16-Bit CPU/MCU

Philo Kaulkin

Designed / Simulated in Hneeman's digital.

Features

- **16, 16-bit general purpose registers**
- **Signed and Unsigned Integer division**
- **48 digital input pins, 48 digital output pins**
- **128kib data memory, 128kib program memory (separate)**
- **40+ Unique instructions, all single cycle execution**

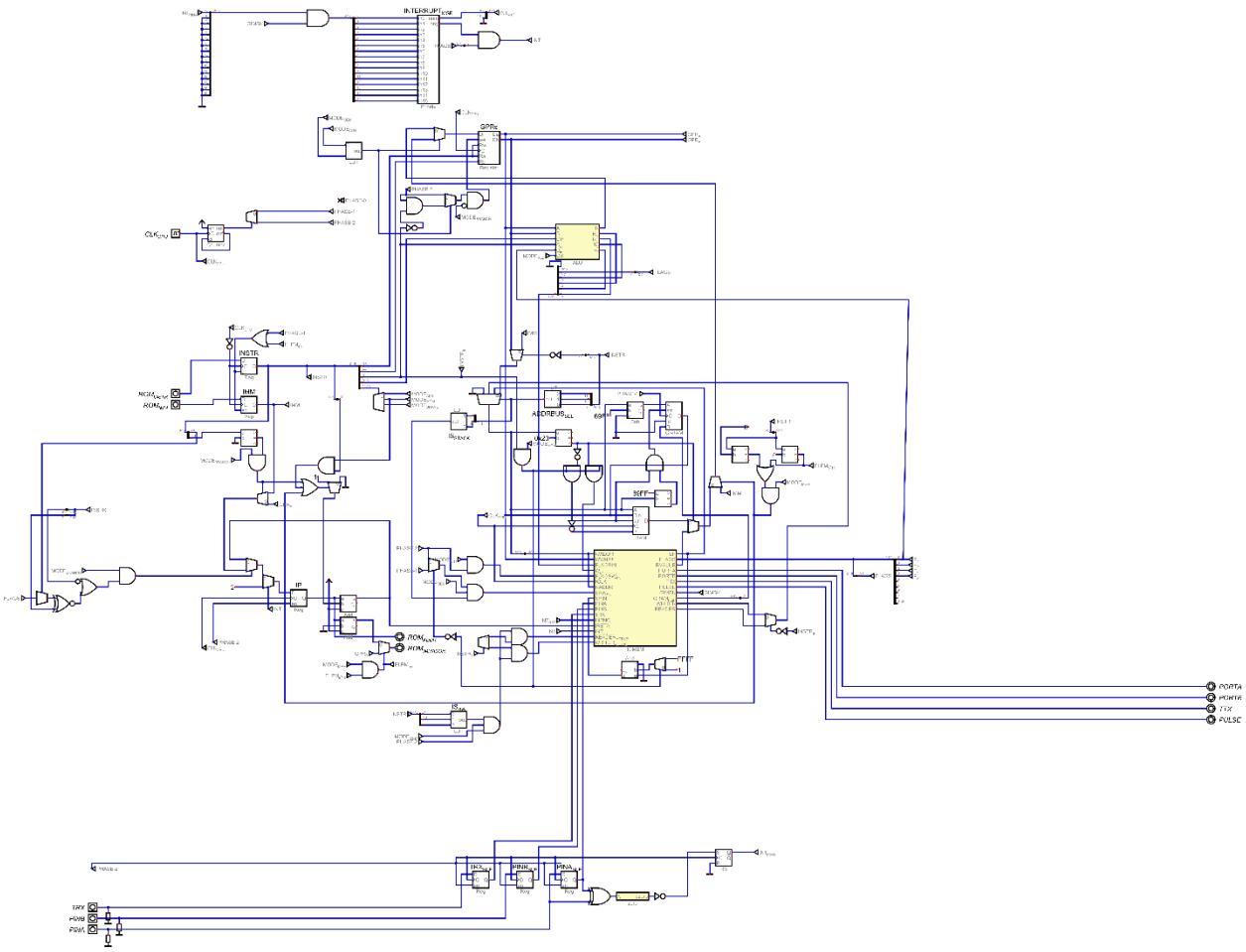


Table of Contents

Registers

General Purpose Registers

All registers will be 16-bit words, with 16-bit address locations. This means there is no way to directly point to the upper half of a register because all values are 16 bits.

The CPU-16 architecture contains 16 general purpose registers with mnemonics **r0-r15**. No special purpose is applied to these, except that **r15** is used as a scratch register in certain compound instructions in the assembler. This means that during some instructions the value of **r15** may be clobbered in what appears to be a single instruction. For this reason, it is highly discouraged to use r15 whenever possible.

These 16 general purpose registers are indexed in the intuitive order using a 4-bit value for register index in instructions. These bits will always be found in the first or second nibble of the instruction (0...3, 4...7). Any instruction that writes to a register will denote that register in the first 4 bits, labeled with letter *d.* Instructions that write to a register, and read from another will read from the register in the second 4 bits, labelled with letter *r.*

Special Purpose Registers

The first 32 bytes of memory are mapped I/O for specific special purpose registers. These are as follows, with their associated memory address:

Address	Register Name	Mutability	Description
0x00	*RESERVED*	Cannot be written, will always read 0x00	*RESERVED*
0x01	*SP*	R/W Normal + Special Instructions	Stack Pointer
0x02	*FLAGS*	R/W Normal + ALU Flag modifies	ALU and other flags
0x03	*PORTA*	R/W Normal	General Purpose Output
0x04	*PORTB*	R/W Normal	General Purpose Output
0x05	*PINA*	Read-only	General Purpose Input
0x06	*PINB*	Read-only	General Purpose Input
0x07	*TTX*		
0x08	*TRX*		
0x09	*PULSE*	Write-only	Apply a 1 cycle pulse on an output pin
0x0A	*RAND*	Read-only	Will contain a new random number every cycle
0x0B	*GRAM_FLIP*	Write-only	Flip graphics ram by writing

0x0C	*WRITER*	R/W Normal + Special Instruction	Specialized fast writing register
0x0D	*READER*	R/W Normal + Special Instruction	Specialized fast reading register
0x0E … 0x1A	*RESERVED*	*RESERVED*	*RESERVED*
0x1B	*PCAMSK*	*TODO*	*TODO*
0x1C	*PCBMSK*	*TODO*	*TODO*
0x1D	*GIMSK*	*TODO*	*TODO*
0x1E	*IRETA*	*TODO*	*TODO*
0x1F	*INTNO*	*TODO*	*TODO*

Additional Notes

0x01 – SP

This is the stack pointer used by instructions such as push, or pop. This can be read or written to by normal memory operations as well.

