

User Manual for the cpu6502 Instruction Set Simulator



Simon Southwell

January 2017

(Last updated 25th February 2024)

Contents

INTRODUCTION	3
FEATURES	3
SOURCE FILES	3
BUILDING CODE	4
API	4
INITIALISATION	5
LOADING A PROGRAM	5
EXECUTION	6
<i>Return Value</i>	7
INTERRUPTS	7
<i>Wait and Stop</i>	7
CALLBACKS	8
SAVE AND RESTORE	8
DISASSEMBLED OUTPUT	9
TIMING MODEL	10
TESTING	10
TEST PLATFORM	10
TEST CODE	11
EXECUTING TESTS	11
COVERAGE	12
SOURCE CODE ARCHITECTURE	13
MAIN EXECUTION FLOW	13
COMPILE OPTIONS	15
CASE STUDY 1: INTEGRATION INTO 'BEEBEM'	16
INTEGRATION DETAILS	16
CASE STUDY 2: WOZ MONITOR	19
CASE STUDY 3: MICROSOFT BASIC	19
CREDITS	20
PERFORMANCE	20
FURTHER READING	21

Introduction

This package comprises an instruction set simulator, modelling the NMOS 6502A 8-bit CPU and variants, with a C++ compatible API. The source is free-software, released under the terms of the GNU licence (see [LICENCE.txt](#) included in the package).

Features

Included Features:

- All supported core instructions for NMOS 6502A
- Support for 65C02 instructions
- Support for Rockwell and WDC instructions, including WAI and STP
- NMI and up to 16 wire-or'ed IRQs
- Configurable internal memory (compile option)
 - Supports up to 64K bytes
- Cycle count functionality for accurate timing
- Run-time disassembly
- Extensibility via callbacks
 - Intercept memory accesses via register of call back functions
- Compile options for Linux (makefile) and windows (MS Visual C++ express 2010)
 - Coverage compile support in makefile (lcov/gcov)
- Can read binary, Intel Hex format or Motorola S-Record program files (standalone compile).

The code is a simple exercise in modelling a popular 8-bit CPU. It comes with absolutely no warranties for accuracy, or fitness for any given purpose, and is provided 'as-is'. Hopefully it is useful for someone, and feel free to extend and enhance the model, and maybe let me know how it's going.

Simon Southwell (simon@anita-simulator.org.uk)
Cambridge, January 2017
Copyright © 2017-2024. All rights reserved.

Source files

Listed and described here are those source files that make up the [cpu6502](#) ISS. These are the source files needed for integration into other C or C++ environments, or to be compiled as a standalone executable.

The main header files comprise those listed below:

- [src/cpu6502_api.h](#)
- [src/cpu6502.h](#)
- [src/read_ihx.h](#)

For integrating the model with external programs only [cpu6502_api.h](#) needs be included in source code that references the API. The [cpu6502.h](#) header is only used by the internal source files and includes all the definitions and types needed by this code. The [read_ihx.h](#) is also an internal header, used for the program loading code.

The following listed files define the methods that belong to the class `cpu6502`, along with the above headers specific to those methods. The class methods are split over two files, but all belong to the single `cpu6502` class.

- `src/cpu6502.cpp`
- `src/read_ihx.cpp`

The entry point methods and program flow methods are all defined in `cpu6502.cpp`, along with the instruction methods themselves. There is almost a one-to-one mapping of 6502A instructions and the instruction methods, but a couple of methods double up for multiple instructions. Code disassembly is handled by internal methods, also defined in `cpu6502.cpp`, control of which is handled indirectly by the external API method `execute()`. The processing of the program files are handled in methods defined in `read_ihx.cpp`, with its header file as `read_ihx.h`.

Building code

Included in the package is a `makefile` to build the code under Linux (and, also, Cygwin) and support is also provided for MSVC 2010. By default (i.e. simply typing 'make' or building under visual C++) it will build the `cpu6502` (or `cpu6502.exe`) standalone target, and also linkable libraries. For windows `lib6502.dll` is built, along with the `.lib` and `.exp` support files for linking with other programs. Under Linux, a shared object `libcpu6502.so` and static library `libcpu6502.a` are built. The standalone build is an executable for running simple programs, particularly the self-test programs provided in the package—see “Testing” section below.

The makefile also, by default, builds the code with optimisations for fast execution, but these can be overridden by defining `COPTS` (e.g. `make COPTS=-g` for debug compilation). Additional user options can be added (such as turning on addition warnings, etc.) using `USROPTS`.

A target of 'test' can be specified, which compiles the test images (`test/test.as65` and `test/test_65c02.a65`) and runs the tests on the compiled model. This requires the 'as65' assembler to be available (see “Testing” section below).

The executable can be compiled to have `gcov` and `lcov` support compiled in, by defining `DOCOV`, and building for a 'coverage' target (e.g. `make DOCOV=1 coverage`). When building for coverage, `COPTS` is overridden to force as '-g', and additional test options are used.

