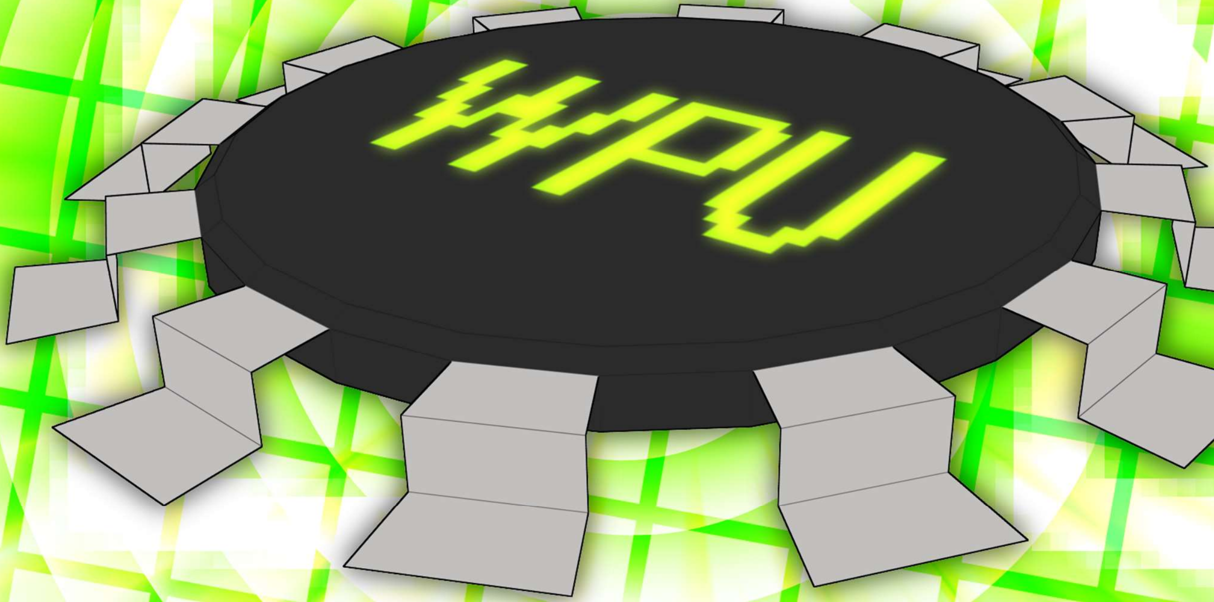
2D WPU SPECIFICATION

Designed and written by:

Tomáš “Frooxius” Mariančík



https://patreon.com/frooxius

Contents

Overview

2DWPU is experimental processor architecture from the WPU (Weird Processing Unit) series with accompanying programming language 2DASM. WPU's attempt to approach the processor architecture design, machine code processing and programming with unusual, unconventional, playful and creative way for several purposes, including curiosity, education, fun and even an artistic intent.

This experimental architecture changes the conventional program flow by logically arranging instructions into a two dimensional grid and making these instructions query their surrounding instructions in four directions and returning a value after they are done. Programs flow is them moving around this two dimensional grid, expanding in various directions, retracting back and "bouncing" into expansion again at some point, usually in a different direction.

Program flow presented by 2DWPU brings various challenges, as having to design various algorithms and loops differently than with traditional architectures or having to optimally arrange the instructions in a 2D grid in addition to writing the program itself, which also makes optimization for both size and speed more difficult and challenging to the programmer. However, thanks to the top-down programming model, some aspects of the assembly programming are simplified.

Furthermore, it presents an option for automatic code parallelism even when evaluating simple expressions, as instructions can branch into two separate execution threads and later join back, after result for each branch is computed. Depending on the amount of arithmetic cores, the two separate branches can furthermore branch into more branches executed in parallel and join afterwards.

Two dimensional grid can hold 32x32 instructions and it's called a program block, thus a program block can contain up to 1024 instructions. There can be several programs blocks stored in the memory, but jump from one block to another clears the expansion path, so any parameters must be passed from block to another block using registers or the memory.

Memory addressing is 24 bit, allowing up to 16 MB of RAM which holds both program and data. However largest possible addressable memory unit is 32 bits and processor internally works with 32 bit values.

Part of the 2DWPU is a IO interface, which is used for communicating with various external devices for input and output. There is a set of default devices defined by this specification, that should be present with every 2DWPU implementation at fixed addresses providing a standard base environment for creating programs, however the interface is expansible and it's possible to implement and connect more devices of any design.

PROGRAM FLOW

2DWPU has a different program flow from ordinary architectures. Instructions are logically organized into a two dimensional grid 32 by 32 allowing up to 1024 instruction in this grid. One block of 32x32 instructions is called a program block. Although logically are instructions in a 2D grid, they're stored linearly in the memory the same way a two dimensional array is stored: one row of instructions after another.

Program Counter

The program counter, a register that determines which instruction to execute, must take into consideration the atypical arrangement of instructions and translate two dimensional position of instruction into the linear memory address. To achieve this, the program counter consists of three registers:

• BS	24 bit	Block Base
• xPC	5 bit	x-axis Program Counter
• yPC	5 bit	y-axis Program Counter

Block Base register holds the address, where current program block starts, xPC and yPC register then determine X,Y position of current instruction in the grid. Together, they form a 34 bit value, that refers to a single instruction. Before the actual block of instructions is a 64 byte program block header, which holds various additional information about the current program block. The linear address in the memory is then calculated according to the following formula:

$$CA\ = 64 + \ BS + xPC \times 2 + (yPC \times 32) \times 2$$

This value is then passed to the 2DWPU core, which loads the instruction pointed by the calculated address (CA) and processes it.

