DWMC-16

the instruction set
version 0.8

DIWhy 16 bit MiniComputer Project

November 22, 2023
Document Version v0.3

This document was set in LaTeX with the following typefaces:

Sans Serif: Noto Sans

Serif: Modern

Typewriter: `IBM Plex Mono`

Additional typefaces: PRicedown

Introduction

This document was written to have an extensive write down of the design for the instruction set of the DWMC-16 16 bit DIWhy minicomputer.

This is revision v0.8 of the Instruction Set.

Memory and Register Layout

This section contains a short overwiew of the DWMC-16 Memory and Register Layout.

Memory Layout

The DWMC-16 has a 16 bit wide data bus with a 24 bit address bus, the system can access up to 16MiWords (32MiByte) is Random Access Memory. It is organised into 256 segments of 64 kiWords (128 kiByte) for ease of addressing, as well as keeping code slightly shorter.

The first segment from 0x000000 to 0x00FFFF is reserved as 'fast' system and OS memory. Of this the first 16 Words are reserved for the Reset and Interrupt Vectors, which can easily be changed by the OS or external Code. The Stack also resides within this Segment, counting down from 0x00FFFF, with a maximum size if 16kiWords.

The second segment from 0x010000 to 0x01FFFF contains the address space memory mapped IO Hardware.

This is followed by six segments from 0x020000 to 0x07FFFF, containing the systems BIOS and the main OS.

The BIOS is contained within the third Memory Segment from 0x020000 to 0x02FFFF. It is made up of the Boot Loader, a simple Monitor OS and simple IO subroutines that can be used by the Main OS and every other program.

The rest of the systems memory segments is general use RAM.

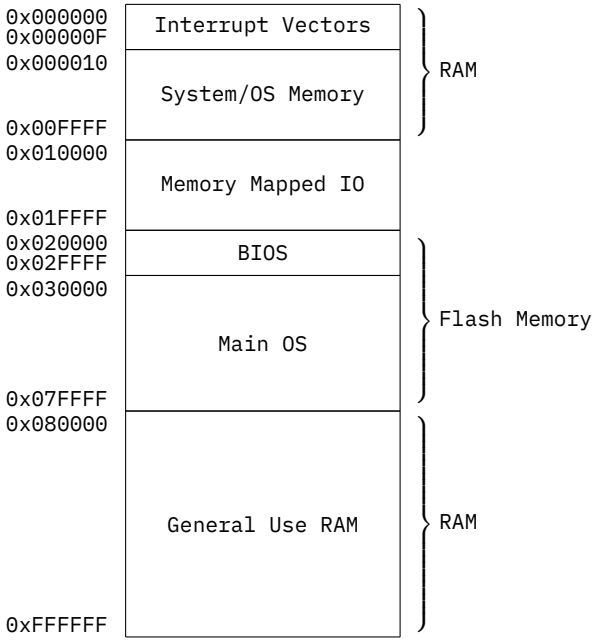


Figure 1: Memory Layout

Register Bank

The **DWMC-16** has a register bank with 16 registers of which several are meant to be special use registers.

Any registers within the register bank can be addressed and accessed like any other register. This allows to `push` and `pop` all registers (safe the Stack Pointer) to/from the stack, though the Stack Pointer can be moved if nessecary for the software run on the **DWMC-16** .

Additionally, some of the flags within the Flag Register can only be set by the hardware and can not be overwritten by overwriting the Flag Register.

R00	R01	R02/WH	R03	R04	R05	R06	R07
R08/YL	R09/YH	R10/ZL	R11/ZH	R12/SPL	R13/F	R14/PCL	R15/PCH

Table 1: Special Regsiters

The majority of the **DWMC-16** registers within the register bank are acting as special use registers.
R07 is the Flag Register, see below.
However the Y and Z Index registers are meant to be used for acting for index addressing of memory. Additionally, but the Y and Z Index registers can act as part of counting loops and can be independently tested if they are empty/zero.
The Stack Pointer register is a 16 bit wide register, as it is only used within the first memory segment, meant to be used to point towards the stack within memory. It can be automatically be incremented and decremented.
The Program Counter register is a 24 bit wide register that is needed by the control logic to run the systems program. It can automatically increment itself by one, two or three, with the latter two needed for branching purposes.

Register	**Register Label**	
R13	F	Flag Register
R08,R09	Y	Y Index Register (32 bit)
R11,R10	Z	Z Index Register (32 bit)
R12	SP	Stack Pointer Register (24 bit)
R15,R14	PC	Program Counter Register (24 bit)

Table 2: Special Use Register Labels

Flag Register

The Flag register is a special use register that is used by the Control Logic to control the program flow and contains all flags of the system. It can be operated on in normal operation like any other register, e.g. pushed and poped to/from the Stack, for subroutine operations.