API

The API to the model is a C++ interface that consists of a single object (of class `cpu6502`, as defined in `cpu6502_api.h`) that has a set of methods for configuring the model, setting control of program flow, and running executable code. Definitions are provided in `cpu6502_api.h` needed to communicate with some of these methods and set their parameters. This is all described in the sections to follow. In summary, the methods are:

```

        cpu6502 ();

void      reset          (cpu_type_e mode = DEFAULT);

void      nmi_interrupt  (void);
void      activate_irq   (const uin16_t id = 0);
void      deactivate_irq (const uin16_t id = 0);

wy65_exec_status_t execute (const uint32_t icount      = 0,
                           const uint32_t start_count = 0xffffffff,
                           const uint32_t stop_count  = 0xffffffff,
                           const bool    en_jump_mrks = true);

void      run_forever    (bool disassem = false);

void      register_mem_funcs (wy65_p_writemem_t p_wfunc,
                              wy65_p_readmem_t  p_rfunc);

int       read_prog       (const char      *filename,
                           const prog_type_e type    = HEX,
                           const uint16_t  start_addr = 0);

int       save_state      (FILE* fp);
int       restore_state   (FILE* fp);
int       save_mem         (FILE* fp);
int       restore_mem     (FILE* fp);

```

Initialisation

The model object is created by instantiating a variable of type `cpu6502` class or creating via 'new'. The constructor, `cpu6502()` does some internal initialisation (but mode of the 6502 CPU state itself) but requires no arguments externally. The CPU itself is initialised with the `reset()` method, which must be called before execution of a program, but can also be called subsequently to emulate a hardware reset.

The `reset()` method normally also requires no arguments, and this configures the model for running as an NMOS 6502A. However, the model can be configured to add additional instructions for later variants. To add 65C02 instructions, an argument of `C02` may be given. To additionally add the instructions supported by Rockwell and WDC (BBR, BBS, RMB, SMB), an argument of `WRK` may be given. The WDC unique instructions (WAI and STP) can be added on top of all the others with an argument of `WDC`. At each reset the model can be reconfigured with one of these arguments. To return to an NMOS 6502A model, for instance, an argument of `BASE` can be given. If a reset does not need to change the support mode it can be called without an argument, or with an argument of `DEFAULT`.

Loading a Program

The `cpu6502` model supports three different file formats for loading a program; raw binary, Intel Hex and Motorola S-Records. A method is provided for reading these files into memory:

- `read_prog()`

The method requires a filename string, which can be an absolute path, or a path relative to the execution directory. There are a couple of optional arguments; one for specifying the program file type (with an enumerated type, `prog_type_e`), and the other for a load address.

In addition, if the file type is `BIN`, for a binary file, the `read_prog()` method requires a load address argument, if this is not the default of 0, as this address is not contained within the program file itself. The default type is `HEX`, for Intel hex format files. S-Record files need an argument of `SREC`. The load address is ignored for Intel hex and Motorola S-Record files.

Execution

Once a model object is created, a program can be run via the `execute()` method. At its simplest, it is called without any arguments, to execute a single instruction, and update its internal state. Optional arguments are available to control run-time disassemble output:

icount	
<i>type</i>	const uint32_t
<i>valid values</i>	Any valid uint32_t number
<i>default value</i>	0
<i>description</i>	An input with the number of instructions that have currently been executed since reset

start_count	
<i>type</i>	const uint32_t
<i>valid values</i>	Any valid uint32_t number less than, or equal to stop_count
<i>default value</i>	0xFFFFFFFF
<i>description</i>	The value that icount must have reached before disassembly is enabled

stop_count	
<i>type</i>	const uint32_t
<i>valid values</i>	Any valid uint32_t number greater than, or equal to stop_count
<i>default value</i>	0xFFFFFFFF
<i>description</i>	The value that icount must not exceed after which disassembly is disabled

en_jump_mrhs	
<i>type</i>	bool
<i>valid values</i>	true / false
<i>default value</i>	true
<i>description</i>	Boolean to enable or disable 'jump' marks in the disassembled output.

	See “Disassembled Output” section.
--	------------------------------------

Return Value

The `execute()` method returns a structure of type `wy65_exec_status_t`, which has the following fields:

- `uint16_t pc`
- `uint8_t flags`
- `uint32_t cycles`

The `pc` and `flags` fields are direct copies of the state of the 6502A model’s program counter and status flags registers *after* the instruction that was just executed. The `cycles` field is the number of cycles taken to execute that instruction.

A `run_forever` method is also provided that wraps the `execute` method in an infinite loop if that’s all that’s required. It has a single argument, `disassem`, with a default value of `false`, so that disassembled output can be enabled when the model is run via the method.

Interrupts

The 6502 processor has both an NMI interrupt (an active low edge triggered interrupt) and maskable IRQ interrupt (active low level sensitive interrupt). The IRQ is often connected as a wired-or of several sources of interrupt, and the model provided support for this.

To emulate the event of the NMI input going low, the method `nmi_interrupt()` is provided. This takes no arguments and will cause the model to jump to the NMI vector on the next call of `execute()`.