0x02 – FLAGS

This is the flags register used by the ALU and for interrupt enable/disable. Each bit is assigned a mnemonic and purpose as follows:

Bit Number	Mnemonic	Description
0	C	Carry ALU Flag
1	Z	Zero ALU Flag
2	N	Negative ALU Flag
3	V	Overflow ALU Flag
4..6	*RESERVED*	*RESERVED*
7	I	Interrupt Enable

0x09 – PULSE

The PULSE register is used to create a 1-cycle pulse on an output pin. Each of the 16 bits of PULSE corresponds to an output pin. By writing a 1 to any bit of PULSE, the corresponding output pin will go high for 1 cycle and reset to 0 on the next. The PULSE register itself will also reset back to 0x0000 every cycle, so reads will always result in 0x0000.

0x0A – RAND

The RAND register will generate a new random number every cycle. Although it can be written to, this will have no effect because any value written to RAND will be overwritten on the next cycle.

0x0B – GRAM_FLIP

The GRAM_FLIP register is used to flip the double buffer used on the graphics simulation. Set GRAM_FLIP to either 1, or 0 to select which buffer is displayed.

0x0C – WRITER

The WRITER register is used to rapidly write data to continuous memory, similar to READER. WRITER points to a location in memory and allows a write to this location followed by an increment to the pointer in a single instruction. For example, to write 3 values to 3 memory locations, assign the pointer of the first location to WRITER, and use the write instruction three times. This is faster than other methods because it encapsulates 3 memory writes and 3 increments (normally 6 cycles) into just 3 instructions (now 3 cycles).

0x0D – READER

The READER register works similarly to the WRITER register to rapidly read values from continuous memory. READER points to a location in memory and allows a read from this location followed by an increment to the pointer in a single instruction. For example, to read 3 values from 3 memory locations, assign the pointer of the first location to READER, and use the read instruction three times. This is faster than other methods because it encapsulates 3 memory reads and 3 increments (normally 6 cycles) into just 3 instructions (now 3 cycles).

Memory

Overview

Memory on CPU-16 is split into two different address spaces following a Harvard Architecture. Each address space has 16-bit addressable units with 16-bit addresses. This is a total of 262,144 bytes of memory with half being for programs and the other for a mix of purposes.

Program Memory

Program memory stores the instructions, and constants used for a given program. This memory is *not* writable once it is flashed; however, it can be read using the *ELPM* instruction. Although the program memory address space begins at 0x0000, programs written using the provided assembler will start at significantly higher addresses due to the boot code inserted at the start of the program.

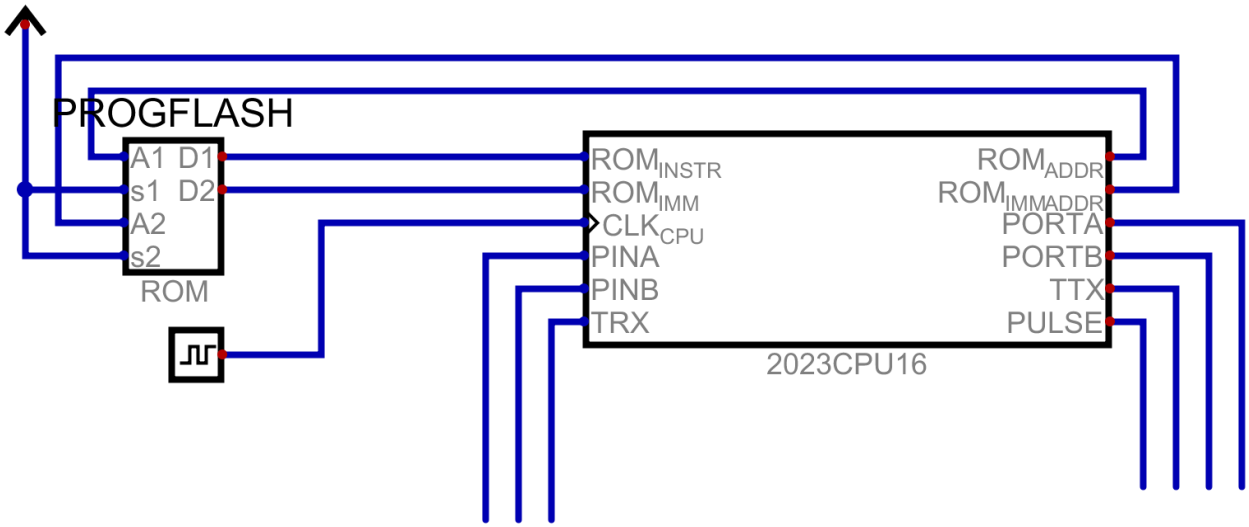
Data Memory

Data memory is mapped into several sections with different purposes. Data memory is written and read using the following instructions: *LD, ST, PUSH, POP, CALL, RET, READ, WRITE.* The different sections of data memory are detailed in the following table:

Address Range	**Name**	**Description**
0x00 – 0x1F	IOSRAM	This is the mapping for special purpose registers (see Special Purpose Registers for details)
0x20 – 0x69FF	General Purpose Memory	This can be used by the program for anything
0x69FF	Stack Start	The stack will grow from this point downwards into general purpose memory
0x7000 – 0xFFFF	Graphics RAM	Graphical memory used to drive the display

Pinout

Minimal Viable Configuration Example



The above configuration includes program memory, an external clock, and the MCU itself. This is the minimum viable configuration for expected behavior. Although anything may be used for the clock, like a push button or another oscillator, the clock source works best with this simulation tool. The program ROM is given two addresses as outputs from the MCU. Each of the corresponding values at these addresses are fed into the ROM_{INSTR} and ROM_{IMM} inputs to the MCU. These provide instructions and immediate values from ROM to the internals of the MCU. The other pins may be used for any additional peripherals desired. See Special Purpose Registers for more information on how to use these pins.