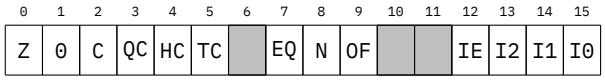


Figure 2: Flag Register Layout

Z	Zero Flag	N	Negative Flag
C	Carry Flag	OF	Overflow Flag
QC	Quarter Carry Flag	IE	Interrupt Enable Flag
HC	Half Carry Flag	I2	Interupt Flag 2
TC	Three Quater Carry Flag	I1	Interrupt Flag 1
EQ	Equal Flag	I0	Interrupt Flag 0

Table 3: Flags

Of these flags, the `IO`, `IE` and `I0-I2` Flag may be of special interest.
Internally to the **DWMC-16** CPU the `IO` is unimportant and is only of use to access external IO devices, such as serial communication ports or mass storage devices.
The `IE` flag is important for the CPUs handling of interrupts. If the `IE` flag is enabled, the **DWMC-16** can react to Interrupts, but during interrupt handling routines, the `IE` flag has to be disabled.
Finally the `I0-I2` flags, cannot be set internally by the **DWMC-16** CPU or any code itself. Instead, these flags are connected to a four layer FIFO memory, allowing the **DWMC-16** to react to interrupts coming over a short time frame in consecutive order. It also allows for Interrupt handling should a interrupt occur when another Interrupt is being handled.

OpCodes and Addressing Modes

The **DWMC-16** Instruction Set uses a 16 bit words to represent its operations. Of these 16 bit, the upper eight bit are used to identify the operation, while the lower eight bit vary in use. Some operations extend the size of the opcode by one to four byte.

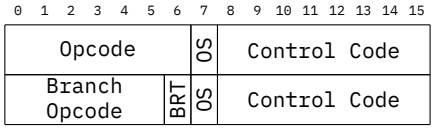


Figure 3: General Opcode Layout

OS Opcode Switch Bit

BRT Branch Target Mode Bit

The upper eight bit are made up of the 7 bit of the actual OpCode for the majority of operations, encoding the actual operation, and an OpCode Switch Bit, which has several uses in the Opcode. In case of Operations that make use of multiple Address Modes, it is used to distinguish between using two Registers and using a single Register and a memory address/constant data. In Other cases, the Opcode Switch can be used to encode two different uses of the same operation.
For Branch Operations, the first 6 bit are the actual OpCpde, while the Branch Target Mode Bit is followed by the OpCode Switch Bit. The remaining lower eight bit are again used for other Control Code.

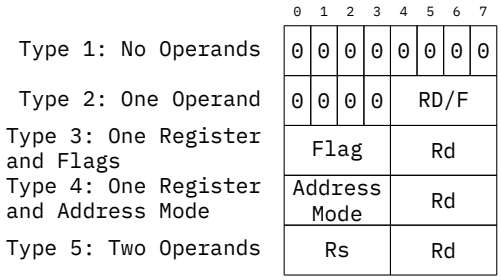


Figure 4: Control Codes

For some operations, The Destination Register Rd is replaced by the Flag/Bit selection F, we select a flag in the Flag Register or a Bit in the Bit Test register.

Addressing Mode	Example	Opcode Switch	Address Mode Code
Register	add R00, R01	0	0b0000
Immediate	ld R00, 0x0F0F	1	0b0001
Direct	ld R00, (0x0F0F)	1	0b0010
Absolute	ld R00, (0x0F0F0F)	1	0b0011
Relative	ld R00, #0x0F0F	1	0b0100
Indirect (Y)	ld R00, @Y	1	0b0101
Indirect (Z)	ld R00, @Z	1	0b1101
Indexed (Y)	ld R00, @Y0x0004	1	0b1110
Indexed (Z)	ld R00, @Z0x0004	1	0b1110

Table 4: Addressing Modes

Register Addressing Mode The Register Addressing Mode is used, when an operation is to operate over two different registers, be it arythmetically, to move the value of one Register to another, or to compare two Register values.

Immediate Mode With the Immediate Mode, the operations machine code is extended by a single 16 bit word containing a value that us directly or immediately used in the operation.

Direct In Direct Mode, the operations machine code is extended by a single 16 bit word containing the local sector address to memory. The upper 8 bit of the memory address will be taken from the Program Counter. The value at the indicated address is then used in the operation.

Absolute With Absolute Mode, the operations machine code is extended by two 16 bit words, which contain the full 24 bit address to memory. The value at the indicated address is then used in the operation.

Relative In Relative Mode, the operations machine code is extended by a single 16 bit word. The value of the word is a signed number with a memory offset to an address. This offset is added to the Program Counter and the result used to point to the address in memory. The value at the indicated address is then used in the operation. The Program Counter itself is not changed.

Indirect In Indirect Mode, the operation uses the Y or Z Index Register to point at a memory address. The value at the indicated address is then used in the operation.

Indexed In Indexed Mode, the operations machine code is extended by a single 16 bit word. The value of the word is a signed number with a memory offset to an address. This offset is added to the Y or Z Index Registers and the result used to point to the address in memory. The value at the indicated address is then used in the operation. The Index Register itself is not changed.

Branch Operations

In the case of four of the Branch Operations, the machine code of the operation can be extended by a further one or two 16 bit words, as the operation contains the jump target as either relative of absolute address. With the addition of the Address Mode to load a value to compare a Register value against, this means that the operations machine code is two to five words in length.

Branch Target Mode Bit	Branch Jump Mode	Example
0	Relative	beq Rs, Rs2, #0x0004
1	Absolute	bew Rs, Rs2, 0x0F0F0F

Table 5: Control Codes

Relative Jump The Relative Jump modifies the Program Counter by adding a signed 16 bit value.

Absolute Jump The Absolute Jump directly replaces the Program Counter with the address contained in the machine code.

Data Transfer Operations

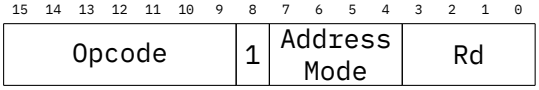
Load `ld`

Mnemonic: `ld Rd, Addr/C`

Operation: `Rd ⇐ Memory[Addr/C]`

OpCode: `0b0000001`

`0x03XX:`



Address Modes: Constant, Direct, Absolute, Relative, Indirect, Indexed

Affected Flags: `None`

The Load operation loads data from a memory address into a register.

