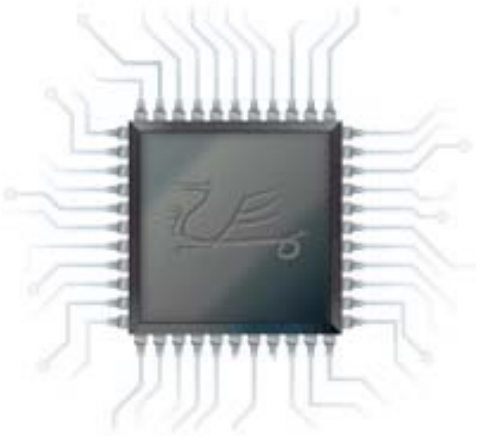
Reference Manual
for the 8051 micorcontroller
Instruction Set Simulator



Simon Southwell

August 2012

Introduction

The 8051 ISS comprises an instruction set simulator, modelling the 8051 8 bit processor. It implements all the non-optional features, and most of the optional features for that core. It has a C compatible API and is extensible to include additional modelled functionality. The source is free-software, released under the terms of the GNU licence (see LICENCE.txt included in the package).

Features

Features include

- All supported core instructions
- Built-in internal and external memory
- Built-in code memory
- Standard SFRs
- Execution timing model based on the Silion Labs CIP-51 core
- Model of timer 0 and 1
- Configurable user callbacks
    - Interrupt generating callback
    - External memory access callback
    - SFR access callback
    - General purpose callback
    - Serial interface callback
- Configurable execution breakpoints
    - On a given address
    - After a fixed number of cycles
    - On 'loop to self' locking instruction
- Access to internal memory and SFRs
- Access to external memory
- Compatible with GNU tool chain

The code is a simple exercise in modelling an 8 bit embedded 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-simulators.org.uk)

Cambridge, August 2012

Source Files

Listed and described here are those source files that make up the 8051 ISS library (e.g. `libcpu8051.so`). These are the source files needed for integration into other C environments. The source files for the example executable and test bench program, `cpu8051`, are not described (i.e. `cpu8051.c`). These are still freely available for use, under the terms of the GNU public license, but do not form part of the core functionality of the simulator, and are not documented.

The main header files comprise those listed below:

- `src/cpu8051.h`
- `src/8051.h`
- `src/read_ihx.h`

For integrating the model with external programs only `cpu8051.h` needs be included in source code that references the API. The `8051.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` header is only used by the internal source code to interface the program loading code.

The following listed files define the functions that belong to the 8051 ISS. The functions are split over several files, but all belong to the 8051 core model.

- `src/execute.c`
- `src/inst.c`
- `src/read_ihx.c`

Building Code

Included in the package is a `makefile` to build the code under Linux or Cygwin, and support is also provided for MSVC 2010. Under the UN*X systems, by default (i.e. simply typing '`make`') it will build the following:

- `cpu8051`
- `libcpu8051.a`
- `libcpu8051.so`

The first is an executable for running simple programs, particularly the self-test programs provided in the package—"Testing" section below. The next two are a static and dynamic library respectively, and are the libraries an external program can use to link with the model, choosing the appropriate one depending whether static or dynamic linking was most appropriate for the particular application. The API for the libraries, and its use, is described in the "API" section below.

The makefile also, by default, builds the code with debug information (with `gcc` option `-g`) and as position-independent code (with option `-fPIC`---though this option is not needed in Cygwin). These are defined in the make variable "`COPTS`", and can be overridden at the make command line.

Support for MSVC 2010 is provided, with a solution file (`.sln`) in the `msvc/` directory, along with the minimal set of project files to read in to the MSVC 2010 IDE, and compile and run the model, but if `MSBuild.exe` is in the `PATH` under Cygwin, then the makefile has support to build from the

command line with "`make MSVC`". The `MSBuild.exe` executable is part of Microsoft.NET, and thus can normally be found in a directory (for example, under Cygwin) such as:

```
<cdrive_path>/Windows/Microsoft.NET/Framework/v4.0.30319
```

The `<cdrive_path>` is the Cygwin path to the windows disk (most likely `/cygdrive/c`) and the final directory name will depend on the particular version of Microsoft NET installed. For 64 bit machines, a 64 bit version of the executable will be under Framework64.

