DOG-1

Danny's Obtuse Gadget, version 1.0

(I'm writing this as if the thing was finished - far from it! - So this is all provisional design)

A very limited computer based on an Arduino Uno and a TM1638 I/O card. Runs as a virtual machine on top of the Arduino. Inspired by the single-board computers of the mid-1970's notably the MK14 and KIM-1. The architecture and instruction set is loosely derived from the processors used on these machines, the SC/MP and 6502 respectively.

Some novelty is offered by the hardware used is that it should be possible to interface with the built-in I/O of the Arduino. So eg. a digital thermometer could be made by attaching the appropriate sensor and programming the DOG-1 directly, without access to a PC.

Basic Architecture

- 16-bit addressing
- 8-bit instructions
- 8-bit data

Memory

uint8*t program[512]; // the code uint8* t pcStack[64]; uint8_t aluStack[64];

Registers

Operations will be carried out using the following registers:

16-Bit

- Program Counter (PC) - steps through program
- X Register (XR) - an auxiliary register
- PC Stack Pointer (PCSP) - for remembering the origin of subroutine jumps

8-Bit

- Accumulators A and B
- Status Register (SR) - system flags
- Auxiliary Stack Pointer (XSP) - for stack-oriented programming

Note - I think I'll change this rather 6502-like setup after reading the 6800 datasheet. That has 2 accumulators and one X index register. Seems to make for a simpler but more versatile instruction set.

Also quite interested in trying stack-oriented programming, 'check Here is a list of several stack manipulation operators, including SWAP' https://www.forth.com/starting-forth/2-stack-manipulation-operators-arithmetic/ Also Stack Operations in 6800 doc.

Addressing Modes

// copied from 6800 spec, need to tweak

- ACC - Accumulator In accumulator addressing, either accumulator A or accumulator B is specified. These are 1- byte instructions. Ex: ABA adds the contetns of accumulators and stores the result in accumulator A

- IMM - Immediate In immediate addressing, operand is located immediately after the opcode in the second byte of the instruction in program memory (except LDS and LDX where the operand is in the second and third bytes of the instruction). These are 2-byte or 3-byte instructions. Ex: LDAA #$25 loads the number (25)H into accumulator A

- ABS - Absolute In absolute addressing, the address contained in the second byte of the instruction is used as the higher eight bits of the address of the operand. The third byte of the instruction is used as the lower eight bits of the address for the operand. This is an absolute address in the memory. These are 3-byte instructions. Ex: LDAA $1000 loads the contents of the memory address (1000)H into accumulator A

- IDX - Indexed In indexed addressing, the address contained in the second byte of the instruction is added to the index register's lowest eight bits. The carry is then added to the higher order eight bits of the index register. This result is then used to address memory. The modified address is held in a temporary address register so there is no change to the index register. These are 2-byte instructions. Ex: LDX #$1000 or LDAA $10,X Initially, LDX #$1000 instruction loads 1000H to the index register (X) using immediate addressing. Then LDAA $10,X instruction, using indexed addressing, loads the contents of memory address (10)H + X = 1010H into accumulator A.

- IMP - Implied In the implied addressing mode, the instruction gives the address inherently (i.e, stack pointer, index register, etc.). Inherent instructions are used when no operands need to be fetched. These are 1 byte instructions. Ex: INX increases the contents of the Index register by one. The address information is "inherent" in the instruction itself. INCA increases the contents of the accumulator A by one. DECB decreases the contents of the accumulator B by one.

- REL - Relative The relative addressing mode is used with most of the branching instructions on the 6802 microprocessor. The first byte of the instruction is the opcode. The second byte of the instruction is called the offset. The offset is interpreted as a signed 7-bit number. If the MSB (most significant bit) of the offset is 0, the number is positive, which indicates a forward branch. If the MSB of the offset is 1, the number is negative, which indicates a backward branch. This allows the user to address data in a range of -126 to +129 bytes of the present instruction. These are 2-byte instructions.

nn = 2 hex digits nnnn = 4 hex digits

Mode	Assembler Format	Description

Immediate	#nn	Value is given immediately after opcode
Absolute	(nnnn)	Value is contained in the given address
Indexed	nnnn, X	-

STAa 99 10 ; store acc A at 0199 lo, hi

Flags

Bit	Flag	Name	Description
0	N	Negative	Set if bit 7 of ACC is set
1	V	Overflow	-
2	Z	Zero	-
3	C	Carry	-
7	X	Aux	-

I/O

TM1638 Card

	0	1	2	3	4	5	6	7
LEDs	N	V	Z	C	X	X	X	X
7-Segs	8	8	8	8	8	8	8	8
Buttons	o	o	o	o	o	o	o	o

Note : on the board switches are labeled S1-S8, but starting at 0 is less confusing while coding

TM1368	Arduino
VCC	5v
GND	GND
STB	D4
CLK	D7
DATA	D8

The LEDs will usually display the contents of the Status Register.