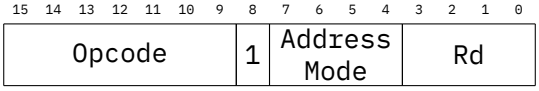
Store `st`

Mnemonic: `st Rs, Addr`

Operation: `Memory[Addr] ⇐ Rs`

OpCode: `0b0000010`

`0x05XX:`



Address Modes: Direct, Absolute, Relative, Indirect, Indexed

Affected Flags: `None`

The Store operation stores data from a register to a memory address.

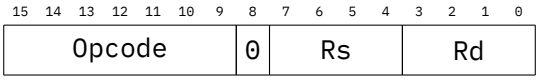
Move `mov`

Mnemonic: `mv Rd, Rs`

Operation: `Rd ⇐ Rs`

OpCode: `0b0000011`

`0x06XX:`



Address Modes: None

Affected Flags: `None`

The move operation moves data from one register to another.

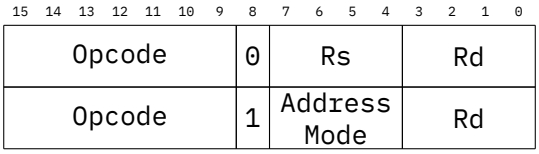
Arithmetic Logic Operations

Add `add`

Mnemonic: `add Rd, Rs`

Operation: `Rd ⇐ Rd + Rs`

OpCode: `0b0100000`



Address Modes: All

Affected Flags: `C, QC, HC, TC, Z, N, O`

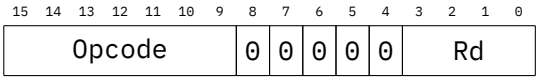
The Add Operation adds a value contained within Rs (or the appropriate addressed memory location, based on the Address Mode) to the value of Rd. Rs remains unchanged.

Increment `inc`

Mnemonic: `inc Rd`

Operation: `Rd ⇐ Rd + 1`

OpCode: `0b0100010`



Address Modes: None

Affected Flags: `C, QC, HC, TC, Z, N, O`

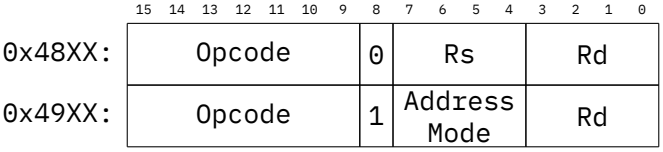
The Increment Operations increments the Register Rd by one.

Substract sub

Mnemonic: `sub Rd, Rs`

Operation: `Rd ⇐ Rd - Rs`

OpCode: `0b0100100`



Address Modes: All

Affected Flags: `C, QC, HC, TC, Z, N, O`

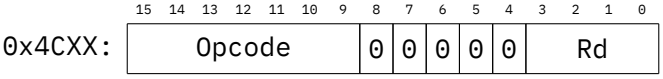
The Substract Operation substracts a value contained within Rs (or the appropriate addressed memory location, based on the Address Mode) a from the value of Rd. Rs remains unchanged.

Decrement dec

Mnemonic: `inc Rd`

Operation: `Rd ⇐ Rd - 1`

OpCode: `0b0100110`



Address Modes: None

Affected Flags: `C, QC, HC, TC, Z, N, O`

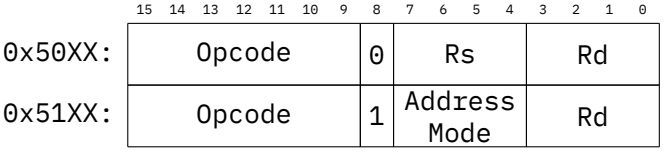
The Increment Operations decrements the Register Rd by one.

Bitwise AND and

Mnemonic: `and Rd, Rs`

Operation: `Rd ⇐ Rd and Rs`

OpCode: `0b0101000`



	15 14 13 12 11 10 9	8	7 6 5 4	3 2 1 0
0x50XX:	Opcode	0	Rs	Rd
0x51XX:	Opcode	1	Address Mode	Rd

Address Modes: All

Affected Flags: Z

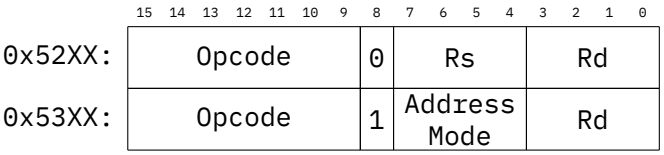
The Bitwise AND Operation combined a value contained within Rs (or the appropriate addressed memory location, based on the Address Mode) and the value of Rd bitwise with a logical AND and saves the result to Rd. Rs remains unchanged.

Bitwise OR or

Mnemonic: `or Rd, Rs`

Operation: `Rd ⇐ Rd or Rs`

OpCode: `0b0101001`



	15 14 13 12 11 10 9	8	7 6 5 4	3 2 1 0
0x52XX:	Opcode	0	Rs	Rd
0x53XX:	Opcode	1	Address Mode	Rd

Address Modes: All

Affected Flags: Z

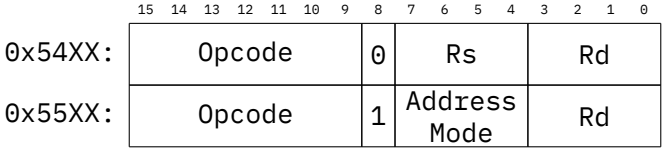
The Bitwise OR Operation combined a value contained within Rs (or the appropriate addressed memory location, based on the Address Mode) and the value of Rd bitwise with a logical OR and saves the result to Rd. Rs remains unchanged.