Instruction calls and returns

While BS register remains unchanged for most of the time, xPC and yPC are altered almost in every cycle, so execution moves to a different instruction. Which instruction will be executed next is always determined by the current instruction, which can call one of its surrounding instructions or return to previous instruction. This means, that each instructions alters the xPC and yPC registers.There are four possible directions and fifth special, they are denoted by following letters:

• L	left
• R	right
• U	up
• D	down
• C	continue
• *(none)*	*no specified direction*

Directions L, R, U, D are obvious: instruction simply transfers the program flow to an instruction in that direction, relative to itself. Direction C is similar, only the actual direction is determined on runtime: instruction will simply continue in the direction it was called. If it was called from the left, it will call instruction on the right, if it was called from the bottom, it will call the instruction above.

Some instructions have no direction specified, these are used mostly in conjunction with double merge instructions, where instruction calls another instruction inside the field, thus in no specific direction (the xPC and yPC don't actually change during this).

At some point, every instruction returns – transfers the program flow back to the instruction that called it and returns a value, that can be used by the previous instructions.

Jumping over instructions

When a surrounding instruction is called, it's possible to jump over several instruction and not call the instruction right next to the current one, but second, third or fourth instruction in the given direction. This means, that xPC or yPC is incremented/decremented by 2, 3 or 4. When returning, the instructions in the gap between these two instructions are jumped over again. This adds more flexibility to the algorithm design and instruction arrangement. If one instruction calls in two directions, it's possible to apply this jump only to one of the two directions, because of the limitation in the instruction word size (see Instructions chapter).

Instruction status

Each instruction in the grid can have a status, that is stored in the IS register (Instruction Status) as a 4 bit number, allowing up to 16 different statuses. This status determines, what will the instruction do when it is executed. Instructions usually have several phases of execution, in each phase, they perform a different action.

Status is required, because when program flow returns to an instruction that was already executed before, it must know that it is not executed for the first time and perform a different action, for example returning too, otherwise the program would loop infinitely. Once instruction returns its status is lost, because it's not needed anymore. However it's required to know the status of every instruction in the program path determined by calls.

For example a simple query instruction QRY_D: when it's called, its status is zero by default. It sets the status to one and calls instruction at the bottom. Once the program flow returns to this instruction it's executed, but this time, its status is one. This instructs the instruction to perform a different action: return.

Instruction Stack

Because it is needed to store a direction, in which the instruction was called, so it knows where to return and also status of every called instruction in the program flow path an Instruction Stack is provided. It is a 1 kB stack, which can hold up to 1024 byte values. One byte is capable of holding all required information for a single instruction.

First 4 bits are used to store the return direction: 4 directions * 4 lengths of the jump = 16 combinations. Second 4 bits then store the instruction status. When instruction calls another one, then these data are pushed onto the stack and popped at return. This byte is called Instruction Flow Word.

RD1	RD0	RL1	RL0	PDF	IS2	IS1	IS0
Return information				Instruction Status			

- RD0, RD1 Return Direction
  Determines in which direction to return – where to transfer program flow.

RD1	RD0	Direction
0	0	Up
0	1	Right
1	0	Down
1	1	Left

- RL0, RL1 Return Length
  Determines how many instructions will be skipped over during the return – what number will be added/subtracted from the xPC/yPC.

RL1	RL0	+/- from xPC/yPC
0	0	1
0	1	2
1	0	3
1	1	4

- IS3, IS2, IS1, IS0 Instruction status
  4 bit number that determines the status of the instruction, allowing up to 16 different statuses.

Register that points to the top of the Instruction Stack is called SI, which stands for Stack of Instructions. Its reset value is zero, which indicates an empty instruction stack. It is an 10b register, which means that it can address 1KB of a single byte cells.

Instruction Flow Word is pushed into the instruction stack each time an instruction calls an another one, storing the return direction and the status of the calling instruction. When instruction returns, the Instruction Flow Word is popped from the top of the stack – the return direction is used to calculate a new value for the xPC and yPC and the Instruction Status is loaded into the Instruction Status register, where it can be read by the 2DWPU core to determine the action that will the current instruction perform.

Argument stack

Apart from the Instruction Stack, there is also Argument stack, which is also 1 kB, but can hold only 256 values, because each one of them is 32 bit. This stack is reserved only for instructions and cannot be used directly. Instructions may store argument values in this stack for later use, for example when operation instruction collects several argument to perform an operation on them. The register that points to the top of this stack is called SA – Stack of Arguments and it is 8 bit.

Possible directions and branching

Each instruction can call instruction only in one of the four ways and return, when the program flow returns back t or it can call another instruction in any of the directions. Instruction Status is used to determine this: when instruction, which calls into two directions, is entered, it will set its status to one and call the first direction. When the execution returns to this instruction, it will read its status, which is 1 and it will know, that it already called the first direction. Then it sets it status to 2 and calls the other direction, when the execution returns again, it will return based on the status 2. Using the same way, it's possible to create instructions that call in three or four directions.

Instructions in the base instruction set call only one or two directions because they are encoded in the instruction opcodes (see the Instructions chapter). When calling in several directions the order of the calls is important and it is possible to call twice in the same direction. The directions are specified as sequence of letters, with order from left to right. For example:

D	call down
DU	call down, then up
UD	call up, then down
RR	call right, then call right again
UC	call up, then continue in the direction the instruction was called from initially
DRUD	call down, then right, then up, then down again

Program start, edge overflow and jump

The reset value of the registers BS, xPC and yPC are zero, so when the processor is started or reset, it will execute the instruction at position [0;0] at the program block located at the beginning of the machine code. This instruction then determines, in which way the execution will continue. If the first instruction returns, that is, the instruction stack is empty when an instruction tries to return, processor will call the instruction at the current position, that is, the same instruction that tried to return. This means, that if programmer does not halt the processor himself using special instruction or a program construct, it will loop indefinitely.

When instructions tries to call instruction over the edge of the grid, it will simply jump over the edge and execute the instruction on the other side, because the xPC or yPC register simply overflows or underflows: if it xPC contains value 31 and instruction calls instruction on the right it will overflow to 0 executing the instruction at the opposite edge. Edges of the logical grid do not represent a border that cannot be crossed, the grid is rather continuous.