Instructions

Overview

The following table summarizes the instruction set of CPU-16:

Mnemonic	Operands	Cycles	Description
ADC	d, r	1	Add with carry
ADD	d, r	1	Add
AND	d, r	1	Bitwise and
CALL	r | pa	3	Call subroutine
CMP	d, r	1	Compare
DEC	d	1	Decrement
DIV	d, r	1	Unsigned integer divide
DIVREM	d, r	1	Unsigned integer division remainder
ELPM	d, r	1	Load program memory
HLT		INF	Halt
IDIV	d, r	1	Signed integer divide
IDIVREM	d, r	1	Signed integer division remainder
IMUL	d, r	1	Signed integer multiply
INC	d	1	Increment
JCC	r | pa	1	Conditional Jump
JMP	r | pa	1	Unconditional Jump
LD	d <- r | sa	1	Memory load
LDI	d, im	1	Load immediate
LEA	d, r * im	1	Load effective address
LSH	d, r	1	Left bitwise shift
MOV	d, r	1	Move register
MUL	d, r	1	Unsigned integer multiply
NOP		1	No operation
NOT	d	1	Bitwise not
OR	d, r	1	Bitwise or
POP	d	1	Pop from stack
PUSH	r	1	Push to stack
READ	d	1	Read using READER register
RET		2	Return from subroutine
ROL	d, r	1	Rotate left
ROR	d, r	1	Rotate right
RSH	d, r	1	Right bitwise shift
SBC	d, r	1	Subtract with carry
ST	d -> r | sa	1	Memory store
SUB	d, r	1	Subtract
WRITE	r	1	Write using WRITER register
XOR	d, r	1	Bitwise exclusive or
ZERO	d	1	Set value of register to 0x0000

Instruction Phases

Each cycle has 3 distinct phases that occur internally. The most notable of which stores the result of ALU operations as flags in the *FLAGS* special purpose register. The functions and states associated with each instruction phase are outlined in the following table:

Phase Number	Phase Name	Phase Properties
0	Setup	During the Setup phase, relevant values from GPR's (General Purpose Registers) are available on source and destination buses, and results and flags are available on ALU output. Memory values for memory reads will be available on the memory output bus.
1	Commit	The GPR file is enabled for writing (on relevant instructions), the relevant section of data memory is enabled for writing (on relevant instructions)
2	Fetch	The internal instruction and immediate buffers are written using new instructions from ROM

Instruction Modes

There are four different 'modes' of instructions, which are specified in the last 2 bits of the opcode. These are ALU, Memory, and Branch (the fourth is reserved and not yet implemented). These modes enable and disable different parts of the processor during the execution of each instruction. The specifics are detailed in the following table:

Mode	Properties
ALU	Data Memory is inactive, Immediate register is unused, *FLAGS* register is always written, ALU is enabled, destination register is always written with ALU result (except for *CMP*).
Memory	ALU is inactive, Data memory is active, immediate may be used as address or loaded directly to general purpose register (*LDI*), stack pointer may be modified.
Branch Mode	Data Memory is inactive, GPR file is never written, immediate may be used as immediate program address, FLAGS are never modified.

Values in FLAGS

The *FLAGS* special purpose register is updated on ALU instructions with certain properties of the result. The actual expressions used to determine each bit of *FLAGS* is detailed in the following table:

(R refers to the ALU result, r refers to the source register, d refers to the destination register, the integers following a register indicate a specific bit of that register)

Bit Number	Mnemonic	Expression	Description
0	C	$C \leftarrow r15 * d15 + r15 * \overline{R15} + d15 * \overline{R15}$	Carry
1	Z	$Z \leftarrow R == 0$	Result was 0x0000
2	N	$N \leftarrow R15$	Negative Result
3	V	$V \leftarrow r15 * d15 * \overline{R15} + \overline{r15} * \overline{d15} * R15$	2's Compliment Overflow
4	*RESERVED*	--	
5	*RESERVED*	--	
6	*RESERVED*	--	
7	I	Not Modified Implicitly By ALU	Interrupt Enable

ADC – Add with Carry

Description

Add register r to register d using the carry flag from FLAGS, storing the result in the destination register (d).

Operation:

$$d \leftarrow (r + d) + c$$

Syntax:

```
adc d, r
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	R	R	R	R
1	1	1	0	0	0	0	0	0

Cycles: 1

Words: 1

ADD – Add without Carry

Description

Add register r to register d, storing the result in the destination register (d). This will update the FLAGS register with new ALU flags according to the result.

Operation:

$$d \leftarrow (r + d)$$

Syntax:

```
add d, r
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	R	R	R	R
1	0	1	0	0	0	0	0	0

Cycles: 1

Words: 1

AND – Bitwise AND

Description

Bitwise AND register r with register d, storing the result in the destination register (d). This will update the FLAGS register with new ALU flags according to the result.

Operation:

$$d \leftarrow (r \ \& \ d)$$

Syntax:

```
and d, r
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	R	R	R	R
1	0	1	0	0	1	0	0	0

Cycles: 1

Words: 1

CALL – Call Subroutine

Description

Call a subroutine using the SP stack to store a return address. CALL can accept either a register containing the program address of the subroutine, or an immediate program address. This will first push the address of the next instruction to the stack, then jump to the specified program address. The RET instruction can then be used to return from the subroutine.

Operation (Immediate Program Address):

$$[SP] \leftarrow PC + 5$$
$$SP \leftarrow SP - 1$$
$$PC \leftarrow IPA$$

Operation (Register Program Address):

$$[SP] \leftarrow PC + 5$$
$$SP \leftarrow SP - 1$$
$$PC \leftarrow r$$

Syntax:

```
call r
call my_label_name
```

Opcode:

This instruction is a composite of *LDI, PUSH, and JMP.* See the architecture assembly definition (arch.asm) for more details.

Cycles: 3

Words: 3 (register program address) or 4 (immediate program address)

CMP – Compare Values

Description

Perform a comparison between the values in two registers. This is essentially equivalent to a *SUB* instruction; however, the result is never committed to the destination register. Values in *FLAGS* will reflect the subtraction operation. See Values in FLAGS for details.

Operation:

$$FLAGS \leftarrow flags(d - r)$$

Syntax:

```
cmp d, r
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	R	R	R	R
1	0	1	1	1	1	0	0	0

Cycles: 1

Words: 1

DEC – Decrement

Description

Subtract 1 from the destination register, update flags.