Bitwise XOR `xor`

Mnemonic: `xor Rd, Rs`

Operation: `Rd ⇐ Rd xor Rs`

OpCode: `0b0101010`



Address Modes: All

Affected Flags: Z

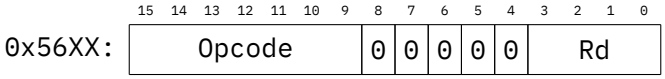
The Bitwise XOR Operation combined a value contained within Rs (or the appropriate addressed memory location, based on the Address Mode) and the value of Rd bitwise with a logical XOR and saves the result to Rd. Rs remains unchanged.

Bitwise NOT `not`

Mnemonic: `inc Rd`

Operation: `Rd ⇐ not Rd`

OpCode: `0b0101011`



Address Modes: None

Affected Flags: Z

The Bitwise NOT Operations logcially inverts the bits of Register Rd and writes the result back to Register Rd.

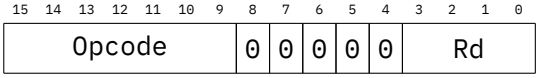
Logic Shift Left `lsl`

Mnemonic: `lsl Rd`

Operation: `Rd ⇐ Rd << 1; LSB ⇐ Carry; Carry ⇐ MSB`

OpCode: `0b0101100`

`0x58XX:`



Address Modes: None

Affected Flags: `Z, C`

The Logical Shift Left Operation logically shifts the value of register Rd one bit to the left. The Lowest Significent Bit of Rd is set to the Carry Flag, while the Carry Flag is set to the Most Siginficent Bit of the Register Rd.

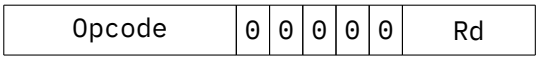
Logic Shift Right `lsr`

Mnemonic: `lsr Rd`

Operation: `Rd ⇐ Rd >> 1; MSB ⇐ Carry; Carry ⇐ LSB`

OpCode: `0b0101101`

`0x5AXX:`



Address Modes: None

Affected Flags: `Z, C`

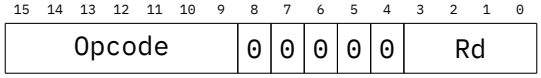
The Logical Shift Left Operation logically shifts the value of register Rd one bit to the left. The Most Significent Bit of Rd is set to the Carry Flag, while the Carry Flag is set to the Lowest Siginficent Bit of the Register Rd.

Logic Rotate Left `lrl`

Mnemonic: `lrl Rd`

Operation: `Rd ⇐ Rd << 1; LSB ⇐ MSB`

OpCode: `0b0101110`



Address Modes: None

Affected Flags: None

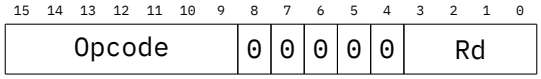
The Logical Shift Left Operation logically shifts the value of register Rd one bit to the left. The Lowest Significent Bit of Rd is set to the Most Siginficent Bit of the Register Rd.

Logic Rotate Right `lrr`

Mnemonic: `lrr Rd`

Operation: `Rd ⇐ Rd >> 1; MSB ⇐ LSB`

OpCode: `0b0101111`



Address Modes: None

Affected Flags: None

The Logical Shift Left Operation logically shifts the value of register Rd one bit to the left. The Lowest Significent Bit of Rd is set to the Most Siginficent Bit of the Register Rd.

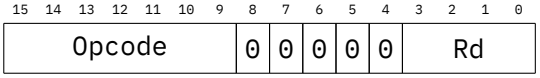
Logic Switch Bytes `lsb`

Mnemonic: `lsb Rd`

Operation: `Rd[7:0] ⇐ Rd[15:8]; Rd[15:8] ⇐ Rd[7:0]`

OpCode: `0b0110000`

`0x60XX:`



Address Modes: None

Affected Flags: None

The Logical Switch Bytes Operation logically switches the lowest eight bit and the highest eight bit of regsiter Rd around. This operation is useful for operating on external devices with an 8 bit data bus.

Control Flow Operations

Compared to the other operations, Branch operations make use of 6 bit opcodes, as they are capable of using two different jump target modes, Relative and Absolute.

Branch if Bit Set bb

Mnemonic: `bb Rs, F, Addr`

Operation: Relative Jump:
`if Rs[F] == 1: PC ⇐ PC + Addr`
`else: PC ⇐ PC + 2`
Absolute Jump:
`if Rf[F] == 1: PC ⇐ Addr`
`else: PC ⇐ PC + 3`

OpCode: `0b100010`

	15 14 13 12 11 10	9	8	7 6 5 4	3 2 1 0
Relative Jump 0x88XX:	Opcode	0	0	Flag Bit	Rd
Absolute Jump 0x8AXX:	Opcode	1	0	Flag Bit	Rd

Address Modes: None

Affected Flags: Z

This operations test whether a bit or flag are set in the Source Register Rs, which can be any register in the Register Bank. Usually this ise used to test the Flag Register F.