Because it's possible to have more then one program block, then it's needed to somehow move between these blocks. This is done via jumps, which rewrite the values of xPC, yPC and most importantly BS to a different value. This will cause the program execution to jump to a specific location of another block. However, the instruction stack is cleared in this process, so it's not possible to call an instruction from another block and then return to the previous block when the instruction returns: the execution is simply started anew in the different location, however the registers are preserved and can be used to transfer control.

PARALLELISM

One of the 2DWPU's main goals is achieving parallelism during the execution, which can be done by instruction branching – in case an instruction requires two results to calculate a value, both branches can be executed at the same time, independently of each other and compute the results (basically operands) for given instruction. There's always one main execution core and several additional cores, that allow the instructions to branch, assuming there's an available core. These additional cores are more limited compared to the main core though.

???

Program data organization

Each program block is defined as a block of instructions with size of 32x32 instructions. However, additional data is stored together with this block the the memory and required by the processor. These data include the program block header stored before the block and indexed table stored after the block.

Program block header

Each program block is preceded by 64 bytes of data (basically equal to one row of the instructions) that describe the program block, determine how exactly it will be executed and allows to implicit limitations on the code within the block. Programmer can use it to create safe subroutines that can alter only limited memory range and thus do not corrupt other data in case of malfunction. Additionally it's possible to specify how exactly will the block be executed, whether parallelism will be enabled and such. There's a lot of space reserved for future additions.

The header is structured as following:

PBF	-	LMB	UMB	LIB	UIB	LCB	UCB	CRB	*Reserved*
4B	4B	4B	4B	2B	2B	4B	4B	5B	31B

- **PBF** **Program Block Flags** **4 bytes**
  - This area contains various settings for the current program block, which are described later
- **LMB** **Lower Memory Bound** **4 bytes**
- **UMB** **Upper Memory Bound** **4 bytes**
  - Values stored at these locations determine the lowest (LMB) and highest (UMB) program and data memory address that the instructions in the program block can access
  - 3 bytes are used as the program and data memory address is 24 bit value, however it allows possible future expansion to 32 bits
- **LIB** **Lower IO Bound** **2 bytes**
- **UIB** **Upper IO Bound** **2 bytes**
  - Values stored at these locations determine the lowest (LIB) and highest (UIB) IO interface address that the instructions in the program block can access
- **LCB** **Lower Call Bound** **4 bytes**
- **UCB** **Upper Call Bound** **4 bytes**
  - Values stored at these locations determine the lower (LCB) and highest (UCB) program and data addresses that can be target of a program block call
- **CRB** **Call Return Block** **5 bytes**
  - This block contains 34bit value, which contains values for BS, xPC and yPC registers
  - These values will be used to call a block once the current block returns, this allows programmer to create one primary block, which can call other blocks that will automatically jump to the calling block back
  - The default block usually points to itself, so when the first instruction returns, then it will be called again
  - LCB and UCB do not apply to this address
  - **CRB** is structured as following

yPC	xPC	BS

30…34	25…29	0…24

Program Block Flags

Individual bits in the program block flags determine various aspects of the following program block execution. Program Block Flags area is structured as following:

											MH	-	-	-	BE

- **BE** **Branching Enabled**
  - If set, automatic branching is enabled, otherwise the code will be executed utilizing only one core
- **MH** **Modify Header**
  - If set, program block can access its own program block header, otherwise the access is restricted

BASIC 2DINSTRUCTIONS

2DWPU has unconventional instructions: they call one or more surrounding instructions and some of them process the returned value and at some point, each instruction returns – transfers the program flow to the instruction that called it. Each instruction has also a set of instruction statuses, which changes what the instruction does when it is executed. This is required, because the work of each instruction is divided into several phases and between each one of them a different instructions are usually executed.

2DWPU instructions can be called 2Dinstructions to signify their unique properties.

Categorization

There are several types of 2Dinstructions, based on their operation, meaning of the individual bits of the opcodes and existing phases. 2Dinstructions are divided as following:

- Query
  - Single
  - Double
- Operation
  - Single
  - Double
- Passthrough action
  - Entry
  - Return
  - Bidirectional
  - Immediate
- Index
  - Return
    - Register
    - Value

        - Memory
        - IO interface
    - Double
        - Merge
        - TwoDir
    - Jump
    - Extension

Query, Operation, Passtrough action and Index are called basic types, other types are called subtypes.

Opcode

Every instruction is 16 bits long and consists only of an opcode. Any possible argument is embedded to the opcode of the instruction, because it's not possible to store the argument in the program block, since every instruction must be exactly 16 bits wide. This is because the program block has to have a fixed size, otherwise this would cause execution problems during the calculated address calculation, because it cannot work with variable length instructions, because the calling instruction has no knowledge of the size of the called instruction, so it must assume an uniform value.

The opcode is divided into several parts, allowing to easy and quickly determine the type of the instruction. The division differs for various basic types of instructions, because they have different subtypes.

BT1	BT0	TS	TS	TS	TS	TS	TS	TS	TS	TS	TS	TS	TS	TS	TS

- BT0, BT1 Basic Type
  These bits determine the basic type of the instruction according to the following table:

BT1	BT0	Basic type
0	0	Query
0	1	Operation
1	0	Passtrough action
1	1	Index

- TS Type Specific
  These bits are specific to each basic type of an instruction. Read about specific basic types for further information about the function of these bits.

Query instructions

Instructions with basic type query simply query surrounding instructions, but do not alter the argument. They are used to manage program flow, create branches and loops. They can query in one or two directions and they can also query based on some condition.

Opcode

The opcode of the query instructions is structures as following:

0	0	D	D	D	D	D	D	D	D	QI5	QI4	QI3	QI2	QI1	QI0

- D direction
  These bits encode the directions, in which will the instruction call the surrounding ones. This part is common with the Operation instructions, please refer to the Encoding directions for Query and Operation for more information about how are the directions encoded.
- QI0…QI5 Query Instruction
  These bits determine the exact type of an query instruction, this type does not determine the direction, because this part is determined by the D bits, since every type of query instruction can use any possible combination of directions.
  Because 6 bits are allocated for the specific type of query instruction, there can be up to 64 unique query instructions, not considering the directions.