Generally, the 7-Segment displays 0-3 will display the current value (address) of the Program Counter. Displays 6 and 7 showing the contents at that address. All in hexadecimal.

The functionality of the I/O will depend on the system's mode : **Program** or **Run**. Display 4 shows the current mode, P or R.

Push-button 4 switches between these modes. At any time, pressing buttons 4 and 5 **together** will reset the PC to 0000.

Program Mode

Pressing the buttons 0-3, 6-7 will increment the value corresponding to that of the display above it. Programming is achieved by pressing button 3 to increment the PC (with overflow occurring, counting up on displays 0-2). Pressing button 7 will increment the value on display 7 (*without* overflowing to display 6), ditto for button

6/display 6, together providing the value at the given address.

Pressing button 5 will switch the response from increment to decrement. The PC buttons/display *does* carry values and wrap at max and min (0000). The code buttons/display act independently to each other and don't wrap in the <0 direction.

Pressing button 4 will switch to **Run** mode.

Double Key Presses

- 0 & 1 - full-on reset & wipe
- 4 & 5 - reset pc
- 0 & 4 - display Accumulators A, B
- 0 & 5 - display Index Register
- 0 & 6 - display PC Stack Pointer
- 0 & 7 - display Auxiliary Stack Pointer & status
- 0 & 3 - flip from single-step to free run

Programming from PC

There are some utilities in the python directory.

- **ass.py** : minimal assembler

This takes quasi-assembly language and looks up the corresponding hex values, producing a version suitable for uploading. The values are taken directly from definitions in DOG-!'s source code right now, so I can change things around without breaking anything.

Example assembly :

LDAi 66 ; put 0x66 in acc A STAa 07 00 ; store acc A at 0070 HALT

- **upload.py**

Will upload a DOG-1 program to the device of the USB/serial port. Hex values for the opcodes should be the first two characters on each line, everything else is ignored. Right now comms will be terminated on reaching an FF (HALT).

Example program for upload :

10 LDAi 66 ; put 0x66 in acc A 66 14 STAa 07 00 ; store acc A at 0070 07 00 FF HALT

(Only implemented enough for now to be able to test opcodes).

Run Mode

Initially the system will be halted at the current address. Pressing button 3 will single-step through the program (pressing buttons 0-3 will cause the PC to skip to the corresponding address [running or skipping code in between? TBD]).

Alternately the program may be run in real time by pressing button 5. Pressing this button again will halt the program.

The HALT opcode will terminate a program and wait for keyboard input before switching to Program mode and zeroing the program counter.

Special Instructions

- Pause

If the instruction PAUSE is encountered in a program, the program will freeze at this point at display 'PAUSE...'. The flags and registers maynow be inspected. Pressing key 4 sets the program running again.

Error Messages

xxxxnoPE - non-existent operation at xxxx xxxxChar - illegal character at xxxx (when uploading program)

Instruction Set

- System-related, starting with : 00 NOP
- Accumulator A
- Accumulator B
- PC-related, jumps etc. (including PC stack)
- Logic ops
- Accumulator arithmetic ops
- Auxiliary stack-related (I want to experiment stack-oriented programming/maths see https://www.forth.com/starting-forth/2-stack-manipulation-operators-arithmetic/ https://en.wikipedia.org/wiki/Stack-oriented_programming_language)
- Hardware-related Finally: FF HALT

note to self - things like LDA will have a version for each of the addressing modes, ~ 6, so it's probably an idea to hop 8 values between base versions...hmm, testing values for switch statements via masks?

Using index register - probably mainly for table lookup, maybe for subroutine-like things too should support ld, st, inc & dec, swap with PC, conditional swap

Ok, save long list until later, start with a subset...

Canonical version is in the code!!

IMPORTANT TODO : serial comms for save/load

Using the display etc. and Arduino I/O from code will need some specialised opcodes. Maybe : USE // to decouple device from system UNUSE PEEK & POKE - yes!!!! To set/get values on/from devices. Interrupt-driven bits?

Inst	Syntax	Mode	Size	C	Z	V	N	Symbolic	Description
NOP	------	----	----	-	-	-	-	--------	-----------
CLRS	------	----	----	-	-	-	-	--------	-----------
LDAi	------	----	----	-	-	-	-	--------	-----------
LDAa	------	----	----	-	-	-	-	--------	-----------
LDAx	------	----	----	-	-	-	-	--------	-----------
LDAxx	------	----	----	-	-	-	-	--------	-----------
STAa	------	----	----	-	-	-	-	--------	-----------
STAx	------	----	----	-	-	-	-	--------	-----------
STAxx	------	----	----	-	-	-	-	--------	-----------
LDBi	------	----	----	-	-	-	-	--------	-----------
LDBa	------	----	----	-	-	-	-	--------	-----------
LDBx	------	----	----	-	-	-	-	--------	-----------
LDBxx	------	----	----	-	-	-	-	--------	-----------
STBa	------	----	----	-	-	-	-	--------	-----------
STBx	------	----	----	-	-	-	-	--------	-----------
STBxx	------	----	----	-	-	-	-	--------	-----------
AND	------	----	----	-	-	-	-	--------	-----------
OR	------	----	----	-	-	-	-	--------	-----------
XOR	------	----	----	-	-	-	-	--------	-----------
COMA	------	----	----	-	-	-	-	--------	-----------
COMB	------	----	----	-	-	-	-	--------	-----------
ROLA	------	----	----	-	-	-	-	--------	-----------
RORA	------	----	----	-	-	-	-	--------	-----------
ROLB	------	----	----	-	-	-	-	--------	-----------
RORB	------	----	----	-	-	-	-	--------	-----------
SWAP	------	----	----	-	-	-	-	--------	-----------