The operation is handled within the ALU, by use of a bit generator and an AND operation, masking the selected bit, which modifies the Z flag.

Branch if Bit Not Set bnb

Mnemonic: `bnb Rs, F, Addr`

Operation: Relative Jump:
`if RsF] == 0: PC ⇐ PC + Addr`
`else: PC ⇐ PC + 2`
Absolute Jump:
`if Rs[F] == 0: PC ⇐ Addr`
`else: PC ⇐ PC + 3`

OpCode: `0b100011`

	15 14 13 12 11 10	9	8	7 6 5 4 3 2 1 0
Relative Jump 0x8CXX:	Opcode	0	0	Flag Bit Rd
Absolute Jump 0x8EXX:	Opcode	1	0	Flag Bit Rd

Address Modes: None

Affected Flags: Z

This operations test whether a bit or flag are not set in the Source Register Rs, which can be any register in the Register Bank. Usually this ise used to test the Flag Register F.

The operation is handled within the ALU, by use of a bit generator and an AND operation, masking the selected bit, which modifies the Z flag.

Branch if Zero, Decrement `bz`

Mnemonic: `bz Rd, Addr`

Operation: Relative Jump:

```
if Rd == 0: PC ⇐ PC + Addr
else: Rd ⇐ Rd - 1;
PC ⇐ PC + 2
```

Absolute Jump:

```
if Rd == 0: PC ⇐ Addr
else: Rd ⇐ Rd - 1;
PC ⇐ PC + 3
```

OpCode: `0b100100`

	15 14 13 12 11 10	9	8	7	6	5	4	3 2 1 0
`Relative Jump 0x900X:`	Opcode	0	0	0	0	0	0	Rd
`Absolute Jump 0x920X:`	Opcode	1	0	0	0	0	0	Rd

Address Modes: None

Affected Flags: `C, Z, N`

This operations is meant to make counting loops simpler to implement.

If the given Register is Zero, the operation jumps to the given relative or absolute address.

If the given Register is not Zero, the operation decrements the Register and goes to the next operation.

Branch if Not Zero, Decrement bnz

Mnemonic: `bnz Rd, Adr`

Operation:
Relative Jump:
```
if Rd != 0: PC ⇐ PC + Adr;
Rd ⇐ Rd - 1
else: PC ⇐ PC + 2
```
Absolute Jump:
```
if Rd != 0: PC ⇐ Adr;
Rd ⇐ Rd - 1
else: PC ⇐ PC + 3
```

OpCode: `0b100101`

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Relative Jump 0x940X:	Opcode						0	0	0	0	0	0	Rd			
Absolute Jump 0x960X:	Opcode						1	0	0	0	0	0	Rd			

Address Modes: None

Affected Flags: `C, Z, N`

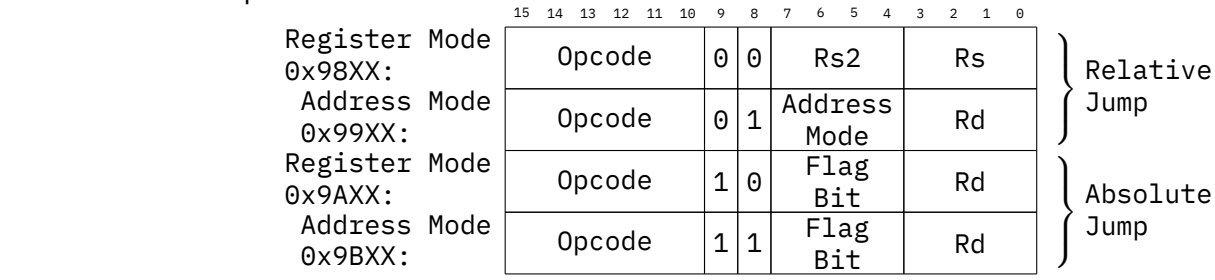
This operations is meant to make counting loops simpler to implement.
If the given Register is Not Zero, the operation jumps to the given relative or absolute address and decrements the Register.
If the given Register is Zero, the operation goes to the next operation.

Branch if equal **beq**

Mnemonic: `beq Rs, Rs2, Adr`

Operation: Relative Jump:
`if Rs == Rs2: PC ⇐ PC + Adr;`
`else: PC ⇐ PC + 2`
Absolute Jump:
`if Rs == Rs2: PC ⇐ Adr;`
`else: PC ⇐ PC + 3`

OpCode: `0b100110`



Address Modes: All

Affected Flags: None

This operation test whether two values are equal. This may either be between two registers (Register Mode) or between a Register and a place in memory (Addressing Mode).

If the two values are equal, the operations jumps to the given relative or absolute address.

If the two values are not equal, the operations goes to the next operation.