Single

Double

Operation instructions

Operation instructions are similar to the Query instructions: they query in one or two directions, however they do alter the argument and they always query unconditionally. These are used for various operations with operands, they usually use the values returned by the surrounding instructions to calculate a new value.

Opcode

The opcode structure is similar to query instructions:

0	1	D	D	D	D	D	D	D	D	OI5	OI4	OI3	OI2	OI1	OI0

- D direction
  These bits encode the direction in which does the instruction call others, for details about how is the direction encoded for the Query and Operation instructions, please refer to the chapter Encoding directions for Query and Operation.
- OI0…OI5 Operation Instruction
  These bits determine a specific Operation Instruction – they determine the type of the operation that will be performed. Up to 64 different operations are possible, excluding the direction variations.

Single

Double

Encoding directions for Query and Operation

Query and Operation instructions can call surrounding instructions once or twice in four directions and one special case direction. Furthermore, it is possible to skip over up to three instructions in the given direction. All possible combinations need to be encoded in the opcode of these instructions in a meaningful way, using the directions bits. They are structured as following:

0	0/1	CD4	CD3	CD2	CD1	CD0	CL1	CL0	LD	x	x	x	x	x	x

- CD0…CD4 Call Direction

These five bits determine the directions in which will the instruction call as well as whether it calls once or twice. Directions can be considered as a numeric base 5 with symbols: URDLC that represent digits 0 to 4. The 5b number represented by these bits is then "converted" to this base 5 with aforementioned symbols, to get corresponding set of directions. For example, number 14 in base 5 will be written as RC.

For the single call, there are 5 possible scenarios. For two calls, there are 25 possible variations (calculated as 5^2). The allocation of decimal numbers represented by the CDx bits is following:

- 0…24 two directions
- 25…29 one direction
- 30…31 unused

To calculate one direction, following formula is used:

$$D = (CD - 25)_5$$

To calculate two directions, following formula is used (all values are considered as two digit, thus first five have zero prefixed). The first digit is the second direction and the second digit is the first direction:

$$D = CD_5$$

Here is a complete table of values and corresponding directions.

Bin	Dec	Dir	Bin	Dec	Dir	Bin	Dec	Dir	Bin	Dec	Dir
00000	0	UU	01000	8	RL	10000	16	LR	11000	24	CC
00001	1	UR	01001	9	RC	10001	17	LD	11001	25	U
00010	2	UD	01010	10	DU	10010	18	LL	11010	26	R
00011	3	UL	01011	11	DR	10011	19	LC	11011	27	D
00100	4	UC	01100	12	DD	10100	20	CU	11100	28	L
00101	5	RU	01101	13	DL	10101	21	CR	11101	29	C
00110	6	RR	01110	14	DC	10110	22	CD	11110	30	-
00111	7	RD	01111	15	LU	10111	23	CL	11111	31	(none)

It is possible to easily detect, whether the direction is a single one or two directions. A single direction is detected as following, to detect the two directions, simply invert the detection of a single direction (SD):

$$SD = (CD4 \wedge CD3) \wedge (CD2 \vee CD1 \vee CD0)$$

- CL1, CL0 Call Length
  Determines how many instructions to skip when calling instruction in a given direction – what number will be added/subtracted from the xPC or yPC according to the following table:

CL1	CL0	+/- from xPC/yPC
0	0	1
0	1	2
1	0	3
1	1	4

- LD Long Direction
  Determines to which one of the two call directions will be the Call Length applied to, the other one is always the simple one. This applies also to the single call: if LD tells the processor to apply the long call to the second direction while there is only one, it will apply to the second, nonexistent one, so it will have no effect.

LD	Apply long call to
0	First direction
1	Second direction

Passtrough action instructions

These instructions perform a certain action independently on the argument – they do not usually modify it, they simply perform an argument unrelated operation whenever the execution passes through them. These operations can include moving data from/to registers, memory locations, IO interface, manipulating registers and so on. The instructions unconditionally query only in one direction – they simply pass on the execution flow. Based on when is the action executed, passtrough action instructions are divided into following categories with corresponding symbols:

- > entry
  Action is executed when is the instruction called first
- < return
  Action is executed when the instruction returns
- = bidirectional
  Action is executed both when is the instruction called and when it returns
- | immediate
  Action is executed when is the instruction called and then the instruction immediately returns, instead of calling another instruction

Opcode

The opcode of passtrough action instructions is structured like this:

1	0	ET1	ET0	CL1	CL0	CD2	CD1	CD0	AI6	AI5	AI4	AI3	AI2	AI1	AI0

- ET0, ET1 Execution Time
  Determines the subtype of the passthrough action instruction – when will be the action executed, according to the following table:

ET1	ET0	Subtype
0	0	Entry
0	1	Return
1	0	Bidrectional
1	1	Immediate

- CL0, CL1 Call Length
  Determines how many instructions to skip when calling an instruction – what number will be added/subtracted to/from xPC/yPC according to the following table:

CL1	CL0	+/- from xPC/yPC
0	0	1
0	1	2
1	0	3
1	1	4

- CD0...CD2 Call Direction
  Determines direction of the call according to the following table:

CD2	CD1	CD0	Direction
0	0	0	U
0	0	1	R
0	1	0	D
0	1	1	L
1	0	0	C
1	0	1	*Undef.*
1	1	0	*Undef.*
1	1	1	(none)

- AI0...AI6 (Passtrough) Action Instruction
  Encodes a specific passtrouch action instruction and thus determines the specific action, that will the instruction perform, not considering the call direction and execution time. Seven bits allow up to 128 unique actions.

Index Instructions

These instructions operate with 32 bit values from the Indexed Data Table and interpret them in various ways, the index key that refers to a specific value is encoded in the instruction opcode.