For IRQ interrupts, the `activate_irq()` and `deactivate_irq()` methods are used. An optional argument `id` (with a default of 0) can be given to control up to 16 wired-or lines (0 to 15). An ID value outside of a 0 to 15 range means the call is ignored. The methods activate or deactivates these lines, and their state is checked each time `execute()` is called. If any are active *and* the model’s interrupt bit in the status register is clear, then the PC will jump to the IRQ vector address before executing from there.

If the system in which the model is integrated only has a single source of IRQ, or maintains wired-or state externally, then the methods may be called without an argument.

Wait and Stop

When the model is configured to support the WDC unique instructions WAI and STP execution of these instructions has an effect on the interrupt and execution behaviour. In this model, if either are executed then, assuming no calls to reset or active interrupts, subsequent calls to the `execute()` method will return a PC continually

pointing to the WAI or STP instruction location. On the first call that executes the instruction, a cycle count is returned for the execution of that instruction. Subsequent calls return a count of 0.

With the STP instruction only a call to `reset()` will terminate this condition, and the model will start executing from the reset vector's indicated location. With the WAI instruction, an NMI or an IRQ when not disabled (`I` bit clear) will make the model jump to the appropriate interrupt vector. An IRQ when the `I` bit is set will clear the wait status, and execution continues with the next instruction after the WAI instruction.

Callbacks

The ISS is a model of a processor core, and its main usage is as a component in a larger system level model. It has an internal memory model for convenience and to aid standalone testing, but it is via the callbacks that the model can be extended or integrated into a system model of arbitrary complexity. The model supports two user defined callbacks that can be registered with the model. These are for calling at each memory read or write access that the CPU performs.

The main use of this extension is to map peripherals (including more memory, if desired) into the memory space via the external memory callback functions, trapping accesses to addresses with memory mapped peripheral registers and implementing the functionality.

The callback registration function is described below:

<code>register_mem_funcs</code> (<code>wy65_p_writemem_t</code> <code>p_wfunc</code> , <code>wy65_p_readmem_t</code> <code>p_rfunc</code>)	
<i>description</i>	<p>The caller must provide as the parameter inputs, firstly a pointer to a function of type <code>wy65_p_writemem_t</code>, e.g.:</p> <pre>void w_func(int addr, unsigned char* data);</pre> <p>secondly a pointer to a function of type <code>wy65_p_readmem_t</code>, e.g.:</p> <pre>int r_func(int addr);</pre> <p>If the integrating system's own read and write memory function do not fit this model exactly, then wrapper functions should be added, and all memory accesses available from the CPU must be routed through these.</p>

When this method is used to register external memory access, the model's internal memory is effectively disabled. In addition, if the method is to be used, it must be called *before* any other method, including `reset()`.

Save and Restore

Four methods are provided in the API to save and restore internal model state to a file. The CPU state is separated from internal memory with the methods as internal memory need not be used if external functions have been registered with `register_mem_funcs()`, and hence does not need to be saved with the CPU state. All

four methods take a single argument of a pointer to a [FILE](#). The file is assumed to have been opened for writing successfully prior to calling these methods, and the flushing and closing operation also taken care of externally.

The methods for saving and restoring CPU state to a file are, respectively, [save_state\(fp\)](#) and [restore_state\(fp\)](#). If the file pointer argument is [NULL](#), the functions will simply return the size of the CPU state to be saved or restored, in bytes, without modifying any files or values. This is useful in extracting a length value of the data without any knowledge of data structure or types. When the [fp](#) argument is a valid [FILE](#) pointer, the returned value is the actual number of bytes written to, or read from, the file. If this is less than the value returned with a [NULL](#) pointer, then an error occurred.

The methods for memory saving ([save_mem\(fp\)](#) and [restore_mem\(fp\)](#)) work in exactly the same way as for the CPU state, only for the internal memory.

Disassembled Output

The model can output (to '[cpu6502.log](#)') fully disassembled output during run-time, showing program flow during a normal execution of code on the model. When disassembly is enabled the output looks something like the example fragment shown below:

331b	C6 0C	DEC	\$0C	a=80 x=0e y=ff sp=ff flags=f1 (sp)=33
331d	E6 0D	INC	\$0D	a=80 x=0e y=ff sp=ff flags=73 (sp)=33
331f	D0 E0	BNE	\$E0	a=80 x=0e y=ff sp=ff flags=f1 (sp)=33
	*			
3301	18	CLC		a=80 x=0e y=ff sp=ff flags=f1 (sp)=33
3302	20 86 35	JSR	\$3586	a=80 x=0e y=ff sp=ff flags=f0 (sp)=33
	*			
3586	A5 11	LDA	\$11	a=80 x=0e y=ff sp=fd flags=f0 (sp)=f1
3588	29 83	AND	#\$83	a=80 x=0e y=ff sp=fd flags=f0 (sp)=f1
358a	48	PHA		a=80 x=0e y=ff sp=fd flags=f0 (sp)=f1
358b	A5 0D	LDA	\$0D	a=80 x=0e y=ff sp=fc flags=f0 (sp)=b0
358d	45 0E	EOR	\$0E	a=e2 x=0e y=ff sp=fc flags=f0 (sp)=b0
358f	30 0A	BMI	\$0A	a=e2 x=0e y=ff sp=fc flags=f0 (sp)=b0