By default, a make build for MSVC builds a "Release" executable, which is placed in the same directory as for the other builds of the makefile. If a "Debug" version is required, then the default can be overridden via the `MSVCCONF` make variable---i.e.:

```
make MSVCCONF="Debug" MSVC
```

Like the make for UN*X, the MSVC build produces a `cpu8051.exe` executable, but only a single library, `libcpu8051.dll`.

API

The API to the model is a C interface that consists of a set of functions for configuring the model, setting control of program flow, and running executable code. Definitions are provided in `cpu8051.h` needed to communicate with some of these functions, and set their parameters. This is all described in the sections to follow.

Initialisation

There is little initialisation to be done on the 8051 model, and it can be run using all the default settings. The following allow some changes to the default settings before code execution:

`void set_verbosity_lvl (int lvl)`	
default	`VERBOSITY_LVL_OFF`

description	Allows the verbosity level to be changed. Its single parameter has only two valid values currently:
	
	`VERBOSITY_LVL_OFF 0`
	`VERBOSITY_LVL_1`
	
	As well as at initialisation, this can be called during execution. This is useful for debugging of long programs, which would generate a large output if verbosity specified from time 0. A break can be set up to return at a known point before the area of interest, and verbosity increased before continuing.

void set_output_stream (FILE* file_pointer)	
default	stdout
description	Set the file pointer to which all output is directed (e.g. verbose output)

void set_disable_lock_break () **void clr_disable_lock_break ()**	
default	Breaking on lock condition enabled
description	Set/clear the disable for breaking on lock condition, i.e.: `label0 : SJMP label0 (0x80 0xfe)` When enabled, the 'jump to self' instruction above will cause a program execution break.

Execution and Breakpoints

A single function call is used to run a program. The function loads a program from the specified filename, with two configurable breakpoint arguments and an enable for the timers.

int run_program (char* ihx_fname, int run_cycles, int break_addr, **int timer_enable)**	
default	Returns NO_ERROR

description	The `ihx_fname` specifies the program filename. This must be a valid intel hex file format 8051 program, otherwise an error results and a non-zero value is returned.
	
	The number of cycles to run is specified in the `run_cycles` parameter. This can be any valid positive number < 2^31, after which the program execution will break. Two special values are defined:
	
	`FOREVER`
	`ONCE`
	
	The first of these disables the break on cycle count, and the second single steps through the code.
	
	A break address (`break_addr`) may also be specified which will terminate the program execution if the PC hits the specified address. If the address is less than 0, the PC cannot reach this address, and breaking on address is disabled.
	
	The function returns and integer:
	
	`NO_ERROR  -- returned successfully`
	`> NO_ERROR  -- breakpoint or error`

Callbacks

The model supports four types of user defined callbacks that can be registered with the model. The first is a general purpose callback that is called at regular intervals, of lengths from a single instruction execution to a user defined sleep time. A similar callback is called regularly but, additionally, can return an interrupt status. A third callback can be registered for invocation on any access to external memory, whilst the final callback is called on any SFR access *not* to a standard SFR register, to allow extension of the SFR space (e.g. for SFR mapper peripherals).

int register_time_callback (ptimecallback_t callback_func)	
default	No callback
description	Registers a callback function to be called at set regular times. The registered callback functions must be of type `ptimecallback_t`, i.e. `int cb_func (int time, int *wakeup_time)` When the function is called, the current model time is passed in as the argument "`time`". By default the callback is invoked after every instructions execution, but the function can delay this invocation to a later cycle by returning a cycle time in the pointer "`wakeup_time`". The function can return an absolute cycle count, or as a delta by adding the offset to the value passed in as "`time`". This callback is useful for extending the model with functionality that causes no exceptions, such as a debug output port.

int register_int_callback (pintcallback_t callback_func, int callback_type)	
default	No callback

description	Registers a callback function to be called at set regular times, but also can generate an interrupt exception to the core. The registered callback functions must be of type `pintcallback_t`, i.e.
	
	`int cb_func (int time, int *wakeup_time)`
	
	When the function is called, the current model time is passed in as the argument "time". By default the callback is invoked after every instructions execution, but the function can delay this invocation to a later cycle by returning a cycle time in the pointer "`wakeup_time`". The function can return an absolute cycle count, or as a delta by adding the offset to the value passed in as "`time`".
	
	If the callback returns 0, then no exception is generated in the core. If the callback returns >0, then an interrupt is generated. The type of the interrupt is defined at registration with the "`callback_type`" argument. This can be one of two values:
	
	`INT_CALLBACK_EXT0`
	`INT_CALLBACK_EXT1`
	
	These map to the two external interrupts defined for the 8051.
	