Indexed Data Table

Because the instruction opcodes in the program block cannot contain arguments to preserve uniform length of the instruction opcode, these arguments are stored in the indexed data table. It's a variable long block of binary data stored right after the corresponding program block in the memory. Each value is 32 bit and there can be up to 1024 values in the table, making the block up to 4 KB large, although this will rarely happen as only the values that are needed are stored. To calculate the starting address of the Indexed Data Table for a current program block, following formula is used:

$$IDT = BS + 64 + 2048$$

Accessing the table

Only instructions called Index instructions can access the data stored in this table, because their opcode contains a 10b index key, which allows to select one of the 32 bit values from the table, that will the instruction load and process. To calculate address of a specific entry for an Indexed Data Table for the current program block, following formula is used:

$$DATA = (BS + 64 + 2048) + (INDEX \times 4)$$

Index Instruction Subtypes

Based on how is the 32bit value from the Indexed Data Table interpreted, instructions are divided into following subtypes:

- Return
- Jump
- Double
- Extension

Opcode

Index Instructions have opcode structured as following:

1	1	II1	II0	SS	SS	IK9	IK8	IK7	IK6	IK5	IK4	IK3	IK2	IK1	IK0

- II0, II1 Index Instruction
  Determines the subtype of the index instruction according to the following table:

II1	II0	Subtype
0	0	Return
0	1	Jump
1	0	Double
1	1	Extension

- SS Subtype Specific
  These bits are specific to each one of the Index Instruction subtypes, please refer to a specific subtype to learn about function of these bits
- IK0...IK9 Index Key
  These instructions hold a 10 bit value, ranging from 0 to 1023, allowing to address up to 1024 different entries to be selected from the Indexed Data Table

Return index instructions

These instructions immediately return when they are called and they return a specific 32-bit value, that is extracted from the Indexed Data Table. Based on what role the 32-bit value plays when returning a value, Return instructions are divided into following subtypes:

- Register
  Index key is simply used to specify one of the processor's registers and does not correspond to a value in the table. Value of specified register is returned.
- Value
  This instruction returns the actual value stored in the Indexed Data Table.
- Memory
  The value in the table is used as address for the program and data memory, instruction returns value stored at this address.
- IO
  The value in the table is used as address for the IO device for communication with devices, instruction returns value at the given address.

Subtype of a return instruction is determined by the Subtype Specific bits as following:

1	1	0	0	RI1	RI0	IK9	IK8	IK7	IK6	IK5	IK4	IK3	IK2	IK1	IK0

- RI0, RI1 Return Instruction
  Determines subtype of the return instruction according to the following table:

RI1	RI0	Subtype
0	0	Register
0	1	Value
1	0	Memory
1	1	IO

Jump Index Instructions

When a jump to another program block is needed, jump index instruction is used. The value from the Indexed Data Table is combined with Subtype Specific bits to create a 34 bit value – new address that will be written to the BS, xPC and yPC registers (24 + 5 + 5 = 34).

1	1	0	1	AB1	AB0	IK9	IK8	IK7	IK6	IK5	IK4	IK3	IK2	IK1	IK0

- AB0, AB1 Address Bit
  These bits are combined with the value from the table to create a 34 bit value – new address in a following way:

AB1	AB0	Value from the table (32-bits)

This 34bit value is interpreted as following:

BS	yPC	xPC

Double Index Instructions

These instructions hold a lot of power, because they interpret the 32-bit value in the table as two 16-bit instructions, thus allowing to place two different instructions into one cell of the grid. Based on how are these two instructions executed, the Double Index Instruction had two subtypes:

- Merge
  The first instruction is executed when the Double instruction is called and when it tries to call the surrounding instruction, the second instruction in the Double instruction is actually called. When the program flow returns to the Merge Double instruction, they are called in reverse order and if the first one calls again, the process repeats.
- TwoDir
  The first instruction is executed first and the second one is ignored. Once the program flow returns to this instruction and the first instruction returns, the Double Index Instruction prevents from actually returning and executes the second instruction and keeps executing it until it returns too. Thus the first instruction is called when the Double instruction is entered and the second instruction is called after the first one returns.

The opcode is structured as following:

1	1	1	0	DI	AP	IK9	IK8	IK7	IK6	IK5	IK4	IK3	IK2	IK1	IK0

- DI Double instruction
  Determines the subtype of a double instruction according to the following table:

DI	Subtype
0	Merge
1	TwoDir

- AP Argument Preserve
  When set to one, the Double instruction will preserve the argument even when both instructions will modify it. It will use the argument stack for this, pushing the value at the start and popping it later. The value is always pushed when is the Double instruction entered, but when it's popped depends on the subtype of the Double instruction:
  - Merge – the value is popped after the call of the second instruction, thus even when the two instructions modify the argument, the whole instruction behaves as if they didn't.
  - TwoDir – the value is popped after the second instruction returns, thus the whole program branch behaves like it didn't modify the argument.

It is possible to isolate even a single instruction like this and put some general instruction as the second one, to prevent it from destroying the argument and still let it use the argument.

Internal Units

Important part of the 2DWPU are its internal units, that allow data storage, calculations, processing input and output, controlling program flow and various operations over data.

Registers

2DWPU contains various registers with various lengths, some with special purpose, usually linked to some other internal units, some free for programmer's use without any special meaning serving only as temporary data storage.

Work registers

For temporary data storage, programmer may use 10 registers with data width of 32bits. Values stored in these registers do not alter the behavior of the processor in any way and thus can be used for any purpose. In spirit of the WPU playfulness, the registers are named rather funnily after basic two dimensional shapes.

Indirect addressing trough HE and ST registers

Two of the work registers are however required for data transfers between the 2DWPU and program and data memory and also the IO interface. Because instructions cannot hold operands, indirect addressing is the only way to access these units, using an address stored in a specific register. One register is dedicated to addressing the program and data memory, while the other one is dedicated to addressing the IO interface. Registers are dedicated as following:

- ST – Star – Program and data memory indirect addressing
- HE – Hexagon – IO interface indirect addressing

Local registers

Because the program execution can branch when parallelism is used and the work registers are read only, as they are shared among the cores, the branched parts of the code cannot use any registers for controlling cycles or storing some results. However, two of the work registers are designated as local registers, which means that each core has its own copy of them. Also, their value is not shared, instead it's always set to zero when the core starts a new branch.

