xDebugger
65C02 Debugger

Last update: 10/10/2021
Bob Applegate bob@corshamtech.com
Corsham Technologies, LLC
www.corshamtech.com

1 Introduction

xDebugger is a fairly complete tool for debugging 65C02 programs under the xKIM monitor. It was written because of a long-standing desire for a better way to debug assembly language code on either a KIM-1 or a KIM Clone computer system.

Full source code is provided so anyone can make changes, take useful code for other projects, etc.

1.1 Key features

- Load Intel hex files from either the console or an SD Card.
- Examine/modify registers.
- Breakpoints.
- Single-step and step-over execution.
- Disassembler
- Mini assembler

The code is written in plain 6502 instructions so it can work on any 6502-based processor. The assembler and disassembler support the WDC65C02 instruction set.

1.2 Requirements

- KIM compatible monitor, although similar functions are available in other monitors.
- xKIM monitor v1.8 or later. Some features of xDebugger require subroutines in xKIM.
- For SD related commands, a Corsham Technologies SD Card System.
- At least 6K of RAM starting from C000.

2 Command Line

All input to the command line must be in upper case.

The user types a command line and then presses RETURN to execute it. Command line processing is a bit loose, so there are scenarios where "bad" input is still interpreted without complaints.

When numeric parameters are required they are entered as hexadecimal values. Each valid digit causes the current value to the shifted left four bits and then the new digit added. There is no range checking done, so 1234567890 will result in the value 7890. If you set an 8 bit register to a larger value, only the bottom 8 bits will be used. No input results in a value of zero.

3 Commands

3.1 ? - Help

Provides a very brief bit of help, showing all the commands. There is no context sensitive help, and no additional help within the debugger.

3.2 A - <addr> - Start miniassembler at address.

The mini assembler makes it easy to enter short test programs or pieces of code for testing. See the Mini Assembler section for details.

3.3 B – List Breakpoints

List all breakpoints and indicates if breakpoints are active or not.

3.4 BD – Disable Breakpoints

Disables all breakpoints. All breakpoints remain but simply are not active until enabled again.

3.5 BE – Enable Breakpoints

Enables breakpoints.

3.6 BC [<addr>] - Clear Breakpoints

Clears breakponts.

With no parameter, all breakpoints are cleared and breakpoints are disabled. If an address is supplied then only that one breakpoint is removed and breakpoints remain enabled. There is no warning if the address does not have a breakpoint.

3.7 BS <addr> - Set Breakpoint

Sets breakpoint at specified address. If a breakpoint for that address already exists, it will be used rather than creating a new one. This also enables breakpoints if they were disabled.

3.8 C – Continue Execution

Continue execution at the current PC and with the registers loaded with their saved values. This is meant to be used after a breakpoint.

3.9 D – Dissemble at Current PC

Disassembles at current PC.

3.10 D <addr> - Disassemble One Instruction

Disassemble one instruction at the specified address.

3.11 D <addr> <addr> - Disassemble Range

Disassemble between a range of addresses.

3.12 E <addr> - Edit Memory

Edit memory starting at specified address. This will display an address and the current contents. To change the value enter exactly two hex digits, then it will advance to the next address.

Pressing RETURN will advance to the next location without modifying the current one.

Pressing Backspace will move to the previous location.

Pressing R will ask which address to branch to. Enter a four digit address and the offset to that address will be placed in the current location.

3.13 H <addr> <addr> - Hexadecimal Dump

Performs a hex dump of the specified address range.

3.14 J <addr> - Jump to Address

Jump to specified address. Register values are loaded with the current values.

3.15 K – Perform SD Card Directory

Perform a directory of the SD card.

3.16 L – Load Intel Hex from Console

Load Intel hex file from the console. If the file sets the xKIM AutoRun vector then that address will be displayed and the Program Counter loaded with the address. This does not automatically run the program.

3.17 L <filename> - Load Intel Hex from SD File

Load hex file from SD card. If the AutoRun address is set, behavior is the same as for the L command.

3.18 R [<name> <value>] - Display or Set Registers

Without any argument, display the registers and also disassembles the instruction at the current PC.