The default on calling [execute](#) is that no output is given. If the optional arguments are set such that it is enabled for a portion of the execution, then the above is the default display. Each line is a single instruction execution, with the 16 bit hex PC address followed by the raw bytes of the instruction (the opcode followed by zero to two operand bytes). The disassembled instruction follows. Any unrecognised instructions will be disassembled as [XXX](#), though an attempt at the size of the instruction is made (i.e. number of operands), based on its opcode. The final set of data is the main internal state of the CPU registers, A, X, Y, SP and flags (PSW). The contents of the value pointed to by the stack pointer is also shown ('[\(sp\)](#)'). Note that the values shown are as set *before* the instruction is executed.

The lines containing '*' are used to indicate where flow is dis-contiguous, due to branches jumps, interrupts etc. If these are not desired, then the [execute\(\)](#) method can set the [en_jump_mrk](#) argument to [false](#).

Timing Model

As has been mentioned before (see “Execution” section), the `execute()` method returns status that includes the cycles taken for the executed instruction. This value returned is cycle accurate for the instructions, including extra cycles for branching or page crossing etc. It does not include any extra delays for accessing memory etc.—for example, wait states via the `RDY` input. These must be added to the returned value to maintain accuracy, if emulating such systems.

Testing

Test Platform

As has been mentioned above, an executable environment, `cpu6502`, is constructed that instantiates the 6502 model, and provides sufficient control and facilities to allow the model to be fully tested. This includes a command line control interface for configuring the model and testing the instructions, as well as interrupts etc.

Detailed discussion of the code is not undertaken here—the code is not complicated, and inspection of the source should be sufficient—but a brief description of the program’s usage is given. The usage message for `cpu6502` is as follows:

```
Usage: cpu6502 [[-f | -I | -M] <filename>][-l <addr>][-s <addr>]
        [-d <addr>][-i <count>][-S <count>][-E <count>][-c][-D]

-f Binary program file name           (default test.bin)
-I Intel Hex program file name
-M Motorola S-Record program file name
-l Load start address of binary image (default 0x000a)
-s Start address of program execution (default 0x0400)
-S Disassemble start instruction count (default 0xffffffff)
-E Disassemble end instruction count   (default 0xffffffff)
-c Enable 65C02 features               (default off)
-D Disable testing and just run prog   (default enabled)
```

The first three options (`-f`, `-I` and `-M`) are used to select the input program file for loading into memory. By default the program expects a binary file named `test.bin`, but this can be overridden using the `-f` option. When a binary file is to be loaded, then a load start address must be supplied using the `-l` option, if this is not the default address (of `0x000a`), as this information is not contained within the raw data. If either an Intel hex file format or a Motorola S-record file format is available, then the `-I` and `-M` options (respectively) are used to specify the filename. These formats contain load address information in their records, and so do not need the `-l` option, which is ignored if specified.

If the program does not contain a record to load the reset vector (at `0xffffc/0xffffd`), then the execution start address must be specified using the `-s` option. Any valid address may be given, from `0x0000` to `0xffff`.

Disassembly is controlled with `-S` and `-E`. These specify the start and end instruction counts that disassembled output is active. By default the values are set to disable disassembled output. If `-S` is used to specify a start count of 0, the disassembly is continuously enabled. However the log file can fill very quickly, and the model runs

very slowly, so a better practice is to specify a start and end count which brackets the suspected error.

The `-c` option enables all the 65C02 additional instructions, including the Rockwell and WDC instructions. When this option is not specified all unmapped opcodes are treated like NOPS, though of varying sizes there would be instruction format.

The `-D` option disables the testing part of the code, and simply runs the loaded program until a termination condition. This allows the use of the standalone executable to be used to run arbitrary code that does not report test pass/fail status.

Test Code

The testing falls into two categories: testing of instructions and testing of interrupts. A piece of luck enabled the first of these to be covered with little effort. A 6502 assembly code test suite was written by Klaus Dormann and made available on github under terms of the GPL licence. There are two test programs we are interested in, which test the base instructions in one program, and the extended instructions (except WAI and STP) in the other. These were modified in a trivial way to write a value `0x900d` at location `0xffff8/0xffff9` (initialised to `0x0bad`) if the test passed. The code does a 'jump to this location' when complete (either good or bad)—i.e. it deliberately hangs. This can be detected externally by seeing a PC value the same on subsequent returns from `execute()`, and the test terminated. The code can be compiled by the `as65` assembler of Frank Cross and loaded with the command line options.

The testing of interrupts is done explicitly in a function (`interrupt_test()`) called from `main()`, by 'poking' values and opcodes into memory, and calling the interrupt API methods. The subsequent `execute()` PC and flags statuses are checked to validate correct operation. Similarly, testing of WAI and STP instructions is affected with a call to a function `wait_stop_tests()`, if extended instructions are enabled with the `-c` option.