- PO – Polygon
- PE - Pentagon

Instruction address registers (xPC, yPC, BS)

To determine current position in the machine code, three registers are used, from which a single linear address is calculated as described in the Program Flow chapter. Registers xPC and yPC are both 5 bit registers, which hold a two dimensional position of the current instruction in the instruction grid. BS register is a 24 bit register that holds the starting address of the current program block.

Status Word register (SW)

This register contains several bits important to the function of the processor and execution of the program, as they provide information about some operations and status of the processor. Register size is 32bits and its split into two parts forming two subregisters. The upper half is shared among the cores, while the lower half is exclusive to each core.

Register SW is thus structures as following:

S-SW (16 bits)	P-SW (16 bits)

- **S-SW Shared Status Word**
  - Values are shared between all the registers
- **P-SW Private Status Word**
  - Values are individual to each of the cores

Private Status Word register (P-SW)

The lower part of the Status Word register is structured as following:

DS3	DS2	DS1	DS0	-	PDF	DR	PCB	AC	SC	FSB	-	-	-	SB	CB

- **Upper 8 bits**
  - Currently unused, reserved for future
- **CB carry/borrow bit**
  - This bit is set by several instructions, when carry or overflow occurs
- **SB sign bit**
  - This bit is set when the sign bit of value in register ARG is set
- **FSB Fork Secondary Branch**
  - Determines if the current execution branch is a secondary branch created by the fork instruction

- **SC Secondary Core**
    - If set, then given core is a secondary core with restricted access and abilities and any attempt to perform restricted operations will cause an interrupt
- **AC Activated Core**
    - If set, the given core is activated and it's processing code, if not, then the core is currently idle
- **PCB Program Counter Block**
    - If set, none of the Program Counter related registers will be updated when the current instruction is executed (that it, whether it calls or returns, program counter stays the same). Used internally by double instructions
- **DR Direction - Return**
    - Indicates what the last action performed was – if it was a call, then this bit is set to zero, if it was a return, then it's set to one
- **PDF Parallel Data Fetch**
    - Bit from the popped instruction flow word is stored here TODO
- **DS0…DS2 Double status**
    - These bits are internally used by the double instructions, they hold additional information about instruction query when calling an instruction fetched from the index table and then the double instruction again. They affect how the instruction flow word stack is operated

Shared Status Word register (S-SW)

This portion of the register is shared between all cores, while only the primary core can access it.

Upper 8 bits	-	-	-	-	UB3	UB2	UB1	UB0

- **UB0…UB3 User Bit**
    - Bits without any special meaning, can be freely used by programmer for whatever purpose

Stack related registers

Interrupt related registers

To control interrupt system, two registers are provided. First contains several bits that enable or disable various interrupts and change some of the interrupt settings and second register contains the base address of the block that contains interrupt vectors.

Interrupt Control – IC

This register contains bits that allow enabling or disabling interrupts and altering the interrupt behavior. Register is 32bit.

Upper 20 bits	ET3	ET2	ET1	ET0	EE	ES	EF	EA	IQ	BB	IP	IE

- **Upper 20 bits**
    - Reserved for future, currently unused
- **IE Interrupt Enable**
    - If set, interrupt system is enabled, otherwise no interrupt can occur
- **IP Interrupt protect**

  - If set, function interrupt protect is enabled. It prevents interrupt to be performed when over a three quarters of the return or argument stack are used.
- **BB** **Block Bit**
  - Read-only only bit. This bit is set when the interrupt is being processed and indicates, that other scheduled bits have to currently wait, until the current interrupt finished processing and returns
- **IQ** **Interrupts Queued**
  - If interrupt control has any interrupts scheduled this bit is set, even when interrupts are disabled by IE (however, ignores individually disabled interrupts)
- Following bits individually enable or disable interrupt from various sources.

EA	**Enable ALU**
EF	**Enable FPU**
ES	**Enable Stack**
EE	**Enable External**
ET0…ET3	**Enable Timer 0…3**

Timer related registers

Processor contains several registers that allow controlling the timers that can be used for timing various parts of program, usually in conjunction with the interrupt system. There is one control register, which allows enabling or disabling various interrupts and controlling the behavior and there are two registers for each timer, one that holds the current value (time) and second which contains prefill value that will be copied to the value register once the timer overflows.

TC	**Timer Control**
TV0…TV3	**Timer Value**
TF0…TF3	**Timer Fill**

Timer Control - TC

Using this register, programmer can enable or disable timers and control their behavior. Register is structured as following:

Upper 24 bits	TO3	TO2	TO1	TO0	TR3	TR2	TR1	TR0

- **TR0…TR3** **Timer Run**
  - If set, processor will increment value of the given timer every cycle
- **TO0…TO3** **Timer Overflow**
  - The processor automatically sets this register, when the timer overflows. This allows checking for overflow by pooling, if programmer doesn't with to use the interrupt system (or temporarily disable it)

Timer Value – TV

There are four TV registers, each one corresponding to one timer. These contain the current value of the timer and they are incremented every cycle if the given timer is enabled. All fours registers are 32 bit. Default value is 0.

Timer Fill - TF

There are four TF registers, each one corresponding to one timer. Value stored in these registers will be copied to the corresponding TV register when given timer overflows, allowing programmer to adjust the ratio of overflows without having to do manual filling in the interrupt subroutine.

Timers

2D WPU contains four timers – devices that can be automatically incremented every cycle and generate an interrupt when they overflow. This allows programmer to write timed code and various periodical checks. All four timers are 32bit and allow programmer to specify the prefill value using the TV registers, thus adjusting the interval between generated interrupts. All four timers are independent.