	This callback is useful for extending the model with functionality that cause exceptions, such as a UART

int register_ext_mem_callback (pmemcallback_t callback_func)	
default	No callback
description	Registers a callback function that is called for any access to external memory. The function must be of type `pmemcallback_t`, i.e.: `int cb_func (int addr, int data, int rnw, uint8_t *mem, int time)` Upon invocation, an address is supplied (`addr`), along with a `data` value (valid only on writes), a `rnw` flag indicating read (1) or write (0), a pointer to a memory buffer (`*mem`) and the current time (`time`). If the callback does not model functionality located at the supplied address, then the function must access the buffer pointed to by `mem[]` on behalf of the model, returning any read value. If the callback models the functionality at the supplied address then it may respond as necessary, including returning any read value, and must not updated `mem[]`. This callback is useful for adding peripherals to the core, with memory mapped registers in the external memory space.

int register_sfr_callback (pmemcallback_t callback_func)	
default	No callback
description	Registers a callback function that is called for any access to the SFR registers outside of the standard set. The function must be of type `pmemcallback_t`, i.e.: `int cb_func (int addr, int data, int rnw, uint8_t *mem, int time)` The functionality is essentially the same as for the external memory callback (see above), but for non-standard SFR accesses.

Memory Access

The API allows inspection of the memory of the model, both external memory and internal memory (including SFRs). If callbacks are registered for access to these memories, these will be invoked by calls to these functions, allowing access to memory modelled externally to the core.

int get_iram_byte (int addr) **void set_iram_byte (int addr, int data)**	
description	Access internal memory, including SFRs. `get_iram_byte` does a read access to `addr`, returning the byte value. `set_iram_byte` does a write access to `addr`, writing the byte value in `data`.

`int  get_ext_ram_byte (int addr)` `void set_ext_ram_byte (int addr, int data)`	
description	Access external memories. `get_ext_ram_byte` does a read access to `addr`, returning the byte value. `set_ext_ram_byte` does a write access to `addr`, writing the byte value in `data`.

Internal State Access

Currently a single function is implemented for internal state access, fetching the current cycle time.

`int get_cycle_time (void)`	
description	return the current model cycle count.

Timing Model

The timings for various 8051 implementations are not standard and vary anywhere from 12 cycles for a particular instruction to 1 cycle for the same instruction in another device. The timing model used here is based on that for the Silicon Laboratories' CIP-51 core (see [1], section 8.1.1).

Each instruction has an base execution cycle count, which is added to the internal cycle count of the core when the instruction is executed. In addition, the jump instructions have an extra cycle if the jump is performed, which is added appropriately.

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

Below is shown some pseudo-code of the main program flow when executing a program. The main functions are shown as "`<funcname>()`", and the phrases between "<" and ">" describe local functionality. The indentation of the pseudo-code shows the calling hierarchy as implemented in the code.

```
run_program(){

      if timers enabled...
         register_time_cb( timer() )
      end if

     // Load program
      read_ihx()

      reset_cpu()

     // Main execution loop
      while no breakpoint condition ...

          process_interrupts() {

              if interrupts enabled and not already interrupting ...

                  if ext mem 0 callback and not sleeping ...
                      if ext_int_cb_0()
                          check if interrupting
                          if interrupting ...
                              interrupt(EXT0_INT_VECTOR)
                          end if
                      end if
                  end if

                  if timer 0 callback and not sleeping ...
                      if timer_cb() ...
                         check if interrupting
                         if interrupting ...
                             interrupt(TIM0_INT_VECTOR)
                          end if
                      end if
                  end if

                  if ext mem 1 callback and not sleeping ...
                      if ext_int_cb_1()
                          check if interrupting
                          if interrupting ...
                              interrupt(EXT1_INT_VECTOR)
                          end if
                      end if
                  end if

                  if timer 1 callback and not sleeping ...
                      if timer_cb() ...
                         check if interrupting
                         if interrupting ...
                             interrupt(TIM1_INT_VECTOR)
                          end if
                      end if
                  end if

                  if serial interface callback and not sleeping ...
                      if serial_cb() ...
                          check if interrupting
```

```
                        if interrupting ...
                            interrupt(SER_INT_VECTOR)
                        end if
                    end if
                end if
            end if
          }


          if timer callback ...
              timer()
          end if

        // Execution instruction
          execute() {

             fetch next opcode from code_mem[pc]
            lookup instruction function and decode data from decode_table[opcode]
            fetch any argument bytes from code_mem[pc++]

            if verbose...
               print verbose output

              execute instruction *func()
        }

          <process breakpoints>
      end while
  }
```

The above pseudo-code is a rough outline only. The main structure is a while loop that continues until a breakpoint (such as reaching a break address or executing a specified number of cycles etc.). Interrupts are processed first, with a hierarchy of inspection, as dictated by the 8051 functionality. Peripherals are then processed, with the timers being the only internal peripherals, before the `execute()` function is called to decode and execute the next opcode. After execution, any new breakpoint state is processed to flag for the next loop iteration.