Executing Tests

The tests are run by simply executing `cpu6502 -I test/test.hex`, for the base instruction tests, or `cpu6502 -I test/test_65c02.hex`, for the extended instruction tests, from the main directory (the defaults match the compiled test code), assuming that the test code has been compiled for Intel hex format with `as65`. E.g., from within the `test/` directory, execute the command:

```
as65 -s2 test.a65
```

Under Linux, the make file can be used to compile the model, test code and run it, by specifying a build of test (i.e. `make test`). The output should be something like that shown below:

```
Executing ./test/test.hex from address 0x0400 ...

*****
* PASS *
*****
```

Executed 30.65 million instructions (32.6 MIPS)

Executing ./test/test_65c02.hex from address 0x0400 ...

```
*****  
* PASS *  
*****
```

Executed 21.99 million instructions (33.5 MIPS)

The make file test build compiles the model with optimisations for fast execution (g++ with an option of `-Ofast`) and compiles the test code for Intel hex format. A debug build can be specified by overriding the `COPTS` definition (e.g. `make COPTS=-g test`).



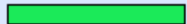


Coverage

Code coverage for the self-tests (for Linux only) was performed using `gcov` and `lcov`, with support within the makefile to build and execute to generate coverage data. Excluded from the coverage was any disassembler or debug output code as, although this can be covered to a level of 100%, it cannot be verified in an automatic self-test, and it does not affect the accuracy of the model.

The diagram below shows the LCOV report generated by executing the following commands:

```
make clean  
make DOCOV=1 coverage
```

The 'make clean' is important as the coverage is accumulative, and any old results must be deleted, unless the intention is to merge results. The report generated is created in the directory `cov_html/src`, and accessed via `index.html`.

<i>LCOV - code coverage report</i>					
Current view: top level - src			Hit	Total	Coverage
Test: cpu6502.info		Lines:	1148	1148	100.0 %
Date: 2017-01-30 15:06:32		Functions:	93	93	100.0 %
Filename	Line Coverage 			Functions 	
cpu6502.cpp		100.0 %	1070 / 1070	100.0 %	85 / 85
cpu6502_api.h		100.0 %	14 / 14	100.0 %	3 / 3
read_ihx.cpp		100.0 %	64 / 64	100.0 %	5 / 5
Generated by: LCOV version 1.12					

In order to obtain a goal of 100% coverage, some waivers on lines of code were needed on unreachable lines of code, (e.g. failed to open a file for reading) and the disassembler code, as mentioned above.

Source Code Architecture

It is not the intention to go into minute detail for the internal architecture of the model here, but a brief overview of the main program flow, internal state, and major structures is in order, to allow anyone wishing to understand or modify the code enough of a handle, that they can explore the details on their own.

Main Execution flow

The main entry point for the model is the `execute()` method, which advances by one instruction. Below is shown some pseudo code of the structure of this method.

```
// -----  
// Execute next instruction  
execute()  
    irq()  
  
    opcode = rd_mem(pc++)          // Fetch instruction  
    curr_instr = instr_tbl[opcode] // Lookup table  
  
    if disassembly enable...  
        disassemble()  
    endif  
  
    num_cycles = curr_instr()      // Execute instruction function  
  
    cycles += num_cycles           // Update cycles with returned count  
    return {num_cycles, pc, flags} // Return status  
end execute
```

The method starts by calling the `irq()` function to check for outstanding interrupts (see below). The opcode is fetched from the PC memory location, and a lookup is performed on the instruction table, which was initialised during `cpu6502`'s construction. The table consists of a 256 entry array, where each entry is of a structure type, `cpu6502::tbl_t`.

```
// Instruction table entry type.  
typedef struct  
{  
    const char*      op_str;  
    pInstrFunc_t     pFunc;  
    uint32_t         exec_cycles;  
    addr_mode_e      addr_mode;  
}  
tbl_t;
```

Each entry has a pointer to a string for the instruction (e.g. "ASL"), a pointer to the instruction function, the base number of cycles for the instruction and the address mode for the particular variant of the instruction. The function pointer is for a function that takes a single argument of type `cpu6502::opt_t` and returns an integer (with the final cycle count for the instruction, including run-time extras, such as page crossing, or branching).

The table lookup is stored locally in `curr_instr`, and the method pointed to by `pFunc` is executed, with the other information from the lookup passed in. The returned cycle count (including extras) is added to the total count, and some status is returned in a structure (`wy65_exec_status_t`) which includes the executed cycles, PC and flags.

The instruction methods vary in function in their detail, but all have a general structure that is similar to all of them. Below is shown some pseudo code for a generic instruction function.

```
// -----
// Generic instruction
<curr_instr>()
  addr = calc_addr()           // Get address of operand (and update PC)

  <do opcode function>         // Perform instruction's function

  update flags

  return opcode cycles          // Minumum #cycles
    [+ page crossed cycle] // page cross on relevant instructions
    [+ branch taken cycle] // Branch take cycle (branches only)

end <curr_inst>
```

Firstly the method fetches the address in memory where the operand data can be found via the `calc_addr()` method. This method is called in all cases, even for those opcodes that have no address such as those with implied addressing or accumulator operations, and these functions ignore `addr`. On entry the PC will be pointing passed the opcode itself to any operand bytes, and on exit the PC is pointing to the next instruction's opcode.