To modify a register, the name must be A, X, Y, SP, PC of F and the register will be loaded with the value.

3.19 S – Single Step

Execute the instruction pointer to by the PC register with all registers loaded with their proper values. Stops at the next instruction. It properly handles instructions that cause changes in program flow (JSR, JMP, branches).

3.20 SO – Step Over

Exactly the same as S except that if the current instruction is a JSR, it will stop at the instruction following the JSR and not step into the subroutine.

3.21 Q - Quit

Quit back to xKIM. It also restores IRQ and NMI vectors to what they were when xDebugger started running.

4 Mini Assembler

The mini assembler is not a complete assembler by any means but it does allow the user to enter 65C02 code for testing. It is started with the first addres to be used and then prompts with the current address, a colon, and then a space. Mnemonics can be typed in there.

4.1 Recognized Mnemonics

ADC, AND, ASL, BBR0, BBR1, BBR1 BBR2, BBR3, BBR4, BBR5, BBR6, BBR7, BBS0, BBS1, BBS2, BBS3, BBS4, BBS5, BBS6, BBS7, BCC, BCS, BEQ, BIT, BMI, BNE, BPL, BRA, BRK, BVC, BVS, CLC, CLD, CLI, CLV, CMP, CPX, CPY, DEC, DEX, DEY, EOR, INC, INX, INY, JMP, JSR, LDA, LDX, LDY, LSR, NOP, ORA, PHA, PHP, PHX, PHY, PLA, PLP, PLX, PLY, RMB0, RMB1, RMB2, RMB3, RMB4, RMB5, RMB6, RMB7, ROL, ROR, RTI, RTS, SBC, SEC, SED, SEI, SMB0, SMB1, SMB2, SMB3, SMB4, SMB5, SMB6, SMB7, STA, STP, STX, STY, STZ, TAX, TAY, TRB, TSB, TSX, TXA, TXS, TYA, WAI

4.2 Syntax

The syntax follows normal assembler syntax except there are no labels. The assembler displays the address and the user enters the mnemonic from the list above, followed by any additional arguments. All input is in upper case and all input values are in hex.

```
ROL A            Accmulator
LDA #42          Immediate
PHA              Implied
LDA 1234         Absolute
BCC 1234         Relative uses the target address
LDA 42           Zero page
JMP (1234)       Indirect
LDA 1234,X       Absolute indexed with X
LDA 1234,Y       Absolute indexed with Y
LDA (12,X)       Zero page indexed indirect
LDA 12,X         Zero page indexed with X
LDA 12,Y         Zero page indexed with Y
LDA (12)         Zero page indirect
LDA (12),Y       Zero Page Indexed Indirect with Y
JMP (1234,X)     Absolute Indexed Indirect with X
```

The WDC65C02 has several instructions with two parameters which all follow the same general format:

```
BBC0 12,0300
BBS7 5,402
```

Notice that the bit number is at the end of and part of the mnemonic. It is followed by at least one space, the zero page address to test, a comma, and then the target address. There are alternate ways to express these mnemonics but this is the one supported.

4.3 Usage

The user simply enters the instruction, presses Return and the assemblet will show the resulting code followed by a prompt for the next instruction. User input is in **bold**:

```
DBG> A 200
0200: LDA #0
0200: A9 00
0202: INC A
0202: 1A
0203: CMP #4
0203: C9 04
0205: BNE 202
0205: D0 FB
0207: NOP
0207: EA
0208:
```

An empty line exits the assembler.

4.4 Weirdness

The assembler does not check much too closely, so it is possible to get unexpected results:

```
DBG> A 300
0300: LDA                Interpreted this as page zero addressing mode
0300: A5 00
0302: JMPPP 1234         Bad mnemonic is not recognized
0302: 4C 34 12
0305: LDX #1234          Value too large is truncated
0305: A2 34
0307:
```

5 Under The Hood

The best source of details for how the debugger works will be the source code, but this gives some high level overview. The disassembler is written in pure 6502 code so it can run on any varient of the 6502, but since it was primarily developed on a KIM Clone witch has a WDC65C02 processor, the debugger provides full support for it.