Branch if not equal **bneq**

Mnemonic: `bneq Rs, Rs2, Adr`

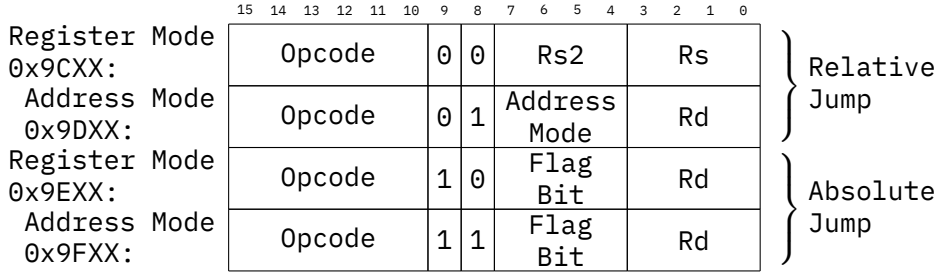
Operation: Relative Jump:

```
if Rs != Rs2: PC ⇐ PC + Adr;
else: PC ⇐ PC + 2
```

Absolute Jump:

```
if Rs != Rs2: PC ⇐ Adr;
else: PC ⇐ PC + 3
```

OpCode: `0b100111`



Address Modes: All

Affected Flags: None

This operation test whether two values are equal. This may either be between two registers (Register Mode) or between a Register and a place in memory (Addressing Mode).

If the two values are not equal, the operations jumps to the given relative or absolute address.

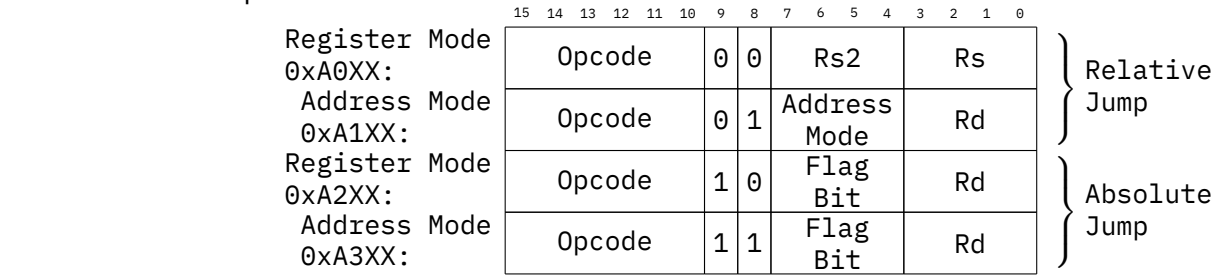
If the two values are equal, the operations goes to the next operation.

Branch if greater then `bgt`

Mnemonic: `bgt Rs, Rs2, Adr`

Operation: Relative Jump:
`if Rs > Rs2: PC ⇐ PC + Adr;`
`else: PC ⇐ PC + 2`
Absolute Jump:
`if Rs > Rs2: PC ⇐ Adr;`
`else: PC ⇐ PC + 3`

OpCode: `0b101000`



Address Modes: All

Affected Flags: None

This operation test whether values Rs is greater then value Rs2. This may either be between two registers (Register Mode) or between a Register and a place in memory (Addressing Mode).

If Rs is greater then Rs2, the operations jumps to the given relative or absolute address.

If Rs is not greater then Rs2, the operations goes to the next operation.

Branch if greater then or equal **bge**

Mnemonic: `bge Rs, Rs2, Adr`

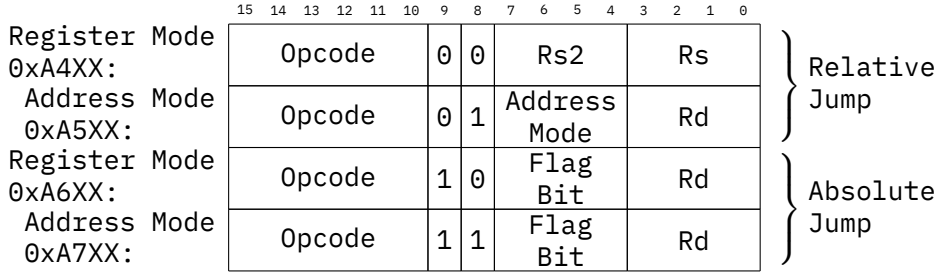
Operation: Relative Jump:

```
if Rs >= Rs2: PC ⇐ PC + Adr;
else: PC ⇐ PC + 2
```

Absolute Jump:

```
if Rs >= Rs2: PC ⇐ Adr;
else: PC ⇐ PC + 3
```

OpCode: `0b101001`



Address Modes: All

Affected Flags: None

This operation test whether values Rs is greater then or equal to value Rs2. This may either be between two registers (Register Mode) or between a Register and a place in memory (Addressing Mode).

If Rs is greater then or equal to Rs2, the operations jumps to the given relative or absolute address.

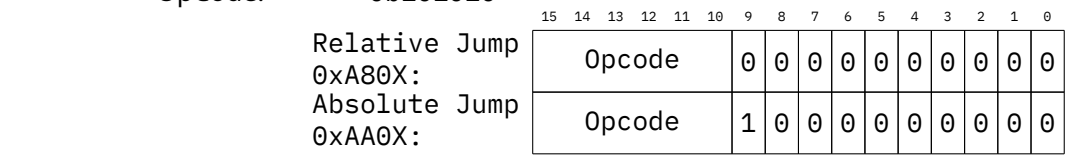
If Rs is not greater then or equal to Rs2, the operations goes to the next operation.

Jump `jmp`

Mnemonic: `jmp Adr`

Operation: Relative Jump:
`PC ⇐ PC + Adr`
Absolute Jump:
`PC ⇐ Adr`

OpCode: `0b101010`



Address Modes: None

Affected Flags: None

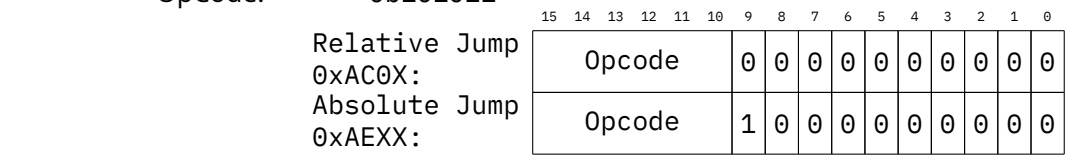
This operations is a jump to another part of the program. It either uses relative or absolute jump addressing.