The particular operation of the function is then performed (ANDing, subtracting, memory read etc.), before the flags are updated as necessary (not all instructions alter the flags). The number of cycles is then returned. The `calc_addr()` method returns a flag if a page was crossed which adds a cycle, and branch instructions add a cycle if the branch is taken.

The `calc_addr()` method selects the address mode and generates a final location where the operand data can be found, incrementing the PC (where appropriate) in fetching the instruction argument bytes. Pseudo code for the function is shown below.

```
// -----
// Calculate address
calc_addr()
  case address mode of ...
    IND : addr = rd_mem16(rd_mem16(pc)); pc += 2;
    IDX : addr = rd_mem16(rd_mem(pc) + x); pc += 1;
    IDY : addr = rd_mem16(rd_mem(pc)) + y; pc += 1;
    ABS : addr = rd_mem16(pc); pc += 2;
    ABX : addr = rd_mem16(pc) + x; pc += 2;
    ABY : addr = rd_mem16(pc) + y; pc += 2;
    ZPG : addr = rd_mem(pc); pc += 1;
    ZPX : addr = rd_mem(pc) + x; pc += 1;
    ZPY : addr = rd_mem(pc) + y; pc += 1;
    REL : addr = rd_mem(pc) + pc + 1; pc += 1;
    IMM : addr = pc; pc += 1;
    IAX : addr = rd_mem16(rd_mem16(pc+x)); pc += 2;
    ZPR : addr = rd_mem(pc+1) + pc + 2; pc += 0; // Leave PC at ZP operand
    IDZ : addr = rd_mem16(rd_mem(pc)); pc += 1;
    ACC : NON: addr = don't care; // Implied, so no operands
  end case

  return addr, pc, page crossed flag
end calc_addr
```

The method is a simple case statement for all the supported addressing modes. The argument bytes following the location of the opcode (if any) are read, and any follow-up read for indirected modes, to create a final location which is returned, along with the updated PC and a page crossing flag (details not shown for clarity). In the case of accumulator and implied addressing modes, where there are no argument bytes, the method does nothing, but the address returned is `INVALID_ADDR`, so that calling routines can detect this condition.

The interrupt method (`irq()`) checks for outstanding interrupts and, if the `I` bit of the flags is clear, updates the PC with the interrupt vector after pushing the PC on to the stack, along with the flags. The flags then have the `I` bit set to mask further interrupts. The NMI is similar, except that there is no check, and the NMI is always actioned. Pseudo-code for these two functions is shown below:

```
// -----
// Interrupt check and update
irq()
  if active IRQ and flags' I bit clear...
    push(pc high byte)
    push(pc low byte)
    push(flags & ~BRK_MASK)

    set flags I bit
    pc = mem[] 16 bit value at 0xfffe and 0xffff
    cycles += IRQ_CYCLES
  end if
end irq

// -----
// NMI Interrupt
nmi()
  push(pc high byte)
  push(pc low byte)
  push(flags)

  set flags I bit
  pc = mem[] 16 bit value at 0xfffa and 0xfffb
  cycles += NMI_CYCLES
end nmi
```

Compile Options

By default, when `cpu6502` is compiled, it has the behaviour as described in the previous sections. However, it can be compiled with various definitions in order to modify its behaviour. There are, presently, three conditional compile definitions that can be set:

- `WY65_STANDALONE` : When defined a main function is included that has the testing facilities as defined above, and the default make and MSVC solution have this defined. If not defined, the code will compile without a `main()` function, so as to be included as part of another environment.
- `WY65_EN_PRINT_CYCLES` : Defining this enables the printing of cycle counts in the disassembled output.

- `WY65_MEM_SIZE` : The internal memory model of the ISS has a size of 65536 (the maximum address space of the 6502). This can be altered by defining this to a different value. If the `register_mem_funcs()` method is used to set external memory accesses in place of the internal memory it is useful to set this to 1 (though not necessary for functionality) to reduce the memory footprint of the unused array.

Case Study 1: Integration into ‘BeebEm’

In order to demonstrate the use of the model within a system context, the ISS was integrated into the BeebEm application that emulates the BBC micro and variants. The heart of the BBC microcomputer was a 6502 processor, and the BeebEm program has a model of the processor within it. This will be replaced with the `cpu6502` model as a study and example of integration, and for further testing of the model. Once integrated the BeebEm executable runs just as before, and is capable of running BBC micro software, but now using the `cpu6502` model.



The BeebEm application is a sophisticated model, supporting many variants of the BBC Micro. Beyond the emulation of the Basic BBC Model B, variants of the 6502 are modelled that have additional opcodes, both documented and undocumented. The `cpu6502` model only models the 6502A processor, and so the resultant BeebEm must stick to Model B configurations and will run only software that runs on variant.

