bit integer)
309H+1101B-VAL      // simple integer math
```

Floating point numbers

It's possible to specify also a single precision floating point number in the attocode, but there are some limitations: there's no floating point math in the attoASM, because all floating point numbers are internally handled like raw 32 bit data (unsigned integer), so any math would produce a gibberish values. Thus, syntax for specifying floating point numbers is different from normal integers, to prevent using it in the integer math. Floating point number must be

preceded with & symbol, everything after this symbol is evaluated as a floating point number, until a whitespace or non-alphanumeric character is met.

When floating point number is specified on its own, it's saved directly in the attocode at the point, where it was specified, similarly to specifying arbitrary data, it can also be specified after the wire number (in most cases, the start of the data bus as it has no meaning on other buses) as an attoinstruction: it's converted to 32 attoinstructions corresponding to the raw binary data that are representing given number.

Example:
```
&2.222388           // store this floating point number in the attocode memory
DATA &3.141592      // write this floating point number on the databus
```

ASCII characters

By specifying a single character in single quotes, it's possible to obtain numeric code of the given character from the ASCII table, this behaves just like integer does, so it can even be used in expressions. Although ASCII values are 8-bit, they are expanded to 32 bits like other integers – the additional bits are simply filled with zeroes.

Example:
```
DATA+23 ['f', 8]    // write code of the character f to the databus
```

Attoinstructions

Attocode itself is specified using one of the four (usually only three) attoinstructions and number (address) of the bus bit they apply to. Each attoinstructions starts by specifying an integer ranging from 0 to 63, which specifies the bit number on the bus (one of four available buses respectively), this integer is then followed by one or more symbols representing the four attoinstructions. If more than one attoinstruction is specified after the start bit, then each following attoinstruction applies to the next bit of the bus, relative to the starting bit. This allows specifying two or more consequent attoinstructions, without having to manually starting bit for each one of them: this is done by the attoassembler tool. For convenience, there are predefined symbols for starting bits of the four buses.

Predefined symbols (symbols are explained later) for buses are following:

```
ADDR = 0     start of the address bus
CTRL = 8     start of the control bus
DATA = 16    start of the data bus
AJMP = 48    start of the Quick aJump bus
```

Attoinstructions are represented by following symbols:

```
0     fall
1     rise
!     invert
|     halt
-     skip (pseudo instruction)
```

It's also possible to add or subtract a number to/from a wire start symbol, to shift the starting wire address, which is basically simple integer math described in the Integers part.

For example this instruction will change control bus bits 3, 4 and 5 all to zero and bit 6 to 1:

```
CTRL+2  0001
```

Pseudo instruction – allows programmer to write a sequence of attoinstructions at once, where one or more of the bits will be skipped, without having to write two separate statements (start bit + attoinstructions). For example:

```
DATA 110-11001
```

Is equivalent of:

```
DATA 110
DATA+4 1101
```

Grouping attoinstructions

It's also possible to shorten group of same, repeating instructions, to reduce required typing, by putting the total number of instructions in parentheses right after the instruction. Attoassembler will automatically generate appropriate number of separate instructions. So previous example can be rewritten like this:

```
CTRL+2 0(3)1
```

It's also possible to repeat a whole group of instructions several times, by placing number of repeats in parentheses right after the starting wire number. This will be equal to placing the same group of instructions several times in the source code sequentially. Most common use will be probably with invert instructions as setting some bits several times to the same value makes no sense. Most common use is for example quick rise and fall of the execution bit:

```
CTRL+7(2) !
```

This is a shorter way for writing:

```
CTRL+7!
CTRL+7!
```

Attoinstruction syntax

Formal form of the attoinstruction is following (parts in square bracelets are optional):

```
<startwire> [(<group repeats>)] <instruction>[(<number of consecutive bits it applies to>)]  …
<instruction startwire+n wire>[(<number of consequent wires it applies to >)]
```

Instruction groups are separated by spaces, each time a new start wire number is specified, it's considered a new group of attoinstructions, so it's possible to write several groups on a single line, but for readability concerns, it is advised to separate them by newlines.

Wire number can be any number from 0 to 63 one of predefined symbols can be used too, as well as basic arithmetic operations: addition and subtraction. It's however important to ensure that the resulting number won't exceed the limit. Also if number 63 is used, only one instruction can follow as the next ones would have to be applied to nonexistent wires 64 and above, so such cases will result in error during attoassembling.

Comments

Given to the nature of the attoassembly, comments are highly required, to keep the code easy to understand. C-like comments are supported. Symbols // indicate start of a single line comment, where everything after these two symbols is ignored, until a newline is encountered. Symbols /* and */ indicate start and end of a multiline comment and everything between these symbols is ignored.

```
DATA 0(7)1   // address the attocode memory
/* Following code will execute previously prepared command,
which instructs it cease all data output, so the buses can be used for
communication of other units */
CTRL(2) !
```

Converting number to attoinstructions

Programmer may convert 32 bit integers to a series of attoinstructions, for example he may write an integer number in decimal and let the attoassembler convert it to proper attoinstructions. He can do the same also for integers in

hexadecimal or octal base and ASCII characters, which are also considered integers. To do so, he must enclose number in square brackets, optionally he may specify how many bits from default maximum 32 will be converted. Whether the integer will be signed or not is determined automatically: for values up to 2 147 483 648, binary representation is same both for signed and unsigned. If the number is larger than this, then it's unsigned, if the number is smaller than zero, it's signed. Number of converted instructions is specified in the square brackets: after the actual integer itself is placed a comma, then number of bits (starting by least significant bit) to be converted, another comma is placed and number of bits (starting by least significant bit) to be skipped. It's possible to use simple expressions in square brackets, that is, adding and subtracting.

Syntax (part in the parentheses is optional):

```
[<number or expression> (,<number of bits to convert>,<number of bits to skip>)]
```

Examples:

```
DATA [15550]              // write number 15550 to the data bus
ADDR [0CH,8,0]            // address unit 0CH
```

Labels

If programmer needs to create a jump to some location of the attocode or find out address at some position in the code, he can use labels. Label is created by specifying label name followed by a semicolon at any place in the source code, separated with spaces. Then this name can be used in any part of the code as a symbol, where it's replaced with the address of the attoinstruction at position, where label was created; label is preceding attoinstruction in question. Address if formed by a sequence of 20 attoinstructions, so it's possible to use the label as a group of attoinstructions. Label name must start with a letter or an underscore and can contain letters, numbers and underscores. Labels can be defined only once, but used as many times as possible, when is label used as a symbol, it represents an integer and it must be converted to series of attoinstructions as any other integer.

Example infinite loop:

```
ADDR 0(8)           // address aPC
CTRL 0(6)01         // prepare command code to write new address to the aPC
DATA+11 [Somelabel, 20]   //write the address of the label to the data bus
Somelabel:
CTRL+7 0            // change execution bit to zero (may be 1, when jump occurred)
CTRL+7 1            // execute the command and thus cause jump back to Somelabel
```

Labels also serve another purpose: after the source is assembled, special text file is generated in addition to the attocode (machine code). This file will contain list of all the labels defined in the source file as well as their corresponding addresses in the attocode, which means that programmer can put labels anywhere in the code and after assembly, he gets address of the first instruction after the label. He may use this information when defining his own instructions in the custom assembly (see below) or discard it.

Arbitrary data specification - data chunks

It's also possible to include arbitrary binary data in the source file and then resulting attocode, even though these data are not attoinstructions themselves: because the attocode memory is accessible for reading, programmer may store some important data which will be then used by the attocode somehow. Arbitrary binary data are called a data chunks. Data chunks must always be padded to bytes, it's not possible to fill only 4 bits for example, because each following attoinstruction would be then split in half and each half would be in a different ememory cell. Data chunk can be specified either using hexadecimal base or as an ASCII string.

Hexadecimal

Hexadecimal form of a data chunk starts by writing $ and then arbitrary number of bytes in hexadecimal form. Data chunk is ended with a space. Bytes are stored in memory in the order they are read: from left to right, so the byte immediately after $ symbol is stored at lowest address. If the last nibble doesn't fit a while byte, then four zero bits are added at the end and warning is generated. For example, programmer can write following:

```
$00A3                    // two bytes of data
$1E892A8C881F DATA+2 0010 // data chunk followed by an attoinstruction
```

ASCII String

Second form is an ASCII string, where each character is converted to a single byte, with corresponding ASCII value. Only basic escaping is done and no zero byte is provided at the end – if programmer needs it, he must specify it himself. Characters must be enclosed in double quotes and similarly to hexadecimal data, they have to be separated from other parts by spaces. For example:

```
DATA 0(31)1 "Test string"        // string data chunk preceded by group of instructions
"This is a longer string.\nTerminated with zero manually" 0x00    // zero terminated string
```

Symbol	Meaning
\n	Newline
\"	Double quote
\t	Horizontal tab
\0	Null
\\	Backslash
\f	Form feed
\r	Carriage return
\b	Backspace

Label

It's also possible to store 32-bit value stored in a label in the attocode for various purposes, for example by creating array of function pointers. This is done by simply putting the name of the label in parentheses right after the $ symbol.

Syntax