5.1 Interrupts

When it starts, the debugger saves the current NMI and IRQ vectors and installs its own ISR instead. Single stepping and breakpoints use the BRK instruction to gain control. Upon exit, the debugger restores the original vectors.

Can someone debug an ISR with the debugger? Maybe, but it'll be tricky and I have not personally done it so there are some thoughts but definitely not guaranteed to work. If NMI is used then it should work, but using IRQ takes some care. The new interrupt handler must save the current IRQ vector, and if the interrupt was not caused by the device, then the handler needs to call the saved IRQ vector. It will be tricky to make this all work.

5.2 Breakpoints and Single Stepping

Breakpoints and single stepping are done with the BRK instruction. When a breakpoint is set, the address of the breakpoint is saved along with the original opcode. A BRK is loaded and the entry is marked as in-use. If the breakpoint is hit while code is running, the ISR puts back the original instruction, then disassembles the instruction and displays register contents.

Single stepping is much more involved because the address of the next instruction must be determined so the BRK can be placed there. For instructions where there is no chance of a flow change, the determination is easy. For JMPs and JSRs, the target must be calculated. For any conditional branch, it must be determined if the branch will take place or not.

5.3 Disassembler

The disassembler is not at all elegant but gets the job done. It uses several tables for the decoding, mostly because trying to write elegant code to decoder the WDC65C02 instrution set is a mess, and it is much easier to understand the table approach. If you only use a plain 6502 or a 65C02 from another manufacturer, feel free to modify the disassembler. You will also need to change the code in the Step subroutine as it knows about instructions.

5.4 XKIM Reliance

Since the xKIM monitor provided a number of useful commands already, it made no sense re-coding the same commands in the debugger, so hooks were added in xKIM to provide access to its internal subroutines. For example, Edit, Hexdump, hex file loading and disk directory commands. Those functions were added to xKIM in version 1.8 so any earlier versions will not support xDebugger.

If you are not using xKIM then the easiest thing to do is grab xKIM sources and pull out the subroutines you need from there.

5.5 Supporting Other Processors

There are several varients of the 6502 processor, and the debugger can be tweaked for them. There are two main places where changes need to be made. The disassembler (dis.asm) is the main area. If you wanted to support just an original 6502 without the WDC extensions then changing the addmodeTbl table will be sufficient. Changes must also be made in the Step subroutine (disassem.asm).

For adding new opcodes there is a table called MNEMS which needs to be updated. Fortunately there is a small C++ program called mnem.cpp which can generate that table for you. Add opcodes in the file, compile, run, and paste the output into your version of the dis.asm file.

If you were to make changes, my suggestion would be to add a meaningful label name which selects the logic for your processor. Ie, do not delete what is there, just add new code that is conditionally assembled as needed.

6 Examples

6.1 Loading and Stepping

```
DBG> L

Waiting for file, or ESC to exit...
:00000001FF0225C904D0FBEAEAC6   File uploaded via terminal emulator
          Success!
Auto-run address and PC set to 0200
DBG> R
0200  A9 00      LDA  #00      PC:0200 A:4C X:FF Y:FF SP:01 Flags:N----I--
DBG> S
0202  1A         INC  A        PC:0202 A:00 X:FF Y:FF SP:01 Flags:-----IZ-
DBG> S
0203  C9 04      CMP  #04      PC:0203 A:01 X:FF Y:FF SP:01 Flags:-----I--
DBG> S
```

```
0205  D0 FB      BNE  0202     PC:0205 A:01 X:FF Y:FF SP:01 Flags:N----I--
DBG> S
0202  1A         INC  A        PC:0202 A:01 X:FF Y:FF SP:01 Flags:N----I--
DBG> S
0203  C9 04      CMP  #04      PC:0203 A:02 X:FF Y:FF SP:01 Flags:-----I--
DBG> S
0205  D0 FB      BNE  0202     PC:0205 A:02 X:FF Y:FF SP:01 Flags:N----I--
DBG> S
0202  1A         INC  A        PC:0202 A:02 X:FF Y:FF SP:01 Flags:N----I--
DBG> S
0203  C9 04      CMP  #04      PC:0203 A:03 X:FF Y:FF SP:01 Flags:-----I--
DBG> S
0205  D0 FB      BNE  0202     PC:0205 A:03 X:FF Y:FF SP:01 Flags:N----I--
DBG> S
0202  1A         INC  A        PC:0202 A:03 X:FF Y:FF SP:01 Flags:N----I--
DBG> S
0203  C9 04      CMP  #04      PC:0203 A:04 X:FF Y:FF SP:01 Flags:-----I--
DBG> S
0205  D0 FB      BNE  0202     PC:0205 A:04 X:FF Y:FF SP:01 Flags:-----IZC
DBG> S
0207  EA         NOP           PC:0207 A:04 X:FF Y:FF SP:01 Flags:-----IZC
DBG>
```