Interrupts

Because various events need to be processed quickly on demand, without having to constantly pool for their status. This improves the performance and makes the programming easier and cleaner. Processor is designed in a way that the interrupt subroutine doesn't affect the main execution anyhow, programmer of course needs to backup all registers he'll use, however, calling the interrupt subroutine and returning back to the main execution thread is managed by the processor. Currently only one interrupt can be processed at time and cannot be interrupted by another interrupt.

All devices that can request for interrupt are connected to the interrupt control, which processes the interrupts before sending them to the 2D WPU core. Interrupt control prepares all necessary data for the interrupt to occur, that is, the vector address (relative to the block) according to the device which requested interrupt and interrupt request bit. It also filters all the interrupts based on individual interrupt settings in the IC register and manages the interrupt queue: determines priority. If two or more devices request interrupt at the same time, it picks one of them and initiates interrupt and holds the others, after it is done processing, it initiates another interrupt.

Interrupt Controller

All devices that can initiate interrupt are connected to the Interrupt Controller, each device has its own bit for requesting an interrupt, this is called interrupt request bit (IRB). In case device needs to request an interrupt, it raises the interrupt request bit to logical one, which will indicate to the Interrupt Controller that the unit requests an immediate attention. Interrupt Controller monitors all units connected to it and also uses values from register IC to determine if given unit should be ignored or not, using the individual interrupt enable settings (see Interrupt Related Registers).

If there is at least one device requesting an interrupt, interrupt controller will set bit IQ (Interrupts Queued) and if bit IE (Interrupts Enabled), it will signal the 2D WPU core to interrupt current execution and jump to given interrupt vector.

Interrupt Sources

Several devices can request an interrupt in various exceptional events. Processor is able to differentiate between them and thus programmer doesn't have to determine which device requested an interrupt in his interrupt subroutine. Following sources of interrupt are defined and supported by the 2DWPU architecture:

- ALU
    - Raises interrupts in case of invalid mathematic or logical operations
    - Each of the invalid operations has its own interrupt vector

- Following illegal operations currently exist:
    - Zero division
    - Illegal negation
        - Occurs when instruction NEG is used on a smallest possible number, which is one smaller than the biggest possible number
- FPU
    - Interrupts can be used to catch several exceptional situations
    - Following exceptional situations exist:
        - Zero division (results in positive or negative infinity)
        - Infinity by infinity subtraction or division (results in NaN)
        - Zero by zero division (results in NaN)
        - NaN operation
            - Occurs when at least one of the operands is NaN
- Instruction and Argument Stack
    - In case one of these two stacks overflows a critical interrupt is generated
    - This usually leads to the corruption of the currently executed code and doesn't allow recovery
- Access restriction
    - These interrupts are raised by the core itself (they however go through the Interrupt Controller and can be blocked) when code within the program block violates the restrictions imposed by the program block header
    - Following interrupts can be raised:
        - Illegal program and data memory address
        - Illegal IO interface address
        - Illegal call address
        - Attempting to modify own header
- Timers
    - Timers can generate interrupt when they overflow
    - Each timer generates independent interrupt and has its own interrupt vector
- External
    - Various external devices can raise interrupt
    - There are up to 32 different external interrupts, if more are needed, they need to be grouped and a subroutine which will scan the grouped devices to determine which one requested the interrupt will be required
    - For standard external devices, interrupt vectors are predefined, the remaining ones are dependent on the specific usage of the unit
    - Following external devices are predefined
        - Switches
        - Keyboard

Interrupt Vectors

When interrupt occurs, program execution jumps to an interrupt vector that corresponds to specific device that requested interrupt. Interrupt vectors are located in a special program block, which can be located anywhere in the memory: register IB (interrupt base) is used to determine where is this block located, so 2D WPU may jump to it accordingly. Horizontal and vertical target position within the block is determined according to the following table. As interrupt subroutines size may vary with each purpose, only one cell of the program block is reserved for given interrupt vector. Programmer may place these an instruction that will jump elsewhere in the block or possibly a different program block.

EI – External Interrupt

Interrupt source	xPC	yPC
ALU – zero division	0	0
ALU – illegal negation	1	0
FPU – zero division	2	0
FPU – zero by zero	3	0
FPU – infinity -/ infinity	4	0
FPU – NaN operation	5	0
Instruction stack overflow	0	1
Argument stack overflow	1	1
Internal – Illegal program+data address	2	1
Internal – Illegal IO interface address	3	1
Internal – Illegal Call Address	4	1
Internal – Modifying own header	5	1
Timer 0	0	2
Timer 1	1	2
Timer 2	2	2
Timer 3	3	2
External - Switches	0	3
External - Keyboard	1	3
External – User 2…31	2..31	3

Memory

IO Inteface

In order to communicate with surroundings and various input, output or other types of devices that extend the functionality of the processor an IO interface is needed, which specifies the way programmer can work with the devices.

Basic Instruction Set

Instructions stored in the default program block are part of the basic instruction set. There is a limited amount of these instructions and they follow the rules described in the Basic 2D instruction chapter. Basic instructions are divided into 4 basic categories, each having several subcategories. This section defines specific instructions of each category.

Extended Instruction Set

Using the Extended Index instruction, it is possible to interpret 32bit value in the indexed table as an alternative, extended instruction. This allow processor to be extended by large number of instructions that are not used that often as instructions in the basic instruction set, so the additional space and cycles required to store and perform these are not detrimental to the program performance and size. Size of 32 bits allows a large number of instructions to be added, including instructions with a single 16bit operand. Using extended instructions is completely transparent for programmer – 2D assembler will automatically create an extended index instruction and store the appropriate extended instruction in the indexed table.

Bit Set/Clear Instructions

Composed Instruction Set

Double instructions from the Index category allow creation of a composed instruction set, because it is possible to store two 16bit instructions in the 32bit cell of the indexed data table. Additionally there are also two ways in which the instructions can be executed, plus using the argument protect bit, the group of two instructions can be "wrapped" and behave like instruction that doesn't alter the argument. All of the composed instructions are transparent to the programmer: he can use them as if they were basic instructions and the assembler will automatically create double instruction and store the appropriate instructions in the indexed data table.