```
$(<labelname>)
```

Example

```
<0>
$(LABEL)     // this will write an integer with value 8 into the attocode
DATA [8034]
LABEL:
…
```

Including files

Especially for larger projects, it's possible to separate the code into multiple files and then include these files into one main file. When a file is included, all of its contents are placed at the point of inclusion, as if they were written directly at the place of include directive. File can be included using the include function and specifying absolute or relative path to the file that will be included. Filename and path can't contain double quotes.

Syntax:

```
include("<path>")
```


Examples:
```
include("somefile.att")
include("includes/lib.att")
```

Symbol definition

To define an arbitrary symbol, which is basically alias for an arbitrary part of the source code of arbitrary length, curly brackets are used. Programmer first specifies unique name of the symbol, followed by opening curly bracket. Then he can write attoASM code of any length, containing any of valid attoASM statements. Then he closes the block by closing curly bracket. Symbol name must start with a letter and can contain only letters, numbers and underscores.

After this definition, he may use the symbol wherever he needs and it will be replaced by the code specified in curly brackets, so symbols can be used to easily place repeating code block at several places in the source or to simply define numerical value or group of attoinstructions. It's however not possible to define another symbols and labels inside the symbol body, because if the symbol was used more than once, this would cause symbol/label redefinition, as the symbol is simply replaced by the code.

Syntax:
```
<symbol name> { <code> }
```

Example:
```
EXECUTE { CTRL+7(2) ! }   // define a whole attoinstruction
EXE { EXECUTE }           // it's possible to use symbols within symbol definitions, to create
aliases
Default { [440, 16, 0] }  // define just attoinstructions without starting wire
TMP_to_aPC {              // defining a whole block of code
ADDR [0x02, 8, 0]
CTRL [0x04, 7, 0]
EXE
ADDR [0x00, 8, 0]
CTRL [0x01, 7, 0]
EXE }

DATA+4 Default            // usage of the symbol Default
```

Symbol redefinition

Normally, symbols can be defined only once and redefinition will cause an assembly error. However, it's possible to force symbol redefinition by adding the exclamation mark right after the first curly bracket. This will cause the previous symbol definition (if there's any) to be replaced with the new one, that will be valid from the point it was redefined, including in the symbols defined before redefinition, but used after the redefinition. This can be useful for example for passing arguments to the symbols. Labels can't be redefined.

Syntax:
```
<symbol name> {! <code> }
```

Example:
```
ARG { 0 }
SOMETHING { DATA [ARG, 32] }

SOMETHING    // writes 0 to the data bus
ARG {! 255 } // redefine symbol
SOMETHING    // writes 255 to the data bus
```

Local symbols and labels

Symbols and labels shouldn't be defined inside another symbol, because using symbol more than once would cause symbol/label redefinition, because the symbol's code is placed completely at the place, where it is used, including

the symbol/label definition. This doesn't allow creating labels that will be always relative to the place where the symbol is used, instead of one predefined label outside the symbol where label is used. This can be solved by adding a % symbol after the label/symbol name, which will make its name unique each time it's used in a symbol: its name will be different each time symbol, where is this symbol/label defined, is used. This is achieved by appending a number after the symbol name that is unique for each symbol usage. Percentage symbol has to be appended both in local symbol definition and its usage. Symbol is also local within an included file.

Syntax:
```
SYMBOL
{
AJMP [LABEL%, 15]
LABEL%:
AJMP+15(2) !
}
```

Attocode organization

It's possible to specify a starting address, where following attoinstructions will be placed. This allows creating gaps that are filled with zeroes, usually to reserve some space for some other purposes. It's technically possible to specify also address pointing before current position. This will however cause the machine code already assembled there to be overwritten and a warning to be generated. Start address is specified using number between symbols < and >. This can be immediate integer number, symbol, combination of both using simple math, but no label can be used.

Syntax (variables are within square brackets):
```
<[address]>
```

Example:
```
// code
<800H>
// more code…
```

Customizable assembly (custASM)

The attocode allows mainly to create processor's function, but not to write programs themselves easily. Although writing programs themselves using attocode is somewhat possible, it's not recommended. Attocode memory also doesn't provide sufficient memory capacity for larger programs, although this varies with the amount of memory provided in given implementation. Also, writing programs directly in attoassembly would be very tedious task. Therefore, programmers will usually use the program+data memory (via Memory Controller A) to store the actual functional program with attocode serving as the controller of the processor, but they need to build machine code for their own attocode, with appropriate format, so they need an assembly language, which allows them to customize the instructions and their opcodes.

For this purpose, special **customizable assembly (custASM)** language is provided: It allows programmer to specify his own assembly instructions together with opcodes and argument layout. Custom assembler tool will then assembly program written in the custASM, using definitions of instructions provided by programmer. This program is then loaded into processor's program+data memory and executed by the attocode. Of course, how exactly will programmer use attoassembly together with custom assembly depends on him and his needs. He may choose to completely ignore program+data memory and for example carry out commands from some input unit, which he will use as a source of instructions. Moreover, it's also possible to use different assembler/compiler for a different architecture or even write his own tool and in some cases simply use just the attoASM to create the program: there's no limitation in this regard and thus, custom assembly is an optional, but recommended tool.

Customizable assembly provides programmer a way to define his own instructions, by creating instruction signature and using special symbols to determine, where arguments will be placed. Custom assembler will then try to match used instruction mnemonics to this signature to generate appropriate machine code, if no matching signature is found, it will issue an error and terminate the assembly process. Technically, custom assembler is not required to program for attoWPU, programmer can for example create his own assembler tool to create a machine code that his attocode can work with or he can use any existing assembler, assuming he creates appropriate attocode for corresponding machine code. Custom assembler however provides quick and easy way to start programming with own instruction set without need to program own tools.

Data units

Binary data are used throughout whole custASM programming language: for specifying opcodes for instructions, arguments for instructions and arbitrary binary dat. There are several ways to specify data: as an arbitrary size integer, as a single precision floating point unit and as ASCII characters/strings. Integer data can be specified in more numerical bases: binary, octal, decimal and hexadecimal.

A data unit is a single piece of data, usually 32 bits wide, which is most usually passed to instructions as an argument.

Integer

Any numerical value without a floating point is considered an integer, with arbitrary bit length, which can be specified by the programmer. If no size is specified and only integer number is placed in the program, it is considered to be a 32-bit integer. If it's a negative value, it's automatically stored as a signed integer, if it's larger than maximum signed value for given bit width, it's automatically unsigned (values from 0 to signed max value are the same for signed and unsigned). It's not needed to manually specify if the number is signed or unsigned, because only raw binary data are stored in the machine code and it's up to programmer how he will use these – there's no concept of type safe variables. Numbers are considered to be in base 10 by default and the base can be changed by adding a letter after the number:

B	binary	(11001101B)
O	octal	(13673O)
D	decimal	(923950D)
H	hexadecimal	(0FF39A8CH)

All values are considered to be 32 bit integers by default (as the attoWPU has 32 bit bus), but this is possible to alter by appending a number after the numeric base letter, specifying how many bits will be used to store the value. It's also possible to use a letter "x", instead of a number, which means that compiler will choose correct size automatically, based on the value, allowing programmer to specify arbitrary sized data, without having to manually count how many bits it does contain. In this case, all data that are specified are counted automatically, including leading zeroes, so the programmer can specify arbitrary binary data for various uses wherever needed. The actual size of the data will however always be a multiple of four. Complete syntax for specifying an integer value is following:

```
<number>[<base>[<bit_width>]]
```

Examples:

```
11001000B8              // 8 bit integer written in binary
35601O                  // 32 b integer written in octal
32D8                    // 8 b integer written in decimal
083AE09C933H64          // 64 b integer written in hexadecimal
```

```
2085442                     // 32 b integer written in decimal
0000FD380B838A0245CAF0Hx    // arbitrary size binary data in hexadecimal
```

Floating point

It's also possible to define floating point values, which are single precision by default, thus occupying 32 bits, because it's the same format, that attoWPU's FPU can directly work with as well as its data bus width. However, it's also possible to store double precision floating point values by appending a D after the floating point number. Floating point numbers can be specified only in decimal base and they must contain a decimal point. Syntax of such numbers is very simple:

```
<number with decimal point>[D]
```

Examples:

```
438.429             // 32 bit single precision floating point value
12.0941229955093D   // 64 bit double precision floating point value
```

Characters and strings

It's possible to specify any ASCII character, by putting the character in double quotes, the value is always 8 bit. Only lower 128 ASCII characters are officially supported, special encoding specific characters will be encoded too, by directly reading the binary value of the from the source file. It's also possible to specify more than one character within the quotes, creating a string. Strings are not zero terminated and programmer needs to add null character at the end himself if needed. If a string is passed to an instruction, then it will be automatically trimmed to the maximal size of the instruction's argument, starting with the first character and moving to the next one. If there are not enough characters in the supplied string, then the rest of the argument binary data is filled with zeroes, meaning that the characters are padded to the left, unlike numerical values, which are padded to the right.

Character escaping is also supported, as it's often required either to escape the single quote symbol, or define some special characters, especially the null character, which is used to make a zero terminated string. Supported characters are same as with the attoWPU:

Symbol	Meaning
\n	Newline
\"	Double quote
\t	Horizontal tab
\0	Null
****	Backslash
\f	Form feed
\r	Carriage return
\b	Backspace

Syntax:

```
"<one or more characters>"
```

Example:

```
"f"
"This string is not fluffy."
"Neither this one, but it's at least zero terminated\0"
```


Arbitrary data specification - Data chunks

If programmer needs to save pure binary data into the machine code, he will use data units for this. Basically, each time a data unit value is specified alone, without being assigned to a symbol or passed to a function, it's stored as is in the machine code. Very useful for this is the possibility to specify arbitrary size binary value, using the *x* symbol after numeric base letter. Values are however not automatically padded to a byte, which can cause problems later, but there's a possibility to enable this padding automatically.

Simple expressions

It's possible to do a simple integer math with the integer data units and floating point data units, resulting in an integer or floating point data unit, which size is the size of the largest unit in the expression. If any floating point unit is used in the expression, then the result will be floating point too, even when some numbers are integers. If there's not floating point data unit, the result will always be integer. Whether the resulting floating point data unit will be single or double is determined only by units used in the expression, custASM doesn't expand single to double by itself. If there is no double precision floating point data unit in the expression, only at least one single precision, then the result is single, if there's at least one double, then the result is double as well.

Programmer may use addition, subtraction, multiplication and division with integer data units at any point in the program, where specifying a data unit is valid (as the expression itself results in a data unit), it's also possible to use symbols and labels in math, assuming they hold an integer data unit. It's also possible to change the precedence using parentheses.

Example:

```
JMP loop-2
ADD 2*123
FADD 2.0/3   // this will result in a floating point data unit
```

Instruction definition and usage

Instructions are defined using the *def* and *as* keywords using following syntax:

```
def <instruction name> [<arguments signature>] as <instruction machine code layout>
```

Instruction name can contain any letters, numbers, an underscore and a space. It's not possible to use word "as" for an instruction name, as this is a reserved keyword for specifying start of the machine code layout. Only spaces between words are part of the instruction name, so space between *def* and first word as well as spaces between last word and *as* keywords are not part of the name.

Arguments signature can contain any number of arguments, including none. Argument is basically a placeholder for a variable value, which will be placed to the machine code during the assembly process, based on the value specified by programmer when the instruction is used. To specify one argument in the arguments signature, unique number for each argument needs to be specified as well as argument size in bits:

```
{<argument number>:<argument size>[:<value override>]}
```

Instruction machine code layout consists of arbitrary binary data, including the arguments, which are replaced by the value passed to the instruction. To specify a layout of a single argument in the machine code layout, following syntax is used:

```
{<argument number>:<starting bit>:<ending bit>}
```

When specifying argument layout, the argument number is important as it allows programmer to change the order of the arguments in the machine code from the order in the source code. He can also specify specific bits from whole value that will be placed at given position, which allows him to split the argument value into two or more parts and place each part at different place in the machine code if he needs to do so. For example, for an 8 bit argument, if programmer needs to, he can place first bit before the instruction opcode and the remaining 7 bits after the opcode.

Opcode can be specified using any available data unit, it can be of arbitrary length, assembler itself won't pad it to certain width, so it's possible to also create atypical sizes, like 13 bit opcode for example. Programmer can also put one or more special symbols before and/or after the argument and that symbols becomes part of the signature, so when the instruction is used, the symbol must be present too. Arguments must be always separated somehow, at least with a space, because data units themselves need to be separated. It's also possible to "overload" instruction: use same instruction name with different argument signatures, as is demonstrated in following example:

```
def ADD {0:32}, {1:32}    as 2AH{1:32}{0:32}
def ADD {0:64}, {1:8}     as 2CH{1:8}{0:64}
def ADD {0:32}, #{1:32}   as 2BH{1:32}{0:32}
def ADD #{0:32}, {1:32}   as 2BH{0:32}{1:32}
def INC A                 as 35H
```

When an instruction is used, custom assembler tool will try to find the matching definition and create appropriate machine code using the instruction machine code layout, by placing specified opcode to the machine code, as well as specified number of bits of the arguments passed to the function. If instruction requires an argument, it must be specified, using any of the possible data units. If the argument is larger than a data unit, then the extra bits will be filled with zeroes, if it's smaller, it will be truncated. Assembler always tries to find the closest match, meaning if there are two definitions, one with 32 bit argument and second one with a 64 bit argument and programmer uses integer larger than 32 bits, the 64 bit version will be used. Generally, compiler always uses the version that can retain most (if not all) of the supplied data, but that also isn't larger than necessary – it always tries to truncate smallest amount of bits from the argument values as possible, but also make the instruction opcode as small as possible. Programmer may use defined instructions in the example as following:

```
ADD 40H, 12AA56H
ADD 255, #0FFFFFFH
ADD #0EEEEEE, 255
INC
ADD "This is a test", 8
```

When assembled, following data will be produced (without the spaces and newlines):

```
2A 0012AA56 00000040
2B 00FFFFFF 000000FF
2B 00EEEEEE 000000FF
35
2C 08 5468697320697320
```

Comments

custASM uses C/C++ style comments. // indicates a single line comment, /* and */ denote multiline comments. All text marked as a comment is ignored if the symbols exist outside other elements. For example using a comment symbol inside a string will be regarded as usual character, not a start or end of a comment.

Syntax:

```
// single line argument
/* multiline argument */
```

Symbols

custASM allows also defining various symbols, assigning the symbol a specific part of the code and then using the symbol one or more times in various parts of the code, to prevent duplicating code. Unlike some other assembler languages, symbol basically holds textual source code data, which are placed as they are at all places, where the symbol is used. This allows not only to use symbols for holding simple numerical values (data units), but also larger parts of the code, for example groups of instructions. Symbol is defined using the *equ* keyword and specifying the text code in curly brackets. It's possible to specify any valid code in the curly brackets as long as the code will be valid at the place, where the symbol will be used. Symbol is used simply by placing the symbol name at any place in the code, but not before the definition of the symbol. Symbol name must start with a letter and can contain any letters, numbers and underscores.

Syntax:

```
<symbol name> equ { <symbol code> }
```

Example:

```
Value equ { 34C8H } // symbol used to represent a single numerical value

Code equ {          // symbol used to store a part of a code
 ADD 30, 80H
 INC
 INC
}
```

Labels

Label is special form of a symbol, that's defined by specifying a symbol name, followed by a colon. Assembler tool will assign a value to the symbol automatically during the assembly process. Assigned value is by default n 32 bit integer, holding the address of the following byte in the machine code corresponding to the place where is the label defined, which is basically address of the instruction/data specified right after the label. Problem occurs, when programmer uses non-standard sizes for opcodes, data chunks, arguments and such. Then following instruction or data chunk can start in a middle of the byte, instead on the first bit, so the byte address stored in the label symbol isn't precise. Programmer may however use the assembly process settings to enable bit addresses, which specifies the address with a precision to a single bit. Bit address is a 35 bit integer and is used the same way as byte address label, except it holds address of the individual bit, instead of byte.

Labels are the only symbols, which can be also used prior to their definition: if an unknown symbol is found, it's considered to be a label and the data placement is delayed, after everything else is assembled, so all the addresses are known. Label name must start with a letter and can contain letters, numbers and an underscore. Label can be defined only once and it can be used at any part of the source code as usual symbol, it will be replaced with a 32/35 bit integer data unit.

There's also special label, which always holds the address, at which the currently processed block (data chunk or an instruction) started. This label uses a dollar sign and can be commonly used with instructions, for example to create instruction that will cause jump at its own address (thus stopping the program) or for various operations, involving calculating relative addresses for jumps.

Syntax:
```
<labelname>:
```

Example:
```
loop:
```

```
INC A
JMP loop

JMP $        // stop the program execution
```

Overriding argument value

By default, the binary value of the argument is derived directly from the value passed to the instruction when the instruction is used. Programmer may however need to adjust this value somehow, so a method to override the default value (which is equal to value passed to the instruction) is provided. It's optional third specification inside the curly brackets of an argument in arguments signature specification. Programmer may use any valid expression, using simple integer math operations (addition, subtraction, multiplication, division, modulo), with immediate numbers and symbols (including labels). There's a special symbol *val*, which contains the value passed to the argument. Programmer may use it anywhere in the expression as needed, or even completely leave it out. Programmer may also use special symbol $, holding the address of the current instruction.

Example:

```
def SJMP {0:8:$-val} as 80H{0:8} // relative jump
```

Including files

To manage the development easily, it's possible to split larger projects into several files and then include the file in the main source. It's also possible to include pure binary file as is, instead of parsing its textual data containing the sources. When an include statement is encountered, the contents of the file are placed into the main source, like they were directly written there. Of course, this means that the file needs to contain valid code. In binary inclusion, the file binary data are placed in the machine code at the same point, where was the include statement was encountered.

Syntax:

```
include("<file>")   // include a source file
binclude("<file>")  // include a file in binary form
```

Example:

```
include("definitions.casm")
binclude("picture.bmp")
```

Assembly process settings

There are some vital settings for the assembling process, which can significantly change the resulting machine code. It's possible to alter these settings using commands in the language itself, by using the *__set* pseudo instruction, followed by the name of the setting and a value. For example programmer may enable or disable automatic padding of all value to a byte or switch between bit and byte mode for labels.

Syntax:

```
__set <setting name> <value>
```

Example:

```
__set BITADDRESS 1              // enable bit address instead of byte ones for labels
__set DATACHUNK_PADDING 1       // enable byte padding for data chunks
```

List of available settings and default values

Setting	Value	Meaning
BITADDRESS	0 def.	Labels produce 32 bit addresses, pointing to a whole byte
	1	Labels produce 35 bit addresses, pointing to a specific bit
DATACHUNK_PADDING	0	Data chunk size is left as is, even if it's not a multiply of 8

	1 def.	Up to 7 zeroes are added at the end of the data chunk, to make its size a multiply of 8, thus filling a whole byte
INSTRUCTION_PADDING	0	Instruction size (including arguments) is left as is, even though it's not a multiply of 8
	1 def.	Up to 7 zeroes are added at the end of the instruction data, to make its size a multiply of 8, thus filling a whole byte
ATOMIC_SIZE	uint def. 8	Size in bits of an atomic data element, which is significant for the endianess type
LITTLE_ENDIAN	0 def.	Values are stored in big endian format
	1	Values are stored in little endian format
LOGFILE	string	Name and location of the file, where will be the compilation log stored
OUTPUT	string	Allows to override default output filename (<filename>.exe) to anything
LIBRARY	string	Path to a folder, where to look for various include libraries

SIMULATION AND USAGE

Simulation

Together with attoassembler and custom assembler tools, also simulator is provided, which allows stepping the attocode attoinstruction by attoinstruction, thus allowing to debug the program and as well as running the processor at full speed (that is, maximum speed possible on given computer), so the programmed doesn't need to build a physical version of the attoWPU himself, for testing the software. Simulation includes all units, including external units as the text display, bitmap display, LEDs and also input controls, allowing user to input data into the processor and view the output easily. Simulator allows loading data into attocode memory and also program+data memory and dump (export) them at any time (the same goes for displays and dumping (saving) status of the whole processor, including its units).

There are two versions of the simulator: GUI and CLI. First one allows programmer to easily interact and simulate the processor, while the second one can be used for processing and testing from the command line, often with a batch script and it also allows to automatically exporting various data from units into a file.

Usage

The purpose of the attoWPU is mostly experimental and somewhat artistic, trying to find a different, uncommon and unconventional approach to a processor design, assembly programming and programming generally. It can be used as entertainment tool for various programmers, who my try this different approach to programming as well as educational tool, which can help understand some processes involved in the processor function and programming at extreme low level.

AttoWPU can also be used to create various programming competitions, where programmers can try their skills in experimental programming and programming generally. Especially programming the attocode in the attoASM can be often very difficult and it offers some space for optimizing the processor function, so it processes the normal instructions as fast as possible. Another option is to create the attocode as small as possible, thus using as least instructions as possible, while retaining the same function, but probably sacrificing some performance. These two main areas can be used for competitions: programmers may be for example provided with a program in custASM and it's intended output and their task will be to create appropriate attocode, which will either execute as fast as

possible (smallest amount of attoinstructions will be processed when the custASM program is run) or will be as small as possible (having smallest amount of attoinstructions in the assembled attocode as possible).

UNITS REFERENCE

There are many units connected to the buses in parallel. Current unit that will receive command codes is selected by the address bus, thus each unit has its own unique address. Each unit has also its own set of command codes, with maximum of 127 commands, some units may however choose to ignore some of the bits on the control bus if they don't need them (they don't need that many commands), thus eliminating need to clear them if they contain unwanted value from previous command. Number of valid bits is always specified for all buses, all other bits, that are not valid, are always ignored and their value can't change the command or result anyhow.

All units however use the least significant bit (execution bit) in the same way: command is activated only when execution bit changes from 0 to 1. This behavior is uniform for all units. Maximum amount of possible units is 256, but only part of this is used in this version, with possibility for future addition of new units. If nonexistent unit is addressed, simply no unit will receive the command code and no action will occur.

See the graphical representation of attoWPU for more details about how units are connected. Value in parentheses after name of each unit is unit's address on the address bus. Most units have their own internal registers used to control their operation. While these are directly inaccessible, they allow better to understand how given unit functions and control it specified command codes accordingly. For specification of affected registers/bits a C-like syntax is used.

Clock (---)

Clock generates clock signal of arbitrary frequency (depending on used physical circuits or performance of computer running the simulator), that instructs the attocore to process one attoinstruction. Clock cannot be controlled by the processor, it's possible however to alter it manually (especially in the simulator, where slowing clock down significantly allows to step the program). Thus, it has neither an address nor command codes. The pulse is first sent to the aPC so it increments its value and then it's forwarded to the attocore, so it processes currently pointed attoinstruction.

Attocore (---)

Core unit that processes attoinstruction on each pulse from the aPC: first it reads current instruction from the Attoprogram memory, which is directly connected to the attocore, decodes it and then alters value of one bit from buses accordingly. It has no other functionality than this, thus cannot be addressed nor configured using buses. This is the only unit that can write to all the buses.

aPC write (accessed directly through Quick aJump)

This special unit allows quick local jump in the attocode, without affecting any of the other buses. Programmer first writes local address to first 15 bits of the Quick aJump bus and then changes 16th bit (jump bit) to one from zero. Once the aPC write unit detects this change, it will read the 15bit address and write it directly at once into lower 15 bits of the aPC, causing immediate local jump within 32KB block. 15 bits are written only when change from 0 to 1 is detected, continuous value of logical 1 won't cause it to continue writing the value constantly. However, in order to write a new value, it's needed to clear the jump bit and then set it again.

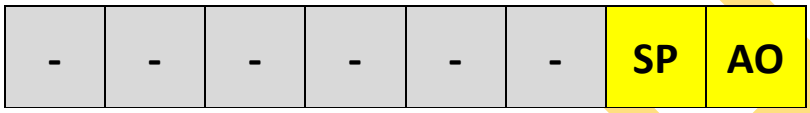
aPC (0x00)

Atto program counter (aPC) holds address of attoinstruction in the attocode memory that is going to be executed. Instruction address is directly outputted to the attocode memory, which then outputs attoinstruction at the address from the aPC directly to the attocore, which then processes the attoinstruction. Each time aPC receives a pulse from the clock, it increments its value by one and forwards the pulse to the attocore, which then processes an attoinstruction. It's possible to write an arbitrary new value using address, control and data buses, as well as instruct it to output current value to the data bus. When a new value is written, one pulse from the attocore will be then ignored; otherwise newly written value would be immediately incremented by one, which is not a desired behavior.

Internal registers

IA instruction address 24 bit Reset value: 0x000000
holds the address of current instruction

CR control register 8 bit Reset value: 0x02



AO address output