Operation:

$$d \leftarrow (d - 1)$$

Syntax:

```
dec  d
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	0	0	0	0
1	1	0	1	1	1	0	0	0

Cycles: 1

Words: 1

DIV – Unsigned Integer Divide

Description

Unsigned integer division of destination register by source register. For finding the remainder of division operations, see *DIVREM.*

Operation:

$$d \leftarrow (d/r)$$

Syntax:

```
div d, r
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	R	R	R	R
1	0	1	0	1	1	0	0	0

Cycles: 1

Words: 1

DIVREM – Unsigned Integer Division Remainder

Description

Unsigned integer division remainder of destination register by source register. For the result of division, instead of remainder, see *DIV*. This instruction sets *FLAGS* based on the remainder result, not the result of the division. See Values in FLAGS for details.

Operation:

$$d \leftarrow (d \,\%\, r)$$

Syntax:

```
divrem d, r
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	R	R	R	R
1	0	1	0	1	1	1	0	0

Cycles: 1

Words: 1

ELPM – Load Program Memory

Description

Load a value from program memory, into a destination register. This instruction is the only way to transfer data from program memory into data memory.

Operation:

$$d \leftarrow program[\, r \,]$$

Syntax:

```
elpm d, r
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	R	R	R	R
1	0	1	1	1	1	0	1	0

Cycles: 1

Words: 2

HLT – Halt Program

Description

Halt the program by continuously jumping to the same location in program memory.

Operation:

$$PC \leftarrow PC - 1$$

Syntax:

```
hlt
```

Opcode:

Byte	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	PC	PC	PC	PC	PC	PC	PC	PC
3	PC	PC	PC	PC	PC	PC	PC	PC

Cycles: inf

Words: 2

IDIV – Signed Integer Divide

Description

Signed integer division of destination register by source register. For the signed division remainder, see *IDIVREM.*

Operation:

$$d \leftarrow (d \,/\, r)$$

Syntax:

```
idiv d, r
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	R	R	R	R
1	1	1	0	1	1	0	0	0

Cycles: 1

Words: 1

IDIVREM – Signed Integer Division Remainder

Description

Signed integer division remainder of destination register by source register. For the signed division result, see *IDIV.* This instruction sets *FLAGS* based on the remainder result, not the result of the division. See Values in FLAGS for details.

Operation:

$$d \leftarrow (d \,\%\, r)$$

Syntax:

```
idivrem d, r
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	R	R	R	R
1	1	1	0	1	1	1	0	0

Cycles: 1

Words: 1

IMUL – Signed Integer Multiply

Description

Signed multiplication of destination register by source register, stored in the destination register.

Operation:

$$d \leftarrow (d * r)$$

Syntax:

```
imul d, r
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	R	R	R	R
1	1	1	1	0	0	0	0	0

Cycles: 1

Words: 1

INC – Increment

Description

Add 1 to the destination register, update flags.

Operation:

$$d \leftarrow (d + 1)$$

Syntax:

```
inc d
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	0	0	0	0
1	0	0	1	1	1	0	0	0

Cycles: 1

Words: 1

JCC – Conditional Jump

Description

Conditionally jump to register or immediate program address. Conditions will check for a specific value of a specific bit of the *FLAGS* register. See Values in FLAGS for details. JCC has multiple forms depending on which flag is being tested:

Mnemonic	Flag To Test	Value of Flag to Pass	Usage
J c	Carry	1	Check for Carry of last Op
J z	Zero	1	Check if last op resulted in 0x0000
J e	Zero	1	Check if source and destination are equal in cmp instruction (equivalent to *J z*)
J n	Negative	1	Check if last operation resulted in negative number (this will consider all results as signed 2's compliment integers, regardless of the signedness of the operation).
J v	Overflow	1	Check if last operation resulted in overflow
J i	Interrupt Enable	1	Check if interrupts are enabled
J nc	Carry	0	Check if last op had no carry
J nz	Zero	0	Check if last op resulted in something other than 0x0000
J ne	Zero	0	Check if source and destination are not equal in cmp instruction (equivalent to *J nz*)
J nn	Negative	0	Check if last operation resulted in positive number (this will consider all results as signed 2's compliment integers, regardless of the signedness of the operation).
J nv	Overflow	0	Check if the last op did not overflow
J ni	Interrupt Enable	0	Check if interrupts are disabled

Operation:

Where c is the given condition, and b is the value of the flag needed to pass.

$$IF\ c == b$$
$$\quad PC \leftarrow destination$$
$$ELSE$$
$$\quad PC \leftarrow PC$$

Syntax:

```
j c my_label
J c r
```

Opcode (Register):

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	d	d	d	d	c	c	c	c
1	b	1	0	0	0	0	0	1

Opcode (Immediate Program Address)

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	0	0	0	C	C	C	c
1	b	1	0	0	0	0	0	1
2	IPA	IPA	IPA	IPA	IPA	IPA	IPA	IPA
3	IPA	IPA	IPA	IPA	IPA	IPA	IPA	IPA

Cycles: 1

Words: 1 or 2

JMP – Unconditional Jump

Description

Unconditionally jump to program address (either register or immediate program address).