Integration Details

Integration of the `cpu6502` model into BeebEm is straightforward. To integrate the `cpu6502` model into the BeebEm model, the instruction execution code and the code

for the two interrupts (NMI/IRQ) must be replaced. In addition, the `cpu6502` model must have access to the local memory system, and two BeebEm memory access routines (`BeebWriteMem()` and `BeebReadMem()`) need to be registered as callback functions with the model, via the `register_mem_funcs()` method. Fortunately, this can all be done with some minor changes to a single file—`6502core.cpp`—found in the `src/` directory of BeebEm package, version 4.14 for windows, which can be downloaded from github:

<https://github.com/stardot/beebem-windows/tree/4.14>

This package comes with a Visual C++ solution file (`BeebEm.sln`) for use with Microsoft Visual C++ 2008, but the testing done by the author is with 2010 express, which will convert this solution for use in that newer version. It does not support Linux (though `cpu6502` does), but the original BeebEm, from which the windows version is derived, can be found at:

<http://beebem-unix.bbcmicro.com/download.html>

The `6502core.cc` file of this version is not dissimilar to the windows derivative and might easily be modified to use the `cpu6502` model in a similar manner to the details given below, but this has not (yet) been tried.

The modified code of the windows version of BeebEm is not included in the package for `cpu6502`, but below is detailed the few changes required to update the source to use the `cpu6502` model. The modifications are such that the default behaviour is to run with the BeebEm processor model. In order to use the `cpu6502` model, definition below must be set at compilation (C/C++ Pre-processor configuration, added to the Pre-processor Definitions field):

- `TEST_CPU6502`

The details of the changes are given below, and the code may be cut and pasted directly into the file. No original code is altered, only new, compile dependant code added. The line numbers refer to the original, unmodified `6502core.cpp` file for version 4.14.

- The `cpu6502` model must be added to the BeebEm file. So after the system include file references, insert the following code between lines 32 and 33:

```
#ifdef TEST_CPU6502
#include "cpu6502_api.h"
cpu6502 cpu6502;
#endif
```

- The model will need access to the BeebEm's internal memory access functions, and then be reset whenever BeebEm is initialised or reset. The two BeebEm memory functions are registered with the `cpu6502` mode via the `register_mem_funcs()` method, and then the `reset()` method called. To implement this change, insert between lines 1066 and 1067, after line containing `'NMILock=0;'`, in the BeebEm function `Init6502core()`:

```

#ifdef TEST_CPU6502
    // Register the local memory access functions with the cpu6502 model
    // to allow use of BeebEm local memory.
    cpu6502.register_mem_funcs (BeebWriteMem, BeebReadMem);

    // Reset the cpu6502 model
    cpu6502.reset();
#endif

```

- The BeebEm main executions functions is called `Exec6502Instruction()`, and contains its 6502 functionality, both for instruction executions and NMI/IRQ interrupts. The instruction execution code is ifdef'd out and replaced with `cpu6502` calls. Insert between lines 1202 and 1203, after the call to `'AdvanceCyclesForMemRead()'` in `Exec6502Instruction()`:

```

#ifdef TEST_CPU6502
    // Execute a single instruction in the model
    wy65_exec_status_t status = cpu6502.execute();

    // Update local variables with returned status
    Cycles                += status.cycles;
    ProgramCounter        = status.pc;
    PSR                   = status.flags;

    // Clear any active maskable interrupt
    cpu6502.deactivate_irq();
#else

```

- To end the `#ifdef/#else` of the above code, insert between lines 2250 and 2251, just before `'PollVIAs(Cycles - ViaCycles)'`, in function `Exec6502Instruction()`:

```

#endif

```

- For the IRQ code, the `DoInterrupt()` function is ifdef'd out and replaced with `cpu6502` calls. In particular, a call to the `activate_irq()` method is made. The equivalent call is made to `deactivate_irq()` after the call to `execute()` (see above). Insert between lines 2260 and 2261, before `'DoInterrupt()'`:

```

#ifdef TEST_CPU6502
    cpu6502.activate_irq();
    IRQCycles = 7;
#else

```

- To finish the `#ifdef/#else` pair, insert between lines 2261 and 2262, after the function `'DoInterrupt()'`:

```

#endif

```

- For the NMI code, the `DoNMI()` function is ifdef'd out and replaced with `cpu6502` a call to the model's `nmi_interrupt()` method. Insert between lines 2270 and 2271, before `'DoNMI()'`:

```

#ifdef TEST_CPU6502
    cpu6502.nmi_interrupt();
    IRQCycles = 7;

#else

    • To finish the #ifdef/#else pair, insert between lines 2271 and 2272, after
    'DoNMI();':

#endif

```

The above additions are all that is required to integrate the `cpu6502` model into BeebEm. When compiled with `TEST_CPU6502` defined, the resultant executable will be using the model in place of its own 6502 ISS.