Jump to Subroutine `jms`

Mnemonic: `jms Adr`

Operation: Relative Jump:
`ST ⇐ PC + 1`
`PC ⇐ PC + Adr`
Absolute Jump:
`ST ⇐ PC + 1`
`PC ⇐ Adr`

OpCode: `0b101011`



Address Modes: None

Affected Flags: None

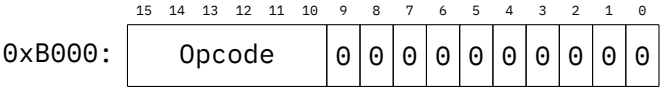
This operations is a jump into a subroutine. Before jumping into the subroutine, the operations increments the program counter by one, before pushing the program counter onto the Stack, before jumping.

Return from Subroutine `ret`

Mnemonic: `ret`

Operation: `PC ⇐ ST`

OpCode: `0b101100`



Address Modes: None

Affected Flags: None

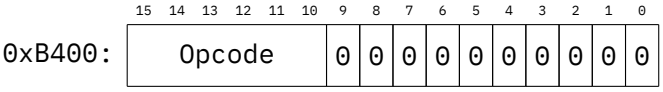
This operations is a jump returning from a subroutine. It pulls the return jump address from the Stack. This means that it does not use relative or absolute jump modes.

Return from Interrupt `reti`

Mnemonic: `reti`

Operation: `PC ⇐ ST`

OpCode: `0b101101`



Address Modes: None

Affected Flags: None

This operations is a jump returning from an interrupt.

Before returning into the normal program, it looks if another interrupt had happened during the exercution of the Interrupt handler. If another Interrupt has happened, it directly jumps into the new Interrupt handling routine.

If no other Interrupt has happened, the return operation pulls the contents of all registers, safe the Stack Pointer, from the Stack to restore the program state. It then pulls the return address from the stack and returns to the main program.

Other Operations

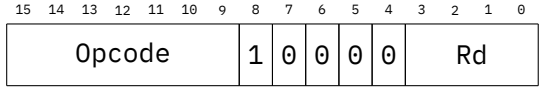
This section of the Operation Listing contains all operations that do not fall into the other categories.

Push to Stack **push**

Mnemonic: `push Rs`

Operation: `ST ⇐ Rs`

OpCode: `0b1100000`



Address Modes: None

Affected Flags: None

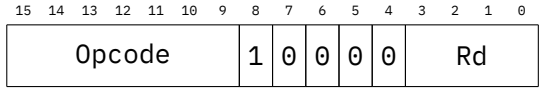
This operations pushes the content of Register Rs onto the Stack.

Pop to Stack **pop**

Mnemonic: `pop Rs`

Operation: `Rd ⇐ ST`

OpCode: `0b1100001`



Address Modes: None

Affected Flags: None

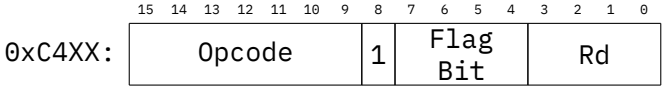
This operations pops a value from the Stack and saves it to Register Rs.

Set Bit `sb`

Mnemonic: `sb Rd, B/F`

Operation: `Rd[F] ⇐ 1`

OpCode: `0b1100010`



Address Modes: None

Affected Flags: Any

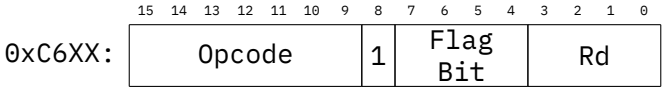
This operations sets a bit or flag, in any of the registers in the Register bank. The operation is handled within the ALU, by use of a bit generator and an OR operation, setting the selected bit.

Reset Bit `rb`

Mnemonic: `rb Rd, B/F`

Operation: `Rd[F] ⇐ 0`

OpCode: `0b1100011`



Address Modes: None

Affected Flags: Any

This operations resets a bit or flag, in any of the registers in the Register bank. The operation is handled within the ALU, by use of a bit generator (whith a negation of the bit mask) and an AND operation, resetting the selected bit.

Halt `hlt`

Mnemonic: `hlt`

Operation: `No Operation`

OpCode: `0xFEFF`

Address Modes: None

Affected Flags: `None`

This operation halts the system until a interrupt happens or the system is reset.

No Operation `nop`

Mnemonic: `nop`

Operation: `No Operation`

OpCode: `0xFFFF`

Address Modes: None

Affected Flags: `None`

This operation does nothing. It may be used for timing purposes.