Operation:

$$PC \leftarrow dest$$

Syntax:

```
jmp my_label
jmp r
```

Opcode (Immediate Program Address):

Byte	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	IPA	IPA	IPA	IPA	IPA	IPA	IPA	IPA
3	IPA	IPA	IPA	IPA	IPA	IPA	IPA	IPA

Opcode (Register):

Byte	0	1	2	3	4	5	6	7
0	d	d	d	d	0	0	0	0
1	0	1	0	0	0	0	0	1

Cycles: 1

Words: 1 or 2

LD – Memory Load

Description

Load a value from data memory into a destination register.

Operation:

Where address is either an immediate data address, or the value of a specified general purpose register

$$d \leftarrow [address]$$

Syntax:

```
ld d <- [0x1234]
ld d <- r
```

Opcode (Register):

Byte	0	1	2	3	4	5	6	7
0	D	D	D	D	R	R	R	R
1	0	0	0	0	0	0	1	0

Opcode (Immediate Data Address)

Byte	0	1	2	3	4	5	6	7
0	D	D	D	D	0	0	0	0
1	0	0	0	0	1	0	1	0
2	IDA	IDA	IDA	IDA	IDA	IDA	IDA	IDA
3	IDA	IDA	IDA	IDA	IDA	IDA	IDA	IDA

Cycles: 1

Words: 1 or 2

LDI – Load Immediate

Description

Load immediate value to general purpose register.

Operation:

$$d \leftarrow imm$$

Syntax:

```
ldi r, 0x1234
```

Opcode:

Byte	0	1	2	3	4	5	6	7
0	D	D	D	D	0	0	0	0
1	0	0	1	1	1	0	1	0
2	IMM	IMM	IMM	IMM	IMM	IMM	IMM	IMM
3	IMM	IMM	IMM	IMM	IMM	IMM	IMM	IMM

Cycles: 1

Words: 2

LEA – Load Effective Address

Description

Add shifted source register to destination register. *LEA* (Load Effective Address), inspired by amd64's similar instruction, combines addition and multiplication by a power of 2 into a single instruction. This is especially useful for computing memory addresses as offsets of a base pointer with elements of size 2, 4, or 8 words. Due to this instruction being an ALU operation, the value of *FLAGS* is set based on the result. See Values in FLAGS for details.

Operation:

Where k is the constant power of 2 provided in the instruction (2,4, or 8) …

$d \leftarrow d + k * r$

Syntax:

```
lea d, [d + r*2]
lea d, [d + r*4]
lea d, [d + r*8]
```

Opcode:

Bits of 'K' will be associated with each power of 2 used in this instruction, detailed in the following table:

Power of 2	K-Value
2	0xA
4	0xB
8	0xC

Byte	0	1	2	3	4	5	6	7
0	D	D	D	D	R	R	R	R
1	0	K	K	K	K	0	0	0

Cycles: 1

Words: 1

LSH – Left Bitwise Shift

Description

Left logical bitwise shift destination register by source register lowest 4 bits. Values greater than 0x10 in the source register will overflow the lowest 4 bits of the source register. For example, source register values of 0x1, 011, and 0x51 will all result in a left shift of 1 place.

Operation:

$$d \leftarrow (d \ll r)$$

Syntax:

```
lsh d, r
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	R	R	R	R
1	0	0	0	1	0	0	0	0

Cycles: 1

Words: 1

MOV – Move Register

Description

Move source register into destination register. By moving the same register into itself, a NOP instruction is essentially made. However, because this instruction is an ALU mode instruction, the values of *FLAGS* are updated. See Values in FLAGS for details.

Operation:

$$d \leftarrow r$$

Syntax:

```
mov d, r
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	R	R	R	R
1	0	0	0	0	0	0	0	0

Cycles: 1

Words: 1

MUL – Unsigned Integer Multiply

Description

Move source register into destination register. By moving the same register into itself, a NOP instruction is essentially made. However, due to this instruction being an ALU mode instruction, the *FLAGS* register will be updated based on the value of r.

Operation:

$$d \leftarrow r$$

Syntax:

```
mov d, r
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	R	R	R	R
1	0	0	0	0	0	0	0	0

Cycles: 1

Words: 1

NOP – No Operation

Description

Do nothing, simply increment the program counter and move to the next instruction.

Operation:

Syntax:

```
nop
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	1

Cycles: 1

Words: 1

NOT – Bitwise NOT

Description

Invert the value in the destination register. This instruction applies a bitwise not operation to the value in destination, overwriting the old value.

Operation:

$$d \leftarrow \bar{d}$$

Syntax:

```
not d
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	0	0	0	0
1	0	0	1	1	0	0	0	0

Cycles: 1

Words: 1

OR – Bitwise OR

Description

Perform bitwise OR between destination and source registers, storing the result in the destination register.

Operation:

$$d \leftarrow d \mid r$$

Syntax:

```
or d, r
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	R	R	R	R
1	0	1	1	1	0	0	0	0

Cycles: 1

Words: 1

POP – Pop from Stack

Description

Perform a pop operation from the stack into the destination register, using the *SP* special purpose register. This stack operation uses pre-incrementation on the special purpose *SP* register before accessing the stack. It is also important to note that the functionality of stack operations is agnostic to the value of *SP*, and will write and read values from any part of data memory that is pointed to by *SP*. See Data Memory for more details.

Operation:

$$SP \leftarrow SP + 1$$
$$d \leftarrow [SP]$$

Syntax:

```
pop  d
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	0	0	0	0
1	0	0	0	1	0	0	1	0

Cycles: 1

Words: 1

PUSH – Push to Stack

Description

Perform a push operation from the source register to the stack, using the *SP* special purpose register. This stack operation decrements the special purpose *SP* register after accessing the stack. It is also important to note that the functionality of stack operations is agnostic to the value of *SP*, and will write and read values from any part of data memory that is pointed to by *SP*. See Data Memory for more details.

Operation:

$$d \leftarrow [SP]$$

$$SP \leftarrow SP - 1$$

Syntax:

```
push r
```

Opcode:

Byte	0	1	2	3	4	5	6	7
0	D	D	D	D	0	0	0	0
1	0	0	0	1	0	0	1	0

Cycles: 1

Words: 1

READ – Read using READER Register

Description

Read a value from data memory using the *READER* special purpose register address, and post-increment the *READER* register. This is especially useful for reducing the number of cycles required to read sequential strings of memory. To use this instruction, store a pointer to the beginning of a piece of memory in *READER*. See 0x0D – READER for details.

Operation:

$$d \leftarrow [READER]$$
$$READER \leftarrow READER + 1$$

Syntax:

```
read d
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	0	0	0	0
1	0	1	0	1	0	0	1	0

Cycles: 1

Words: 1

RET – Return from Subroutine

Description

Pop a program counter from the stack and jump to it. This is effectively a return at the end of a subroutine.