6.2 Editing

```
DBG> E 300
0300 00 A2       LDX #3
0301 02 03
0302 E3 CA       DEX
0303 FF 10       BPL
0304 00 Relative offset to: 0302 FD  Pressed R key for branch calculator
0305 FF EA
0306 00 .        Non-hex input stops edit mode
DBG>
DBG> E 303
0303 10
0304 FD .        Offset calculated from before
DBG>
```

6.3 Disassembler

```
DBG> D 200 208
0200  A9 00      LDA  #00
0202  1A         INC  A
0203  C9 04      CMP  #04
0205  D0 FB      BNE  0202
0207  EA         NOP
0208  EA         NOP
```

6.4 Breakpoints

```
DBG> B
Breakpoints:
Breakpoints are disabled
DBG> BS 205
DBG> B
Breakpoints: 0205
Breakpoints are enabled
DBG> J 200
0205  D0 FB      BNE  0202     PC:0205 A:01 X:FF Y:FF SP:01 Flags:N----I--
DBG>
```

Breakpoints can cause some weird side effects, such as this:

```
DBG> J 200
0205  D0 FB      BNE  0202     PC:0205 A:01 X:FF Y:FF SP:01 Flags:N----I--
DBG> C
0205  D0 FB      BNE  0202     PC:0205 A:01 X:FF Y:FF SP:01 Flags:N----I--
DBG> C
0205  D0 FB      BNE  0202     PC:0205 A:01 X:FF Y:FF SP:01 Flags:N----I--
DBG> C
0205  D0 FB      BNE  0202     PC:0205 A:01 X:FF Y:FF SP:01 Flags:N----I--
DBG>
```

Why is the value of A not changing? Because when the C command is executed, breakpoints are enabled again, and immediately breaks at address 0205. Ideally the breakpoint will not be active again until after the instruction at 205 executes but there is no way to do this is software.

Instead, after a breakpoint, either delete that breakpoint or single-step over the instruction and then do a Continue:

```
DBG> J 200
0205  D0 FB      BNE  0202     PC:0205 A:01 X:FF Y:FF SP:01 Flags:N----I--
DBG> S
0202  1A         INC  A        PC:0202 A:01 X:FF Y:FF SP:01 Flags:N----I--
DBG> C
0205  D0 FB      BNE  0202     PC:0205 A:02 X:FF Y:FF SP:01 Flags:N----I--
DBG> S
0202  1A         INC  A        PC:0202 A:02 X:FF Y:FF SP:01 Flags:N----I--
DBG> C
0205  D0 FB      BNE  0202     PC:0205 A:03 X:FF Y:FF SP:01 Flags:N----I--
DBG> S
0202  1A         INC  A        PC:0202 A:03 X:FF Y:FF SP:01 Flags:N----I--
DBG> C
0205  D0 FB      BNE  0202     PC:0205 A:04 X:FF Y:FF SP:01 Flags:-----IZC
DBG> S
0207  EA         NOP           PC:0207 A:04 X:FF Y:FF SP:01 Flags:-----IZC
DBG>
```

7 Resources

7.1 xDebugger Sources

https://github.com/CorshamTech/xDbg

7.2 xKIM Sources

https://github.com/CorshamTech/xKIM

7.3 Corsham Technologies, LLC SD Card System

https://www.corshamtech.com/product/sd-card-system/