MOV instructions

Basic instruction set supports MOV instructions that always move value between some location and ARG register. If programmer needs to move value between two locations, it has to be done through the ARG register. This might be sometimes a bit tedious, as programmer needs to find space for two instructions, plus the style of writing such construct is a bit more confusing. Using the composed instruction set, it is possible to combine two MOV operations into one cell and protect it using the AP bit, so the ARG register isn't modified outside the given composed instruction.

Composed MOV instruction can be written as following:

```
MOV [DESTINATION], [SOURCE]
MOV8 [DESTINAION], [SOURCE]
MOV16 [DESTINATION], [SOURCE]
```

Destination and source of standard MOV can be any of the registers accessible by MOV instructions from the basic instruction set as well as indirect addresses. Alternative variants MOV8 and MOV16 can be used only with indirect addressing.

MOV [DESINATION], [SOURCE]

Destination, Source = Any accessible register or indirect addressing.

Instruction is composed as following.

```
(AP = 1)
MOV ARG, [SOURCE]
MOV [DESTINATION], ARG
```

MOV8 [DESINATION], [SOURCE]

Destination, Source = *ST or *HE

Instruction is composed as following.

```
(AP = 1)
MOV8 ARG, [SOURCE]
MOV8 [DESTINATION], ARG
```

MOV16 [DESINATION], [SOURCE]

Destination, Source = *ST or *HE

When this variant is used with *HE (addressing IO interface), then MOV instruction is used instead of MOV16. This is because MOV and MOV16 are same for the IO interface, because data width is 16bits. Instruction is composed as following.

```
(AP = 1)
MOV16 ARG, [SOURCE]
MOV16 [DESTINATION], ARG
```

Composing directions

Standard query and operation instructions can query only up to two directions, however if programmer needs three or four directions, composed instructions are used. Composing of directions is independent on the specific query or operation instruction, what only matters are the directions.

Composed instructions can be written as following:

```
XXX_123
XXX_1234
```

- XXX – Query or Operation instruction
- ABCD – Direction 1, 2, 3 and 4

XXX_123

Instruction is composed as following:

```
XXX_12
XXX_3
```

XXX_1234

Instruction is composed as following:

```
XXX_12
XXX_34
```

PUSH and POP

Common programming practice is to backup registers by pushing them at the beginning of some subroutine and popping them at the end of it. This method possesses a risk when programmer forgets to pop some of the values or pops them in wrong order. With 2DWPU, both of these instructions can be combined into one instruction BCKP (BaCKuP) which is simply placed at the beginning of the branch of code that should be shielded from altering the registers.

This instruction is basically a TwoDir variant of the Double instruction, combining the PUSH and POP into one instruction, where the PUSH is executed when the branch is entered and POP is automatically executed when the execution flow returns from the branch. Programmer then only needs to place one instruction at the beginning of the code that he'll need to protect from altering certain registers.

BCKP register

Instruction backups one register or memory address using the indirect addressing *ST, it is composed as following:

```
PUSH register
POP register
```

PUSH address, POP address

Because basic instruction set allows to use PUSH or POP only with combination with a register value or indirect memory address, altering the ST register. By compositing instructions, it's possible to provide a solution to push or pop any arbitrary memory address without altering any register outside of the instruction.

Instruction PUSH address is composed as following:

```
(AP = 1)
MOVF ST      // backup ST

      (AP = 1)     // backup ARG again
      MOVT ST      // move the address to the ST register
      RET address  // alters ARG, but it's restored because of (AP = 1)

PUSH *ST
MOVT ST
```

Manually composing instructions

To allow maximum flexibility and usage of the architecture according to programmer's needs, 2DASM allows to manually compose any two arbitrary instructions to perform any operation he desires. The language provides a way to define a composed instruction and then use it anywhere in the code. However, if the instruction combination is to be used only once then it's also viable to compose them inline.

There are several variants of the composed instructions, based if it's merge or twodir type. Optionally, programmer can also protect the argument register, basically "wrapping" the instruction. This can either protect ARG from being altered only by the inside of the double instruction in case of merge type, or protect whole program branch in case of twodir type.

Composed instruction nesting

While one merge instruction allows to merge only two instructions into one field in the program block, storing both of them in the indexed value table, one or two instructions in this indexed data table can be merge instructions too, referencing two other groups of two instructions, thus allowing effectively to merge four instructions into one field. This method can be used to nest even more merge instructions, allowing putting a large amount of instructions into one field of the program block. Although theoretically possible, merging too many instructions is impractical and will affect the performance. This method should be used only in rare occasions, where it's beneficial to use such construct.

2D ASSEMBLER

To program the 2DWPU processor a special programming language is provided, called 2D Assembler or 2DASM for short. It provides required constructs to utilize all of the processor's features and to simplify the programming. A set of tools is provided together with the language, to compile the code and also to write the code, as it requires special formatting. Although it's possible to write 2DASM code using plaintext, it might be beneficial to use special tool which will automatically align instructions in a grid and allow marking the code, branches and so on.

EXTERNAL DEVICES

A set of external devices for input and output is provided with the processor, allowing programmers to immediately start developing and testing the architecture with having some feedback, other than manually watching specific memory locations for desired results. Additionally, a specification of the communication protocol is provided, so anyone can add his own external devices to 2DWPU and work with them in his programs. All of the standard devices are fully supported in the emulator and also implemented in hardware. Standard devices can also be removed if the programmer wishes so, as they're not crucial for 2DWPU function, however they provide an easy option for interaction.

IO Interface

The 2DWPU communicates with all the external devices via the IO Interface. Devices are mapped at various addresses, which can be accessed through indirect addressing using the HE register. Processor can then move data from or to the device. How this will be interpreted by the device is based on the exact address within the address range of the device and the specific device itself.

Communication protocol

Standard set of devices

Switches

Keyboard

LCD display

Text Display

Sound Generator

LED Indicator