CLRS	------	----	----	-	-	-	-	--------	-----------
SETS	------	----	----	-	-	-	-	--------	-----------
SETC	------	----	----	-	-	-	-	--------	-----------
CLC	------	----	----	-	-	-	-	--------	-----------
CLV	------	----	----	-	-	-	-	--------	-----------
BITAi	------	----	----	-	-	-	-	--------	-----------
BITAa	------	----	----	-	-	-	-	--------	-----------
BITAx	------	----	----	-	-	-	-	--------	-----------
BITAxx	------	----	----	-	-	-	-	--------	-----------
BITBi	------	----	----	-	-	-	-	--------	-----------
BITBa	------	----	----	-	-	-	-	--------	-----------
BITBx	------	----	----	-	-	-	-	--------	-----------
BITBxx	------	----	----	-	-	-	-	--------	-----------
PUSHXA	------	----	----	-	-	-	-	--------	-----------
POPXA	------	----	----	-	-	-	-	--------	-----------
PUSHXB	------	----	----	-	-	-	-	--------	-----------
POPXB	------	----	----	-	-	-	-	--------	-----------
SWAPS	------	----	----	-	-	-	-	--------	-----------
DUP	------	----	----	-	-	-	-	--------	-----------
OVER	------	----	----	-	-	-	-	--------	-----------
ROT	------	----	----	-	-	-	-	--------	-----------
DROP	------	----	----	-	-	-	-	--------	-----------
TUCK	------	----	----	-	-	-	-	--------	-----------
SPCa	------	----	----	-	-	-	-	--------	-----------
SPCx	------	----	----	-	-	-	-	--------	-----------
PUSHA	------	----	----	-	-	-	-	--------	-----------
POPA	------	----	----	-	-	-	-	--------	-----------
PUSHB	------	----	----	-	-	-	-	--------	-----------
POPB	------	----	----	-	-	-	-	--------	-----------
JMPi	------	----	----	-	-	-	-	--------	-----------

JMPa	------	----	----	-	-	-	-	--------	-----------
JMPr	------	----	----	-	-	-	-	--------	-----------
JSRa	------	----	----	-	-	-	-	--------	-----------
JSRr	------	----	----	-	-	-	-	--------	-----------
RTS	------	----	----	-	-	-	-	--------	-----------
BZS	------	----	----	-	-	-	-	--------	-----------
BZC	------	----	----	-	-	-	-	--------	-----------
BCS	------	----	----	-	-	-	-	--------	-----------
BCC	------	----	----	-	-	-	-	--------	-----------
BNS	------	----	----	-	-	-	-	--------	-----------
BNS	------	----	----	-	-	-	-	--------	-----------
BVS	------	----	----	-	-	-	-	--------	-----------
BVC	------	----	----	-	-	-	-	--------	-----------
BGE	------	----	----	-	-	-	-	--------	-----------
BGT	------	----	----	-	-	-	-	--------	-----------
BLT	------	----	----	-	-	-	-	--------	-----------
ADDA	------	----	----	-	-	-	-	--------	-----------
SUBA	------	----	----	-	-	-	-	--------	-----------
CMPA	------	----	----	-	-	-	-	--------	-----------
USE	------	----	----	-	-	-	-	--------	-----------
UNUSE	------	----	----	-	-	-	-	--------	-----------
RND	------	----	----	-	-	-	-	--------	-----------
DEBUG	------	----	----	-	-	-	-	--------	-----------
OK	------	----	----	-	-	-	-	--------	-----------
ERR	------	----	----	-	-	-	-	--------	-----------
HALT	------	----	----	-	-	-	-	--------	-----------

Demo code

- 16x16bit multiply
- 16x16bit multiply
- Pseudorandom number generator

as in https://en.wikipedia.org/wiki/Linear-feedback_shift_register

Games? Lunar Lander? noughts & crosses? Space Invaders!!!!!!

Using Arduino I/O - digital thermometer? beepy machine?

See Also

This blog post gave me enough of how-to on interfacing with the TM1638 to get started:

Using a TM1638-based board with Arduino

See also :

- Retro Computing - loads on single-board computers
- KIM Uno - remake of a 6502 SBC on the Arduino