when set to 1, value from the IA will be outputted to the data bus (24 valid bits, rest are left unaffected). When 0, no data are outputted: aPC is either in high impedance mode or outputs logical 1 at all bits

SP skip pulse
when set to 1, value in IA won't be incremented when pulse from the attocore is received. Instead, value of SP is cleared, so following pulses are processed normally

Command codes 2 valid bits

Code	Symbol	Action	Affected registers/bits
0x00	STOP	Stop data output	AO = 0
0x01	APC_W	Write new address	IA = (24b)DATA; SP = 1
0x02	APC_O	Output current address from IA	AO = 1; (24b)DATA &= IA
0x03	APC_R	Reset	IA = 0x00000; SP = 1;

Attocode memory (0x01)

This is 16 MB big RWM RAM memory which stores the attocode of the processor, that's being processed by the attocore. Capacity of 16 MB with 8 bit memory unit/cell allows to store precisely 16 777 216 attoinstructions, which should be enough for most purposes. Attocode memory reads an address from the aPC and outputs instruction at this address directly to the attocore. Furthermore, it's possible to read and write data from it using data bus, which allows changing the attocode during program execution, thus allowing creating **self modifying processor**, although such technique will probably be rarely used.

Internal registers

AD address 24 bit Reset value: 0x000000
currently addressed memory cell (1 byte), which will be used for various operations. It's independent on the memory cell addressed by the aPC

CR control register 8 bit Reset value: 0x00

-	-	-	-	-	-	DO	AO

AO address output
when set to logical 1, content of the AD register will be outputted to the data bus

DO data output
when set to logical 1, content of the currently addressed memory cell by AD will be outputted to the data bus

Command codes 4 valid bits

Code	Symbol	Action	Affected registers/bits
0x00	STOP	Stop data output	AO = 0; DO = 0
0x01	AM_AD	Write new address	AD = (24b)DATA
0x02	AM_OA	Output current address	AO = 1; DO = 0; (24b)DATA &= AD
0x03	AM_OD	Output addressed data	DO = 1; AO = 0; (8b)DATA &= *AD
0x04	AM_WR	Write data from the data bus	*AD = (8b)DATA
0x05	AM_WN	Write data from the data bus and increment	*AD = (8b)DATA; AD++
0x06	AM_WP	Write data from the data bus and decrement	*AD = (8b)DATA; AD--
0x07	AM_NX	Move to the next element	AD++
0x08	AM_PR	Move to the previous element	AD--
0x09 … 0x0F		No action	

TEMP register (0x02)

This is a standalone independent register, allowing storing up to 32 bits of data (thus entire value on the data bus). It's also directly connected to the ALU and FPU, where it is used for storing one of the operands. It will be often used to temporarily and quickly store various types of data, both addresses and values. It doesn't contain only data themselves, but also 32 bit mask, which determines which bits will be read from the bus or which bits will be outputted.

Internal registers

DT data 32 bit Reset value: 0x00
stored data

MK mask 32 bit Reset value: 0xFF
I/O mask

CR control register 8 bit Reset value: 0x00

-	-	-	-	-	ME	DO	MO

MO mask output
when set to 1, current I/O mask from the MK register is outputted to the data bus

DO data output
when set to 1, current data are outputted to the data bus, but only bits that are specified by the I/O mask if ME is set. Other bits of the data bus are unaffected.

ME mask enable
When set to 1, input and output data are filtered using the mask in the MK register. If zero, then mask is ignored.

Command codes 4 valid bits

Code	Symbol	Action	Affected registers/bits
0x00	STOP	Stop data output	MO = 0; DO = 0
0x01	TMP_WRM	Write value (with mask)	DT = (DATA&MK) | (DT&~MK)
0x02	TMP_ODM	Output value (with mask)	DO = 1; ME = 1; MO = 0; DATA = (DT&MK) | (DATA&~MK)
0x03	TMP_WR	Write value (without mask)	DT = DATA
0x04	TMP_OD	Output value (without mask)	ME = 0; DO = 1; MO = 0; DATA = DT
0x05	TMP_WM	Write mask	MK = DATA
0x06	TMP_OM	Output mask	MO = 1; DO = 0;
0x07	TMP_ME	Enable mask	ME = 1
0x08	TMP_MD	Disable mask	ME = 0
0x09	TMP_CLR	Clear	DT = 0x00000000
0x0A	TMP_FLL	Fill	DT = 0xFFFFFFFF
0x0B ... 0x0F		No action	

Register memory (0x03)

Special independent small memory allowing to store up to 256 32-bit values (thus allowing storing 1 kB of data), which can be used as additional registers. This memory is independent on the main program+data memory and thus, allows creating storage for various registers that the attocode will use, without affecting the program+data memory. However, the usage of this memory is completely up to programmer, he may use it in any way he wishes to or ignore it altogether and use program+data memory for registers for example.

Internal registers

AD address 8 bit Reset value: 0x00
currently addressed memory cell

PA previous address 8 bit Reset value: 0x00
previously addressed memory cell

MK mask 32 bit Reset value: 0xFFFFFFFF
I/O mask

CR control register 8 bit Reset value: 0x00

-	-	-	-	AO	ME	DO	MO

MO mask output
when set to 1, the content of the MK register is outputted to the data bus

DO data output
when set to 1, the content of the currently addressed memory cell is outputted to the data bus

ME mask enable
when set to 1, then all data outputted to the data bus are filtered using mask in the MK register: if bit in MK is zero, the corresponding bit won't be outputted to the data bus

AO address output
when set to 1, value of the register AD (currently addressed memory cell) is outputted to the data bus

Command codes 5 valid bits

Code	Symbol	Action	Affected registers/bits
0x00	STOP	Stop data output	MO = 0; DO = 0; AO = 0
0x01	RG_AD	Write new address	PA = AD; AD = (8b)DATA
0x02	RG_AO	Output current address	AO = 1; DO = 0; MO = 0; (8b)DATA &= AD
0x03	RG_ODM	Output addressed data (with mask)	DO = 1; AO = 0; MO = 0; ME = 1; DATA &= (*AD&MK) | (DATA&~MK)
0x04	RG_WRM	Write data from the data bus (with mask)	*AD = (DATA&MK) | (*AD&~MK)
0x05	RG_WNM	Write data from the data bus and increment (with mask)	*AD = (DATA&MK) | (*AD&~MK); AD++;
0x06	RG_WPM	Write data from the data bus and decrement (with mask)	*AD = (DATA&MK) | (*AD&~MK); AD--;
0x07	RG_NX	Move to the next element	AD++
0x08	RG_PR	Move to the previous element	AD--
0x09	RG_WM	Write new mask	MK = DATA;
0x0A	RG_OM	Output current mask	MO = 1; DO = 0; AO = 0; DATA &= MK;
0x0B	RG_ME	Enable mask	ME = 1;
0x0C	RG_MD	Disable mask	ME = 0;
0x0D	RG_OD	Output addressed data (without mask)	DO = 1; AO = 0; MO = 0; ME = 0; DATA &= *AD;
0x0E	RG_WR	Write data from the data bus (without mask)	*AD = DATA;
0x0F	RG_WN	Write data from the data bus and increment (without mask)	*AD = DATA; AD++;
0x10	RG_WP	Write data from the data bus and decrement (without mask)	*AD = DATA; AD--;
0x11	RG_RES	Restore address	AD = PA
0x12 ... 0x1F		No action	

ALU (0x04)

This unit allows carrying on arithmetic and logic operations on 32 bit data, such as addition, subtraction, multiplication, division, AND, OR, XOR, NOT and so on. Because two 32 bit operands are needed, ALU uses value on the data bus and value stored currently in the TEMP register. Value from the TEMP register is directly outputted into the ALU, no matter in what mode is the TEMP register in and without any modification using mask. Result is written into the read-only (relative to data bus) register OUT when an execution bit and proper command code is detected.

Only one operation can be carried at the time and the result is always stored in the OUT register, there are no special bits for indicating carry and overflow, nor additional registers for 64 bit results. Instead, each of these operations needs to be done in separate steps, with separate commands. Thus, it's up to the programmer, whether he'll use carry for example or just ignore it and how to handle it (for example store at some register). ALU also contains special functions for decision logic.

Internal registers

None

Command codes 6 valid bits

Code	Symbol	Action	Affected registers/bits
0x00	ZERO	ZERO	OUT = 0
0x01	ADD	ADD (addition)	OUT = (uint)DATA + TEMP
0x02	SUB	SUB (subtraction)	OUT = (uint)DATA - TEMP
0x03	MULL	MUL_LOW (multiplication – lower 32 bits)	OUT = (lower 32b) (uint)DATA*TEMP
0x04	MULH	MUL_HIGH (multiplication – higher 32 bits)	OUT = (higher 32b) (uint)DATA*TEMP
0x05	DIV	DIV (integer division)	OUT = (uint)DATA/TEMP
0x06	REM	REM (division remainder – modulo)	OUT = (uint)DATA % TEMP
0x07	CR	CARRY	OUT = (bool)CARRY(DATA+TEMP)
0x08	BO	BORROW	OUT = (bool)BORROW(DATA-TEMP)
0x09	SADD	SADD (signed addition)	OUT = (int)DATA + TEMP
0x0A	SSUB	SSUB (signed subtraction)	OUT = (int)DATA - TEMP
0x0B	SMULL	SMUL_LOW (signed multiplication – lower 32 bits)	OUT = (lower 32b) (int) DATA*TEMP
0x0C	SMULH	SMUL_HIGH (higher 32 bits, contains sign bit!)	OUT = (higher 32b) (int) DATA*TEMP
0x0D	SDIV	SDIV (signed division)	OUT = (int)DATA/TEMP
0x0E	SREM	SREM (signed division remainder – modulo)	OUT = (int)DATA%TEMP
0x0F	SCR	Signed CARRY	OUT = (bool)sCARRY(DATA+TEMP)
0x10	SBO	Signed BORROW	OUT = (bool)sBORROW(DATA-TEMP)
0x11	ANDB	ANDB (bitwise)	OUT = DATA & TEMP