Case Study 2: Woz Monitor

In the `wozmon/` directory contains a model that instantiates the `cpu6502` CPU model and has a rudimentary PIA model mapped into memory space at the location used in the Apple I computer for user input and display output. This maps the PIA registers as follows:

- KBD 0xD010
- KBDCR 0xD011
- DSP 0xD012
- DSPCR 0xD013

The model and the monitor code can be compiled using the provided `makefile` which can build for both Linux and Windows (using `MSYS2/mingw-64`). The Linux code compilation depends on the availability of the `CC65` toolchain for assembling and linking. In particular that `ca65` and `ld65` are in the `PATH`. The source code repository found [here](#).

The generated model executable (`main.exe`), when run, will load the compiled monitor program at location `0xFF00` in main memory and automatically run it from there. A backslash prompt of the monitor should then be displayed and commands may be typed in to use the program to inspect and set memory.

A useful reference on using the program can be found [here](#), and the Apple I manual is available from the Computer History Museum [here](#), which also details the use of the monitor and even has a listing.

Case Study 3: Microsoft Basic

The `msbasic/` directory contains a version of Microsoft Basic ported for the `cpu6502` model. It uses the model from the `wozmon` directory and the `makefile` in this directory can be used to compile basic and the model (type `make`). This uses the `CC65` toolchain (see Case Study 2 above). Building generates a `main.exe` executable and a `cpu6502.bin` binary. To run MSBASIC on the model you can use `make run`. Alternatively the executable can be run directly with the following command line options (to suppress line feed generation):

```
main.exe -n
```

The usage message (use the -h option) for the executable is:

```
Usage: main.exe [-f <filename>][-l <addr>][-t <program type>][n][-d]

-t Program format type           (default BIN)
-f program file name             (default cpu6502.[ihex|bin] depending format)
-l Load start address of binary image (default 0x8000)
-r Reset vector address          (default set from program)
-n Disable line feed generation  (default false)
-d Enable disassembly            (default false)
```

The format type is either HEX or BIN for Intel hex format or binary files. The default program can be overridden with -f, where the default name will be either cpu6502.ihex or cpu6502.bin, depending on the type (default BIN). The load address (-l option) can be overridden for binary files (Intel Hex files ignore this parameter) and the reset vector value updated (-r option) to jump to a given location on reset. The -n option disables generating linefeed on carriage return characters, and the -d option enable generation of disassembly output from the cpu6502 model.

When run, you will be asked for a memory size, but pressing enter without giving a number instigates an auto-detection of RAM size. You will then be asked for a terminal width, and you can just press enter for this as well. You will then be in MSBASIC. Note that a running basic program can be interrupted with <ESC> and the model executable exited with ^C.

Credits

Derived from the Ben Eater (@beneater) [project](#) which was forked from the Michael Steil (@mist64) [project](#). See also Ben Eater's [YouTube video](#).

Performance

Running the standalone tests executes some 30 million instructions, and the program contains code for measuring the time it takes to execute the main instructions' test loop. The code is measured with optimised compilations (i.e. Release mode for MSCV, and -Ofast for gcc). The platform used was an Intel® i7 920 CPU, running at 2.67GHz, with a system having 6GB RAM on an ASUS P6T SE Motherboard.

The results are summarised in the table below:

OS	Compiler	Perfomance
Windows 10	MSVC Express 2010	32.5 MIPS
Ubuntu 16.04 LTS	gcc v5.4.0 (-m32)	31.7 MIPS
	gcc v5.4.0 (-m64)	34.7 MIPS
Cygwin	gcc v5.4.0 (-m64)	34.3 MIPS

It is interesting to note that a model of the 8 bit processor, introduced in 1975, runs at up to 17 times faster than that actual hardware used on, say, the Acorn BBC Micro, on what is a fairly modest desktop computer, whose particular processor was launched in 2008, and illustrates the rapid advancement in the intervening, though relatively short, time period.

The performances measured far exceed anything that might be needed for any real-time application using the model (such as BeebEm, for example), and no optimisation of the code has been done to increase the performance of the code. I'm sure that if some effort were put into optimising the code further speed increases would be possible—this might be useful if integrating in a larger system model, where the processor model mustn't be the limiting factor in order to achieve real-time simulation, or if the platform is a much lower spec. machine. In reality, if using the model to simulate a system in real time, some synchronisation between the model's cycle counts, simulated target frequency, and a system clock needs to be done, just as is done with the BeebEm simulator.

Further Reading

- [1] MCS6501 – MCS6505 Microprocessors Datasheet, MOS Technology Inc., August 1975
- [2] R650X and R651X Microprocessors (CPU), Rev 8, Rockwell, June 1987
- [3] SY6500/MCS6500 Microcomputer Family Programming Manual, Synertek Inc, August 1976
- [4] BeebEm website, <http://www.mkw.me.uk/beebem/> , retrieved Jan 2017.
- [5] BeebEm for UNIX website, <http://beebem-unix.bbcmicro.com/>, retrieved Jan 2017
- [6] 6502_65C02_functional_tests git hub page,
https://github.com/Klaus2m5/6502_65C02_functional_tests/ , Klaus Dormann,
retrieved
Jan 2017
- [7] Kingswood Consulting's as65 Cross Assembler,
<http://www.kingswood-consulting.co.uk/assemblers/>, Frank Kingswood, retrieved
Jan 2017.