List of Opcodes

Hex	Mnemonic	Hex	Mnemonic	Hex	Mnemonic	Hex	Mnemonic
Data Operations							
0x00	---	0x10	---	0x20	---	0x30	---
0x01	---	0x11	---	0x21	---	0x31	---
0x02	---	0x12	---	0x22	---	0x32	---
0x03	ld Rd, Adr/C	0x13	---	0x23	---	0x33	---
0x04	---	0x14	---	0x24	---	0x34	---
0x05	st Rs, Adr	0x15	---	0x25	---	0x35	---
0x06	mv Rd, Rs	0x16	---	0x26	---	0x36	---
0x07	---	0x17	---	0x27	---	0x37	---
0x08	---	0x18	---	0x28	---	0x38	---
0x09	---	0x19	---	0x29	---	0x39	---
0x0A	---	0x1A	---	0x2A	---	0x3A	---
0x0B	---	0x1B	---	0x2B	---	0x3B	---
0x0C	---	0x1C	---	0x2C	---	0x3C	---
0x0D	---	0x1D	---	0x2D	---	0x3D	---
0x0E	---	0x1E	---	0x2E	---	0x3E	---
0x0F	---	0x1F	---	0x2F	---	0x3F	---
ALU Operations							
0x40	add Rd, Rs	0x50	and Rd, Rs	0x60	lsb Rd	0x70	---
0x41	add Rd, Adr	0x51	and Rd, Adr	0x61	---	0x71	---
0x42	---	0x52	or Rd, Rs	0x62	---	0x72	---
0x43	---	0x53	or Rd, Adr	0x63	---	0x73	---
0x44	inc Rd	0x54	xor Rd, Rs	0x64	---	0x74	---
0x45	---	0x55	xor Rd, Adr	0x65	---	0x75	---
0x46	---	0x56	not Rd	0x66	---	0x76	---
0x47	---	0x57	---	0x67	---	0x77	---
0x48	sub Rd, Rs	0x58	lsl Rd	0x68	---	0x78	---
0x49	sub Rd, Adr	0x59	---	0x69	---	0x79	---
0x4A	--	0x5A	lsr Rd	0x6A	---	0x7A	---
0x4B	---	0x5B	---	0x6B	---	0x7B	---
0x4C	dec Rd	0x5C	lrl Rd	0x6C	---	0x7C	---
0x4D	---	0x5D	---	0x6D	---	0x7D	---
0x4E	---	0x5E	lrl Rd	0x6E	---	0x7E	---
0x4F	---	0x5F	---	0x6F	---	0x7F	---
Branch Operations							
0x80	---	0x90	bz Rd, Off	0xA0	bgt Rs, Rs2, Off	0xB0	ret
0x81	---	0x91	---	0xA1	bgt Rs, Adr, Off	0xB1	---
0x82	---	0x92	bz Rd, Adr	0xA2	bgt Rs, Rs2, Adr	0xB2	---
0x83	---	0x93	---	0xA3	bgt Rs, Adr, Adr	0xB3	---
0x84	---	0x94	bnz Rd, Off	0xA4	bge Rs, Rs2, Off	0xB4	reti
0x85	---	0x95	---	0xA5	bge Rs, Adr, Off	0xB5	---
0x86	---	0x96	bnz Rd, Adr	0xA6	bgt Rs, Rs2, Adr	0xB6	---
0x87	---	0x97	---	0xA7	bgt Rs, Adr, Adr	0xB7	---
0x88	bb Rs, F, Off	0x98	beq Rs, Rs2, Off	0xA8	jmp Off	0xB8	---
0x89	---	0x99	beq Rs, Adr, Off	0xA9	---	0xB9	---
0x8A	bb Rd, F, Adr	0x9A	beq Rs, Rs2, Adr	0xAA	jmp Adr	0xBA	---
0x8B	---	0x9B	bew Rs, Adr, Adr	0xAB	---	0xBB	---
0x8C	bnb Rs, F, Off	0x9C	bneq Rs, Rs2, Off	0xAC	jms Off	0xBC	---
0x8D	---	0x9D	bneq Rs, Adr, Off	0xAD	---	0xBD	---
0x8E	bnb Rs, f, Adr	0x9E	bneq Rs, Rs2, Adr	0xAE	jmp Adr	0xBE	---
0x8F	---	0x9F	bneq Rs, Adr, Adr	0xAF	---	0xBF	---
Other Operations							
0xC0	push Rs	0xD0	---	0xE0	---	0xF0	---
0xC1	---	0xD1	---	0xE1	---	0xF1	---
0xC2	pop Rd	0xD2	---	0xE2	---	0xF2	---
0xC3	---	0xD3	---	0xE3	---	0xF3	---
0xC4	sb Rd, B/F	0xD4	---	0xE4	---	0xF4	---
0xC5	---	0xD5	---	0xE5	---	0xF5	---
0xC6	rb Rd, B/F	0xD6	---	0xE6	---	0xF6	---
0xC7	---	0xD7	---	0xE7	---	0xF7	---
0xC8	---	0xD8	---	0xE8	---	0xF8	---
0xC9	---	0xD9	---	0xE9	---	0xF9	---
0xCA	---	0xDA	---	0xEA	---	0xFA	---
0xCB	---	0xDB	---	0xEB	---	0xFB	---
0xCC	---	0xDC	---	0xEC	---	0xFC	---
0xCD	---	0xDD	---	0xED	---	0xFD	---
0xCE	---	0xDE	---	0xEE	---	0xFE	hlt
0xCF	---	0xDF	---	0xEF	---	0xFF	nop

Appendix

Rd	Destination Register
Rs, Rs2	Source Register
PC	Program Counter
C	Constant Number
B	Branch Jump Mode
Adr	Memory Address (16/24 bit)
Off	Address Offset (16 bit)
ST	Stack Pointer
F	Flag Bit
MSB	Most Significant Bit
LSB	Least Significant Bit

Table 6: Legend of Abbreviations