0x12	ORB	ORB (bitwise)	OUT = DATA | TEMP
0x13	NOTB	NOTB (bitwise)	OUT = ~DATA
0x14	XORB	XORB (bitwise)	OUT = DATA ^ TEMP
0x15	RL	RL (rotate left – with carry)	OUT = (DATA << TEMP) | (DATA >> (32-TEMP))
0x16	RR	RR (rotate right – with carry)	OUT = (DATA >> TEMP) | (DATA << (32-TEMP))
0x17	ANDL	ANDL (logical)	OUT = DATA && TEMP
0x18	ORL	ORL (logical)	OUT = DATA || TEMP
0x19	NOTL	NOTL (logical)	OUT = !DATA
0x1A	XORL	XORL (logical)	OUT = (bool)DATA ^ (bool)TEMP

0x1B	SL	SL (shift left – no carry)	OUT = DATA << TEMP
0x1C	SR	SR (shift right – no carry)	OUT = DATA >> TEMP
0x1D	NAND	NAND (bitwise)	OUT = ~(DATA & TEMP)
0x1E	NOR	NOR (bitwise)	OUT = ~(DATA | TEMP)
0x1F	BOOL	BOOL convert any value to bool (0 or 1)	OUT = (bool)DATA
0x20	MAX	MAX output bigger number into the out	OUT = MAX(DATA, TEMP)
0x21	MAXN	MAXN output which number is larger DATA => 0, TEMP =>1	OUT = TEMP > DATA
0x22	MIN	MIN output smaller number to the out	OUT = MIN(DATA, TEMP)
0x23	MINN	MINN output which number is smaller DATA => 0, TEMP =>1	OUT = TEMP < DATA
0x24	SMAX	SMAX output bigger signed number to the out	OUT = MAX((signed)DATA, (signed)TEMP)
0x25	SMAXN	SMAXN output which signed number is larger DATA => 0, TEMP =>1	OUT = (signed)TEMP > (signed)DATA
0x26	SMIN	SMIN output smaller signed number to the out	OUT = MIN((signed)DATA, (signed)TEMP)
0x27	SMINN	SMINN output which signed number is smaller DATA => 0, TEMP =>1	OUT = (signed)TEMP < (signed)DATA
0x28	EQL	EQL determine whether numbers are equal	OUT = DATA == TEMP
0x29	ZSET	ZSET (zero set – copy DATA value into the OUT only when TEMP is zero)	If(!TEMP) OUT = DATA
0x2A	NZSET	NZSET (non-zero set – copy DATA value into the OUT only when TEMP is non-zero)	If(TEMP) OUT = DATA
0x27 … 0x3F		No action	

OUT register (0x05)

This is 32b standalone, independent, read only (from the DATA bus) register, which is used both by ALU and FPU to store result of an operation. This result can be then outputted to the DATA bus and used wherever needed.

Internal registers

DT store data themselves 32bit Reset value: 0x00000000

CR control register 8bit Reset value: 0x00

-	-	-	-	-	-	-	**DO**

DO data output, when set to one, value stored in the DT is outputted to the DATA bus

Command codes 1 valid bit

Code	Symbol	Action	Affected registers/bits
0x00	STOP	Stop data output	DO = 0
0x01	OUT_D	Output data to the data bus	DO = 1; DATA = DT

FPU (0x06)

This unit allows carrying on mathematical operations with single precision floating point numbers (32bits wide) and storing the output in the OUT register. All values on the DATA and the TEMP bus are considered to be floating point, there's no way of doing a type checking, so it's up to the programmer to fill both TEMP register and DATA bus with correct values, so the calculation has meaning.

Internal registers

None

Command codes 5 valid bits

Code	Symbol	Action	Affected registers/bits
0x00	ZERO	ZERO	OUT = 0.0F
0x01	FADD	FADD addition	OUT = (float)DATA + (float)TEMP
0x02	FSUB	FSUB subtraction	OUT = (float)DATA – (float)TEMP
0x03	FMUL	FMUL multiplication	OUT = (float)DATA * (float)TEMP
0x04	FDIV	FDIV division	OUT = (float)DATA / (float)TEMP
0x05	FSIN	FSIN sine (angle is in radians)	OUT = sin((float)DATA)
0x06	FTAN	FTAN tangent	OUT = tan((float)DATA)
0x07	FEXP	FEXP exponential	OUT = exp((float)DATA)
0x08	FSQRT	FSQRT square root	OUT = sqrt((float)DATA)
0x09	FLOG2	FLOG2 $\log_2$	OUT = log2((float)DATA)
0x0A	FLOG10	FLOG10 $\log_{10}$	OUT = log10((float)DATA)
0x0B	FLN	FLN ln	OUT = ln((float)DATA)
0x0C	FISINF	ISINF is an infinity	OUT = ((float)DATA == 1.#INF) || ((float)DATA == -1.#INF)
0x0D	FTOINT	TOINT convert float to int	OUT = (int)DATA
0x0E	FTOFLT	TOFLOAT convert int to float	OUT = (float)DATA
0x0F	FMAX	FMAX outputs larger number	OUT = max((float)DATA, (float)TEMP)
0x10	FMAXN	FMAXN outputs which number is larger	OUT = ((float)TEMP>(float)DATA)
0x11	FMIN	FMIN outputs smaller number	OUT = min((float)DATA, (float)TEMP)
0x12	FMINN	FMINN outputs which number is smaller	OUT = ((float)TEMP<(float)DATA)
0x13	FABS	FABS outputs absolute value	OUT = abs((float)DATA)
0x14	FPOW	FPOW power	OUT = pow((float)DATA, (float)TEMP)
0x15 … 0x1F		No operation	

Memory controller A (0x07)

This unit provides access to the operating memory, which can store actual program (not the attocode) and data. This unit allows addressing memory cells and reading and writing data. The Memory controller unit is more general and basically same as Memory controller B, the only difference is the underlying unit which is used to store data, which determines how many bytes can be accessed (the size can wary). The memory controller allows using up to 64 bit addressing, but this will be seldom, if ever used, usually up to 32 bits are used in case of the operating memory, which allows 4 GB of the memory, but this is much more than usually needed. Default amount of operating memory is 16 MB, but less or more can be used as needed.

Size of a single accessible memory cell of the operating memory is 32 bits, but smaller units can be used: 16 bit, 8 bit and 4 bit, for example for Memory controller B, 8 bit is used more often, to access external memory unit by a single byte. Size of the cell determines how many bits on the data bus are valid, if a larger memory cell, than the underlying memory can provide, is defined, then the upper bits are zero. If smaller cell is set, then upper bits are in high impedance mode.

Internal registers

AD address 64 bit Reset value: 0x0000000000000000
currently addressed memory cell

SZ memory size 64 bits Reset value: (memory size)
contains number of bytes, that can be addressed (not the number of memory cells, which varies on the cell size)

CR control register 8 bit Reset value: 0xC0

CEH	CEL	-	CO	SO	AHO	DO	ALO

ALO address low output
when set, lower 32 bits of the AD register are outputted to the data bus

DO data output
when set, value at currently addressed location is outputted to the data bus

AHO address high output
when set, upper 32 bits of the AD register are outputted to the data bus

SO size output
when set, content of the SZ is outputted to the data bus, to determine the size of the used memory and thus let the program know, how much of the given memory is available

CO cell size output
when set, currently configured size of a memory cell is outputted to the data bus (2 bit value composed of bits CEL and CEH).

CEL and **CEH** cell size low and cell size high
combination of these two values determines how big the memory cell that's being manipulated with is

CEH	CEL	Cell size	Address multiplier
0	0	8 bit	1
0	1	16 bit	2
1	0	24 bit	3
1	1	32 bit	4

Command codes 5 valid bits

Code	Symbol	Action	Affected registers/bits
0x00	STOP	Stop data output	ALO = DO = AHO = SO = CO = 0;

0x01	M_WRL	Write new lower address	(low 32b)AD = DATA;
0x02	M_OAL	Output current lower address	DO = AHO = SO = CO = 0; ALO = 1; DATA = (low 32b)AD;
0x03	M_OD	Output addressed data	ALO = AHO = SO = CO = 0; DO = 1; (cell size)DATA = *AD;
0x04	M_WR	Write data from the data bus	*AD = (cell size)DATA;
0x05	M_WN	Write data from the data bus and increment	*AD = (cell size)DATA; AD++;
0x06	M_WP	Write data from the data bus and decrement	*AD = (cell size)DATA; AD--;
0x07	M_NX	Move to the next element	AD++
0x08	M_PR	Move to the previous element	AD--
0x09	M_WRH	Write higher address	(higher 32b)AD = DATA;
0x0A	M_OAH	Output higher address	ALO = DO = SO = CO = 0; AHO = 1; DATA = (higher 32b)AD;
0x0B	M_SZ	Output memory capacity (in bytes)	ALO = DO = AHO = CO = 0; SO = 1; DATA = SZ;
0x0C	M_CL	Set memory cell size	CEL = (1st bit)DATA; CEH = (2nd bit)DATA;
0x0D	M_32	Set memory cell size to 32b	CEL = 1; CEH = 1;
0x0E	M_24	Set memory cell size to 24b	CEL = 0; CEH = 1;
0x0F	M_16	Set memory cell size to 16b	CEL = 1; CEH = 0;
0x10	M_8	Set memory cell size to 8b	CEL = 0; CEH = 0;
0x11	M_OCL	Output memory cell size	ALO = AHO = DO = SO = 0; CO = 1; (2b)DATA = CEL | (CEH << 1)
0x12 **…** **0x1F**		No action	

Memory controller B (0x08)

Same as Memory controller A, except it's usually connected to some external memory media, in simulator, this can be some file on the hard drive, which means that usually 48 bit addressing is used, to allow big files (dozens of gigabytes).

SmallQueue (0x09)

This small special memory allows to store up to 32 32bit values from the data bus. The main difference is ability to automatically store or output value from/to the data bus in predefined intervals, allowing programmer to simply just instruct units to output data to the data bus, without having to instruct other units to store these values, this should improve speed of some operations, especially when processing larger amounts of data.

Operation is controlled by the countdown register: when it reaches zero, SmallQueue will start reading/writing data from/to the data memory, until the AD reaches maximum value. Each read/write operation is one every n attocycles, where n is number predefined by programmer in a special register. After each operation, this number n is used to fill countdown register. Programmer may give the countdown register some initial value, different from the value in the fill register.