Operation:

$$SP \leftarrow SP + 1$$
$$r15 \leftarrow [SP]$$
$$PC \leftarrow r15$$

Syntax:

```
ret
```

Opcode:

This instruction is a compound instruction, comprised of a *POP* and *JMP*. Equivalent assembly is as follows:

```
pop r15
jmp r15
```

Cycles: 2

Words: 2

ROL – Rotate Left

Description

Perform a left bitwise rotation using the value of the carry flag in *FLAGS* to rotate values in the destination register. Rotates the value in the destination register by the value of the first 4 bits of the source register. This instruction is simply a left bitwise shift (*LSH*) with carry in enabled. Therefore, bits will not rotate into the carry flag, but will instead be determined by the expression listed in Values in FLAGS.

Operation:

$$d \leftarrow (d \ll r) \mid \text{c}$$

Syntax:

```
rol d, r
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	R	R	R	R
1	1	0	0	1	0	0	0	0

Cycles: 1

Words: 1

ROR – Rotate Right

Description

Perform a right bitwise rotation using the value of the carry flag in *FLAGS* to rotate values in the destination register. Rotates the value in the destination register by the value of the first 4 bits of the source register. This instruction is simply a right bitwise shift (*RSH*) with carry in enabled. Therefore, bits will not rotate into the carry flag, but will instead be determined by the expression listed in Values in FLAGS.

Operation:

$$d \leftarrow (d \gg r) \mid (c << 15)$$

Syntax:

```
ror d, r
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	R	R	R	R
1	1	1	0	1	0	0	0	0

Cycles: 1

Words: 1

RSH – Right Bitwise Shift

Description

Right logical bitwise shift the destination register by the source register's lowest 4 bits. Values greater than 0x10 in the source register will overflow the lowest 4 bits of the source register. For example, source register values of 0x1, 011, and 0x51 will all result in a right shift of 1 place.

Operation:

$$d \leftarrow (d \gg r)$$

Syntax:

```
lsh d, r
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	R	R	R	R
1	0	1	0	1	0	0	0	0

Cycles: 1

Words: 1

SBC – Subtract with Carry

Description

Subtract source register from destination register with carry flag from *FLAGS,* storing the result in the destination register.

Operation:

$$d \leftarrow (d - r) - c$$

Syntax:

```
sbc d, r
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	R	R	R	R
1	1	0	1	0	0	0	0	0

Cycles: 1

Words: 1

ST -- Memory Store

Description

Store a value from source register into a data memory.

Operation:

Where 'address' is either an immediate data address, or the value of a specified general-purpose register.

$$[address] \leftarrow r$$

Syntax:

```
st r -> [0x1234]
st r1 -> r2
```

Opcode (Register):

In the case of this instruction 'D' (Destination Register) refers to the register containing the data to be written to the address stored in the source register, 'R'.

Byte	0	1	2	3	4	5	6	7
0	D	D	D	D	R	R	R	R
1	1	0	0	0	0	0	1	0

Opcode (Immediate Data Address)

In the case of this instruction 'D' (Destination Register) refers to the register containing the data to be written to the immediate data address.

Byte	0	1	2	3	4	5	6	7
0	D	D	D	D	0	0	0	0
1	1	0	0	0	1	0	1	0
2	IDA	IDA	IDA	IDA	IDA	IDA	IDA	IDA
3	IDA	IDA	IDA	IDA	IDA	IDA	IDA	IDA

Cycles: 1

Words: 1 or 2

SUB – Subtract without Carry

Description

Subtract source register from destination, storing the result in the destination register.

Operation:

$$d \leftarrow (d - r)$$

Syntax:

```
sub d, r
```

Opcode:

Byte	0	1	2	3	4	5	6	7
0	D	D	D	D	R	R	R	R
1	0	0	1	0	0	0	0	0

Cycles: 1

Words: 1

WRITE – Write using WRITER Register

Description

Write a value from a source register to data memory using the *WRITER* special purpose register address, and post-increment the *WRITER* register. This is especially useful for reducing the number of cycles required to write sequential strings of memory. To use this instruction, store a pointer to the beginning of a piece of memory in *WRITER*. See 0x0C – WRITER for details.

Operation:

$$[WRITER] \leftarrow r$$
$$WRITER \leftarrow WRITER + 1$$

Syntax:

```
write d
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	R	R	R	R	0	0	0	0
1	1	1	0	1	0	0	1	0

Cycles: 1

Words: 1

XOR – Bitwise Exclusive OR

Description

Perform bitwise exclusive or (XOR) between destination and source registers, storing the result in the destination register.

Operation:

$$d \leftarrow d \text{ ^ } r$$

Syntax:

```
xor d, r
```

Opcode:

Byte	0	1	2	3	4	5	6	7
0	D	D	D	D	R	R	R	R
1	0	0	0	0	1	0	0	0

Cycles: 1

Words: 1

ZERO – Set Register to Zero

Description

Set the value of the destination register to 0. This is slightly better than the *LDI* instruction for this purpose because *LDI* uses 2 words of program memory, but *ZERO* only uses 1. *ZERO* is equivalent to *XOR* with both operands being the same register.

Operation:

$$d \leftarrow (d + 1)$$

Syntax:

```
zero d
```

Opcode:

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	D	D	D	D	D	D	D	D
1	0	0	0	0	1	0	0	0

Cycles: 1

Words: 1

Assembly Library

This repository contains a series of assembly files designed to make programming this MCU more streamlined. These are designed to be used with the CustomASM assembler, which allows easy scripted definitions for instruction syntaxes and opcodes. These files include assembly definitions for the entire instruction set, some GRAM (Graphics RAM) operations, some I/O operations, some common memory operations, and some other miscellaneous utilities.

Calling Convention

The assembly library provided follows a simple calling convention, which may be extended to other functions as well if desired. The following table lists the properties assigned to the general-purpose registers, and their status as either caller or callee maintained. Registers labelled caller maintained must be preserved by the caller of a function because that function may clobber them without saving their previous state. Registers labelled callee maintained must be preserved by the function being called because the caller expects their value to remain unchanged.

GPR	Purposes	Ownership
R0	Function argument 0, return value 0	Caller
R1	Function argument 1, return value 1	Caller
R2	Function argument 2, return value 2	Caller
R3	Function argument 3, return value 3	Caller
R4	Function argument 4, return value 4	Caller
R5	Function argument 5, return value 5	Caller
R6	Argv* pointer used when number of arguments exceeds 7, Function argument 6, return value 6	Caller
R7	Argc value used when number of arguments exceeds 7, Function argument 7, return value 6	Caller
R8		Callee
R9		Callee
R10		Callee
R11		Callee
R12		Callee
R13		Callee
R14		Callee
R15	May be clobbered at any time, including within compound instructions.	SCRATCH

include/arch.asm

arch.asm defines the instruction set for use with the CustomASM assembler. It also defines a series of constants used to describe various parameters of the device. At the end of this file is the starter snippet, a piece of code that contains the first instructions to be executed before calling main (). The constants defined are as follows:

Architecture Constant Definitions

Name	Value	Description
IOMEM_BASE	0x00	Base address of I/O mapped memory
IOMEM_SIZE	0x20	Size of I/O mapped memory
SRAM_BASE	0x20	Base address of general-purpose data memory
GRAM_WIDTH	160	Width of the GRAM internal buffer in words
GRAM_HEIGHT	120	Height of the GRAM internal buffer in words
GRAM_SIZE	0x9600	Size of the GRAM internal buffer in words
SRAM_SIZE	0x69DF	Size of general-purpose memory
GRAM_BEGIN	0x69FF	GRAM internal buffer base address.
PFLASH_SIZE	0xFFFF	Size of program memory
VECTORS_SIZE	*UNIMPLEMENTED*	
BOOT_SIZE	*UNIMPLEMENTED*	
PBOOT_SIZE	*UNIMPLEMENTED*	
SP	0x1	Address of *SP* register
FLAGS	0x2	Address of *FLAGS* register
RAND	0xA	Address of *RAND* register
WRITER	0xC	Address of *WRITER* register
READER	0xD	Address of *READER* register
PCAMSK	0x1B	
PCBMSK	0x1C	
GIMSK	0x1D	
IRETA	0x1E	
INTNO	0x1F	

Include/gram.asm

gram.asm defines buffer and color related constants and macros along with two useful functions for gram usage.

GRAM Constant Definitions

Name	Value	Description
BUFSEL0	0	Buffer select number 0 for *GRAM_FLIP* register
BUFSEL1	1	Buffer select number 1 for *GRAM_FLIP* register
GRAM_FLIP	0xB	Address for *GRAM_FLIP* register
WHITE	0	Color constant
BLACK	1	Color constant
RED	2	Color constant
GREEN	3	Color constant
BLUE	4	Color constant
YELLOW	5	Color constant
CYAN	6	Color constant
MAGENTA	7	Color constant
ORANGE	8	Color constant
PINK	9	Color constant

GRAM Macro Definitions

Prototype	Source	Description
rgb(r, g, b)	((1`1 @ r`5 @ g`5 @ b`5))	Bit combination to create 5-bit color in a 16-bit word using R, G, and B values.

GRAM Function Definitions

GRAM flip

```
1. void* gram_flip();
```

Description

Flip the GRAM buffer.

Returns

The address of the currently active buffer.

GRAM blit

```
1. void gram_blit_p(void* paddr, void* gram_addr, int height, int width);
```

Description

Blit a graphics buffer from program memory into a graphics buffer.

Parameters

1. void* paddr
    a. Program address source of the graphics data to copy.

2. void* gram_addr
    a. Data memory address as destination for graphics data to copy.
3. int height
    a. Height of the graphics (in words)
4. int width
    a. Width of the graphics (in words)

Include/io.asm

Io.asm includes some simple functions and defines constants relating to IO functionality.

IO Constant Definitions

Name	Value	Description
PORTA	0x3	Address of *PORTA* register
PORTB	0x4	Address of *PORTB* register
PINA	0x5	Address of *PINA* register
PINB	0x6	Address of *PINB* register
TTX	0x7	Address of *TTX* register
TRX	0x8	Address of *TRX* register
PULSE	0x9	Address of *PULSE* register

IO Macro Definitions

Prototype	Source	Description
sbmsk(bit)	(1 << bit)	Set bit mask (value can be OR'd to set a bit of a register).
cbmsk(bit)	(!(1<<bit))	Clear bit mask (value can be AND'ed to clear the bit of a register).

IO Function Definitions

```
1. void putchar(char16 c);
```

Description

Output a single char on TTX, using bit 0 of *PULSE* for the terminal peripheral in the simulation.

Parameter

1. char16 c
    a. The character to print.

```
1. char16 getc();
```

Description

Collect a single character on TRX using bit 1 of *PULSE* for the keyboard input peripheral in the simulation.

Returns

The next available character, or 0x0000 if none are available.

```
1. void put_ps(void* paddr)
```

Description

Print a string on the output by repeatedly calling putchar, reading a null terminated string from program memory. The input to this function must be in the program memory address space.

Parameter

1. void *paddr
    a. A null-terminated UTF-16 string in program memory.