Internal registers

AD address register 8 bits Reset value: 0x00
contains address of currently addressed cell in the SmallQueue memory (only 5 bits used)

MK mask 32 bits Reset value: 0xFFFFFFFF
mask for reading and writing data

CD countdown 16 bits Reset value: 0x0000
contains current number of attocycles, before next read/write operation

FL fill register 16 bits Reset value: 0x0000
fill register contains value, which will be used to fill countdown when it reaches zero

CR control register 8 bits Reset value: 0x00

AO address output
when set to 1, current value of the AD is outputted to the data bus

DO data output
when set to 1, currently addressed memory cell is outputted to the data bus

DW data write
when set to 1, each time CD reaches zero, current data from the data bus are written into addressed memory location and AD is incremented

ME mask enable
when set to 1, mask is used for read/write operations: mask bits with value 0 indicate that corresponding data bits are ignored and left unchanged

QR queue run
when set to 1, CD register is decremented each step and when it reaches zero, appropriate operations are taken: data read/write (depending on DO and DW, if neither one of them is set, then this operation is skipped), CD is refilled with value from FL, AD is incremented. If AD reaches maximum value, then QR is automatically changed back to 0, thus stopping the function

Command codes 5 valid bits

Code	Symbol	Action	Affected registers/bits
0x00	STOP	Stop data output/input	AO = DO = DW = 0;
0x01	SQ_AD	Write new address	AD = (8b)DATA;
0x02	SQ_OA	Output current address	DO = DW = 0; AO = 1;
0x03	SQ_ODM	Output addressed data (with mask)	AO = DW = 0; DO = 1; ME = 1; DATA = (*AD & MK) | (DATA & ~MK);
0x04	SQ_WRM	Write data from the data bus (with mask)	*AD = (DATA & MK) | (*AD & ~MK);
0x05	SQ_NX	Move to the next element	AD++
0x06	SQ_PR	Move to the previous element	AD--
0x07	SQ_OD	Output addressed data (without mask)	AO = DW = 0; DO = 1; ME = 0; DATA = *AD;

0x08	SQ_WR	Write data from the data bus (without mask)	*AD = DATA;
0x09	SQ_ME	Enable mask	ME = 1;
0x0A	SQ_MD	Disable mask	ME = 0;
0x0B	SQ_O	Enable data output mode	AO = DW = 0; DO = 1;
0x0C	SQ_I	Enable data write mode	AO = DO = 0; DW = 1;
0x0D	SQ_CD	Set countdown	CD = (16b)DATA;
0x0E	SQ_FL	Set fill	FL = (16b)DATA;
0x0F	SQ_R	Start (run)	QR = 1;
0x10	SQ_S	Stop	QR = 0;
0x11 .. 0x1F		No action	

LED control (0x0A)

This is the simplest provided output unit, it allows displaying 32 bit binary value using LEDs, and there are 4 rows of LEDs. Once the value is written, it's kept, until its overwritten: LED control stores it in its own special registers.

Internal registers

ROW0 32 bits Reset value: 0x00000000
ROW1 32 bits Reset value: 0x00000000
ROW2 32 bits Reset value: 0x00000000
ROW3 32 bits Reset value: 0x00000000

Values from these register are directly shown using corresponding row of LEDs.

Command codes (2 valid bits)

Code	Symbol	Action	Affected registers/bits
0x00	LED_R0	Output to row 0	ROW0 = DATA;
0x01	LED_R1	Output to row 1	ROW1 = DATA;
0x02	LED_R2	Output to row 2	ROW2 = DATA;
0x03	LED_R3	Output to row 3	ROW3 = DATA;

Text display controller (0x0B)

This allows to quickly outputting textual information, using a text display with 40x4 characters. Text is written indirectly: display controller stores it in its 160B memory and then periodically updates the display itself, independently on the other parts of the processor. Text has to be in 7b ASCII format (upper 128 values do not have any characters assigned).

Internal registers

AD address 8 bits Reset value: 0x00
currently addressed character (up to 160, addresses above will output zero)

CR control register 8 bits Reset value: 0x00

-	-	-	-	-	-	DO	AO

AO address output
when set to 1, current value of the AD is outputted to the data bus

DO data output
when set to 1, currently addressed character is outputted to the data bus

Command codes (4 valid bits)

Code	Symbol	Action	Affected registers/bits
0x00	STOP	Stop data output	AO = DO = 0;
0x01	TX_ADR	Write character address	AD = (8b)DATA;
0x02	TX_WR	Write character	*AD = (8b)DATA;
0x03	TX_WN	Write character and move to the next	*AD = (8b)DATA; AD++;
0x04	TX_NX	Next character	AD++;
0x05	TX_PR	Previous character	AD--;
0x06	TX_OA	Output current address	DO = 0; AO = 1; (8 valid bits)DATA = AD;
0x07	TX_OD	Output current character	AO = 0; DO = 1; (8b)DATA = AD;
0x08	TX_R	Reset cursor	AD = 0;
0x09	TX_CLR	Clear memory	AD = 0; Clear *AD;
0x0A … 0x0F		No action	

LCD Display Controller (0x0C)

This unit allows displaying bitmap data with resolution 128x128 pixels, thus it can be used for displaying any kind of information. Similarly to the text display controller, unit has its own memory to store the data to be displayed, so it can update the displaying unit independently on the processor and also allow rereading values. Each pixel holds 24 bits of data, 8 bits for each color channel. Bitmap data are addressed linearly, starting on the top left pixel and moving right, if programmer needs to address pixel by its X and Y coordinates, he needs to write his own subroutine to do the necessary calculations.

It's also possible to enable double buffering, where two memories will be used, each one storing a full 128x128x24 image. One memory will be used to write new data, while the second one will be used to display data on the display. Once programmer finishes writing new data, he uses a command code to swap the function of the memories: the new data will be displayed, while the second memory will be available for writing new data. This prevents user form seeing how is the picture rendered pixel by pixel.

Internal registers

AD address register 16 bit Reset value: 0x0000
holds address of current pixel (addresses 24b memory cells, thus maximum value is 16383, if this value is exceeded, it will automatically reset to zero. When double buffering is used, only the write memory is available through this register, the memory used for displaying data is inaccessible by the WPU

CR control register 8 bit Reset value: 0x00

-	-	-	-	BM	BE	DO	AO

AO address output
when set to 1, current value of the AD is outputted to the data bus

DO data output
when set to 1, currently addressed pixel is outputted to the data bus

BE buffer enable
when set to 1, double buffering is used – one memory is used for displaying data, while the second one for writing new data

BM buffer memory
when set to 0, first memory is used to write new data, while the second one for displaying them on screen, when set to 1, the function is swapped

Command codes (5 valid bits)

Code	Symbol	Action	Affected registers/bits
0x00	STOP	Stop data output	AO = DO = 0;
0x01	LCD_AD	Write pixel address	AD = (16b)DATA;
0x02	LCD_WR	Write pixel	*AD = (24b)DATA;
0x03	LCD_WN	Write pixel and move to the next	*AD = (24b)DATA; AD++;
0x04	LCD_NX	Next pixel	AD++;
0x05	LCD_PR	Previous pixel	AD--;
0x06	LCD_AO	Output current address	DO = 0; AO = 1; (16b)DATA = AD;
0x07	LCD_DO	Output current pixel	AO = 0; DO = 1; (24b)DATA = AD;
0x08	LCD_R	Bitmap start	AD = 0;
0x09	LCD_CLR	Clear write memory	AD = 0; Clear *AD;
0x0A	LCD_SB	Single buffer (disable double buffering)	BE = 0; BM = 0;
0x0B	LCD_DB	Double buffering (enable)	BE = 1;
0x0C	LCD_BS	Buffer switch	BM = !BM
0x0D … 0x0F		No action	

Input Controller (0x0D)

This controller provides access to several input methods, allowing inputting data to the processor at the run time, by user. The simplest method is to use a row of switches. Each switch can be only in two states: logical 0 and logical 1, thus each switch corresponds to one bit. There are 4 rows, each having 32 switches, which is the same as the data bus width. Another method, which is however optional, is to use special simplified numeric keyboard, with each key having its own code, corresponding directly to the digit on the key plus one, floating point dot has code 11. Last method, which is however optional and may not be implemented in physical units, is an alphanumeric keyboard:

Input Controller allows reading scan codes of currently pressed keys corresponding to letters. It's possible to read several pressed keys at once, by skipping some keys when scanning.

Numeric keyboard layout

Each key has same scan code as its digit plus one. Thus, zero has scan code 1, one scan code 2 and so on. Scan code 0 means that no key is pressed.

7	8	9
4	5	6
1	2	3
0		. (11)

Reading several scan codes

Keys are scanned in sequential ascending order, by their scan code. Simplest command code returns scan code of the first key found, however another command code allows skipping some keys found during scanning, using 8 bit value from the data bus. If this argument is non-zero, specified number of pressed keys is skipped during scanning. For example, if the argument is 2, then first two pressed keys are ignored and Input Controller continues scanning. If it finds a third key, it returns its value, if not, zero is returned, meaning no third key is pressed.

Internal registers

TD temporary data 32b Reset value: 0x00000000

register used to temporarily store data, which will be outputted to the data bus, programmer first sends a command, which stores input value from selected unit to this register and this value is then outputted to the data bus. This means, that only one register and only one bit is needed to handle data output.

CR control register 8b Reset value: 0x00

-	-	-	-	-	-	BM	DO

DO data output

when set to 1, value in the TD is outputted to the data bus

BM byte mode

when set to 1, only 8 bits are valid when outputting data (this is used for keyboard scan codes, where using 32b is unnecessary)

Command codes (4 valid bits)

Code	Symbol	Action	Affected registers/bits
0x00	STOP	Stop data output	DO = 0;
0x01	IN_DO	Start data output	DO = 1;
0x02	IN_R0	Read switch row 0	TD = SW0; BM = 0; DO = 1;
0x03	IN_R1	Read switch row 1	TD = SW1; BM = 0; DO = 1;
0x04	IN_R2	Read switch row 2	TD = SW2; BM = 0; DO = 1;
0x05	IN_R3	Read switch row 3	TD = SW3; BM = 0; DO = 1;
0x06	IN_RN	Read numkey (no skip)	TD = get_key(NUM, 0); BM = 1; DO = 1;
0x07	IN_SN	Read numkey (skip specified amount of keys)	TD = get_key(NUM, (8b)DATA); BM = 1; DO = 1;
0x08	IN_RK	Read keyboard key (no skip)	TD = get_key(KEYB, 0);

			BM = 1; DO = 1;
0x09	IN_SK	Read keyboard key (skip specified amount of keys)	TD = get_key(KEYB, (8b)DATA); BM = 1; DO = 1;
0x0A … 0x0F		No action	

Timer controller (0x0E)

To precisely measure number of attocycles or milliseconds passed, programmer can use the timer controller unit. This unit provides access to four 16b counting up timers with each attocycle and another special timer, which counts up each millisecond, allowing creating real-time applications. It's possible to start and stop counting timers (real-time one counts all the time), set fill value, read their current value and also read number of overflows as timer controller keeps track of these. By reading current number of overflows occurred, it's automatically reset.

Internal registers

TD temp data 32 bits Reset value: 0x00000000
used for capturing timer data and outputting them to the data bus

T0, T1, T2, T3 timer 16 bits Reset value: 0x0000
timer data (current value)

TF0, TF1, TF2, TF3 timer fill 16 bits Reset value: 0x0000
these registers hold fill value for corresponding timers (they are filled with this value each time they overflow)

OC0, OC1, OC2, OC3 overflow count 16 bits Reset value: 0x0000
holds number of overflows occurred

RT real time 32 bits Reset value: 0x00000000
this register holds number of milliseconds passed since the processor started

CR control register 8 bits Reset value: 0x00

TR0	TR1	TR2	TR3	-	-	WM	DO

DO data output
when set to 1, value stored in TD is outputted to the data bus

WM word mode
when set to 1, only 16b of the 32b are valid when data are outputted to the data bus

TRx timer x run
when set to 1, corresponding timer runs

Command codes (5 valid bits)

Code	Symbol	Action	Affected registers/bits
0x00	STOP	Stop data output	DO = 0;
0x01	TI_DO	Start data output	DO = 1;
0x02	TI_TR0	Run timer 0	TR0 = 1;

0x03	TI_TR1	Run timer 1	TR1 = 1;
0x04	TI_TR2	Run timer 2	TR2 = 1;
0x05	TI_TR3	Run timer 3	TR3 = 1;
0x06	TI_TS0	Stop timer 0	TR0 = 0;
0x07	TI_TS1	Stop timer 1	TR1 = 0;
0x08	TI_TS2	Stop timer 2	TR2 = 0;
0x09	TI_TS3	Stop timer 3	TR3 = 0;
0x0A	TI_TF0	Fill timer 0	T0 = (16b)DATA;
0x0B	TI_TF1	Fill timer 1	T1 = (16b)DATA;
0x0C	TI_TF2	Fill timer 2	T2 = (16b)DATA;
0x0D	TI_TF3	Fill timer 3	T3 = (16b)DATA;
0x0E	TI_AF0	Set timer 0 auto fill	TF0 = (16b)DATA;
0x0F	TI_AF1	Set timer 1 auto fill	TF1 = (16b)DATA;
0x10	TI_AF2	Set timer 2 auto fill	TF2 = (16b)DATA;
0x11	TI_AF3	Set timer 3 auto fill	TF3 = (16b)DATA;
0x12	TI_OC0	Output timer 0 overflows	TD = OC0; DO = 1; WM = 1; OC0 = 0; (16b)DATA = TD;
0x13	TI_OC1	Output timer 1 overflows	TD = OC1; DO = 1; WM = 1; OC1 = 0; (16b)DATA = TD;
0x14	TI_OC2	Output timer 2 overflows	TD = OC2; DO = 1; WM = 1; OC2 = 0; (16b)DATA = TD;
0x15	TI_OC3	Output timer 3 overflows	TD = OC3; DO = 1; WM = 1; OC3 = 0; (16b)DATA = TD;
0x16	TI_OV0	Output timer 0 value	TD = T0; DO = 1; WM = 1; (16b)DATA = TD;
0x17	TI_OV1	Output timer 1 value	TD = T1; DO = 1; WM = 1; (16b)DATA = TD;
0x18	TI_OV2	Output timer 2 value	TD = T2; DO = 1; WM = 1; (16b)DATA = TD;
0x19	TI_OV3	Output timer 3 value	TD = T3; DO = 1; WM = 1; (16b)DATA = TD;
0x1A	TI_ORT	Output real time timer value	TD = RT; DO = 1; WM = 0; DATA = TD;
0x1B … 0x1F		No action	

Speaker Output (0x0F)

This is very simple sound output device, allowing to produce simple beep sounds with given frequency. To produce square audio waveform with specified frequency, programmer simply writes the frequency multiplied by four to the special register and the unit immediately starts generating sound. To stop the sound output, simply write value 0 to the register.

Internal registers

FRQ frequency 16b

contains the frequency of sound (square waveform) to be generated, multiplied by four. If the value is zero, then no sound is generated.

Command codes (0 valid bits)

Code	Symbol	Action	Affected registers/bits
anything	---	Write the frequency	FRQ = (16 bit)DATA;

GLOSSARY

List of various terms important to the attoWPU processor.

- **Address bus**
  Bus consisting of 8 bits controlled by the attocore, used for addressing an internal unit in the processor
- **atto Program Counter (aPC)**
  Special register containing the address of the attoinstruction, that will be executed by the attocore with the next pulse from clock
- **Attoassembler**
  Tool used to generate an attocode from source codes in the attoASM language
- **Attoassembly (attoASM)**
  Programming language designed to write programs for the attoWPU's core, which is then assembled to a attocode
- **Attocode**
  Machine code, usually stored in the attocode memory, consisting of an attoinstruction opcodes, processed by the attocore
- **Attocode memory**
  Memory with capacity of 16 MB, allowing to store up to 16 777 216 attoinstructions. The attocore reads the current instruction directly from the memory and executes it
- **Attocore**
  Crucial unit of the attoWPU. It reads an attoinstruction pointed by aPC in each cycle and executes it: alters value of one of the 64 bits
- **Attoinstruction**
  A special instruction for the attoWPU's core, containing a number of a bit to change and its new value, it can only change one bit at the time
- **AttoWPU**
  Name of this experimental processor, from the WPU (Weird Processing Unit) series
- **Control bus**
  Bus consisting of 8 bits controlled by the attocore, used for sending a command codes to various units in the processor
- **Custassembler**
  Tool used to generate a machine code from source codes written in custom assembly
- **Custom assembly (custASM)**
  Special programming language, allowing programmer to define his own instructions and opcodes. It can be used to create actual program, that will be executed by the attocode
- **Data bus**
  Bus consisting of 32 bits controlled by the attocore and units in the processor, used for exchange of data between attocore and some unit, or units themselves
- **Execution bit**
  Least significant bit on the control bus. Change from 0 to 1 indicates, that command code on the control bus has to be executed by the currently addressed unit

- **Jump bit**

  Least significant bit on the Quick aJump bus. Change from 0 to 1 indicates, that 15 bits have to be written to the lower 15 bits of the aPC at once

- **program+data memory**

  Large memory (size may vary with implementation) used to hold data and also the program executed by the attocore if programmer chooses to

- **Quick aJump**

  Special bus, consisting of 16 bits controlled by the attocore, allowing to create a quick local jump within a 32 KB block in the attocode, without using other three buses

- **Unit**

  A logical part of the processor, performing some specified task and usually responding to various command codes when addressed

- **WPU**

  Acronym for Weird Processing Unit, a series of experimental processor using new, unconventional and weird approaches to assembly programming and programming in general, allowing programmers to explore new ways of programming for fun, curiosity, and education and similar. Processors have even somewhat artistic intent, as they are rather unique creations, but not intended for normal production use

AUTHORS

attoWPU specification, design and programming

Tomáš "Frooxius" Mariančík

E-mail: Tomas.Mariancik@gmail.com
Website: https://patreon.com/frooxius
WLM/MSN: Tomas.Mariancik@hotmail.com
Skype: Frooxius
Discord: Frooxius#4483
Telegram: @Frooxius