```
1. void puts(void* saddr)
```

Description

Print a null terminated UTF-16 string from data memory by repeatedly calling putchar.

Parameter

1. void* saddr
    a. A pointer to a null-terminated UTF-16 string in data memory.

Include/mem.asm

Mem.asm includes a few memory related utility functions.

MEM Function Definitions

```
1. void p2s_memcpy(void* paddr, void* saddr, size_t size)
```

Description

Program to data memory copy. Copies data from program memory starting at paddr, to data memory starting at saddr, for 'size' words.

Parameters

1. void* paddr
    a. Program memory source address.
2. void* saddr
    a. Data memory destination address.
3. size_t size
    a. Words to copy.

```
1. void memcpy(void* dest, const void* src, size_t size)
```

Description

Copy data between two locations in data memory. The destination pointer must not be within the range of source pointer, to source pointer plus size. This will cause unexpected behavior as the function begins to overwrite the original source data.

Parameters

1. void *dest
    a. Destination pointer
2. const void *src
    a. Source pointer
3. size_t size
    a. Size in words to copy.

```
1.  void memset(void* dest, int value, size_t size)
```

Description

Set a region of data memory to a single value.

Parameters

1. void *dest
    a. Destination to set to value.
2. int value
    a. Value to assign.
3. size_t size
    a. Number of words to assign.

Code Examples

The following section lists a few examples of code written for the MCU and goes through how they work.

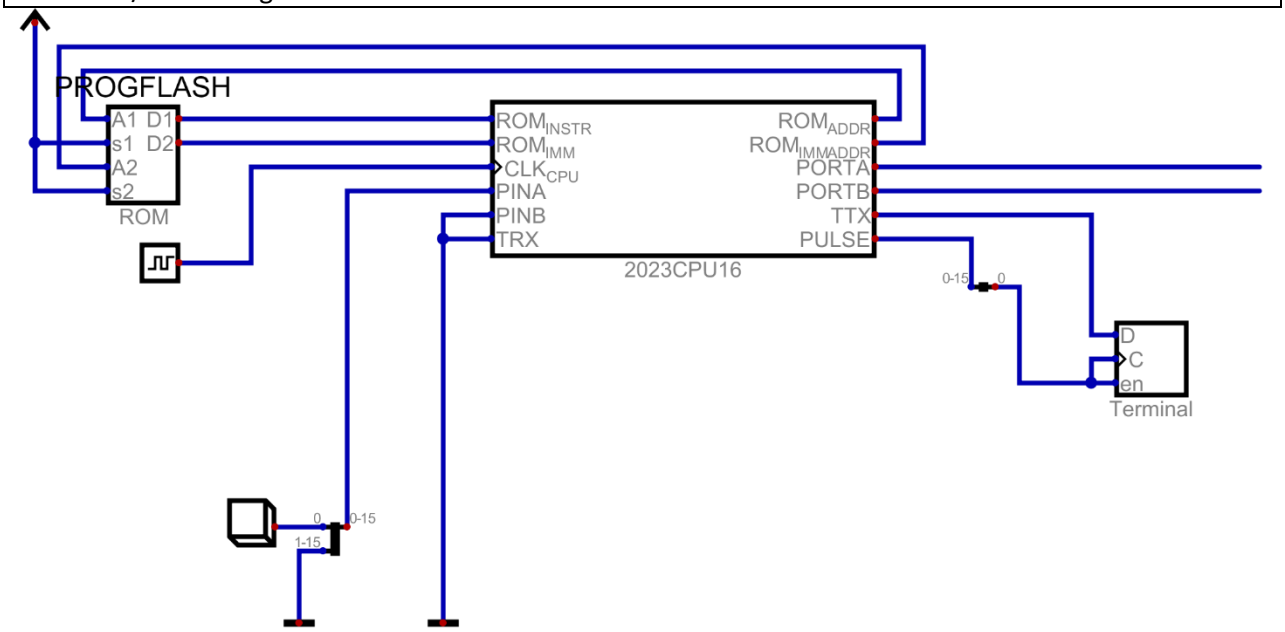
Hello World

software/Demo1.asm

```
#include "include/arch.asm"
#include "include/io.asm"
#include "include/mem.asm"


; Store the UTF-16, null-terminated string "Hello World!" in program memory
; using __test to mark the starting address.
__test:
    #d utf16le("Hello World!\0")
__test_end:

; Switch address space to data memory SRAM
#bank sram
; reserve space to copy Hello World
__test_sram:
#res (__test_end-__test)
__test_sram_end:
; reserve space to copy Hello World again
__test_memcpy_sram:
#res (__test_end-__test)
__test_memcpy_sram_end:

; Switch address space back to program memory
#bank pflash




main:

    ; load parameters for p2s_memcpy to copy Hello World
    ; into the space reserved at __test_sram
    ldi r0, __test
    ldi r1, __test_sram
    ldi r2, __test_end - __test

    call p2s_memcpy

    ; load parameters for memcpy to copy Hello World
    ; from the first reserved buffer to __test_memcpy_sram
    ldi r0, __test_memcpy_sram
    ldi r1, __test_sram
    ldi r2, __test_end - __test

    call memcpy

    ; load call puts on the finally copied value to print it to the terminal
    ldi r0, __test_memcpy_sram
```

```
50.     call puts
51.
52.     ret
```

hardware/Demo1.dig



Fibonacci

software/Fib.asm

```
 1. #include "arch.asm"
 2.
 3.
 4.
 5. main:
 6.     ; initial constants
 7.     ldi r0, 1
 8.     ldi r1, 1
 9.     ; main loop
10. .loop:
11.     ; sequence ...
12.     mov r2, r1
13.     add r2, r0
14.     ; check for carry, if so exit (found 16-bit max)
15.     j c .found_max
16.     ; setup for next sequence ...
17.     mov r0, r1
18.     mov r1, r2
19.     ; output
20.     st r2 -> [PORTA]
21.     ; repeat
22.     jmp .loop
23. .found_max:
24.     hlt
25.
```

Simulation and Software Tools

The main tools used to simulate and assemble software for this project are listed below.

CustomASM:

CustomASM is the custom assembler used to easily write assembly for this project. More information and installation information is available at https://github.com/hlorenzi/customasm/.

Hneeman/Digital:

Digital is a digital logic simulator which is used to run the simulations of this project. More information and installation information is available at https://github.com/hneemann/Digital.

License