Key Model State

The list below shows some of the key model state used in the model.

- `code_mem[], ext_ram[], int_ram[]` : internal 8 bit memories
- `acc, sp, pc, b` : variables containing the specific 8051 register state
- `*r` : pointer to the currently active bank of r registers
- `dptr, tcon, tmod, tl0, tl1, th0, th1, scon, sbuf, pcon, p0, p1, p2, p3, psw, ie, ip` : Variables holding the standard function register state.
- `cycle_count` : counter of execution cycles.
- `int_level` : current model interrupt level
- `decode_table[]` : table of instruction decode information. See the 'Decode Table' section below for more details.

Decode Table

At the heart of the execution of the model is a decode table used for quick lookup of decode information for a given instruction's opcode. The decode table consists of 256 entries with the following structure type:

```
typedef struct {
    pfunc_t  func;             // Pointer to execution function
    char*    instr_name;       // Instruction name
    int      instr_size;       // Instruction size in bytes
    int      clk_cycles;       // Number of clock cycles for execution
    int      addr_mode_op1;    // Addressing mode operand 1
    int      addr_mode_op2;    // Addressing mode operand 2
} DecodeData_t, *pDecodeData_t;
```

It is a constant table, and held in the global `'decode_table'` variable, initialised at compilation. The `instr_name` field is a string for verbose/debug purposes, whilst the instruction size defines the number of bytes used by the instruction. The timing model information for instruction is defined in the `clk_cycles` field, and is taken directly from [1]. The `addr_mode_op1` and `addr_mode_op2` fields define the addressing mode operand's 1 and 2 (or just operand 1, or both don't care, as appropriate for the instruction. These can be one of the following:

- `ACC` : accumulator
- `REG[0-7]` : registers 0 to 7
- `DPTR` : data pointer
- `REL` : relative
- `CODE` : code address
- `CC` : carry flag
- `BIT` : bit address
- `NBIT` : bit address
- `BREG` : register B
- `IMM16` : immediate, 16 bits
- `PC` : program counter

All instructions have the same structure type in the decode table, but fields not relevant for some instructions are set to 'don't care' definitions. An example entry is show below:

```
{DIV,  "DIV   AB              ", 1, 8, ACC,  BREG}
```

During execution, the opcode is used to index into the decode table and the entry retrieved. If further bytes are required, as indicated by "`instr_size`", then these are fetched.

A new structure type is populated with this information, and has the following definition:

```
struct DecodeStruct {
    uint8_t             opcode;  // Instruction opcode
    uint8_t             arg0;    // First byte (where applicable)
    uint8_t             arg1;    // Second byte (where applicable)
    pDecodeData_t       decode;  // Ptr to entry in decode table for opcode
}
```

The `opcode` field is set to the raw fetch opcode byte, whilst `arg0` and `arg1` are populated with any subsequent opcode bytes. The decode table entry is pointed to by `decode`, and the function pointed to by "`func`" is called, passing in the `DecodeStruct` variable just constructed.

Instruction Functions

The actual instruction execution functions are defined in the source file `inst.c`, and all have a similar basic format. An example is shown below for the divide instruction.

```
void DIV (pDecode_t d) {
    int op1, op2, tmp;

    // Update the cycle count
    cycle_count += d->decode->clk_cycles;

    fetch_arg(d->decode->addr_mode_op1, &op1, d->arg0, d->arg1);
    fetch_arg(d->decode->addr_mode_op2, &op2, d->arg0, d->arg1);

    SET_PSW_OV(psw, op2 == 0 ? 1 : 0);

    // Clear carry
    SET_PSW_CY(psw, 0);

    if (op2) {
        tmp = op1 % op2;
        op1 = op1 / op2;
    }

    write_arg(d->decode->addr_mode_op1, op1, d->arg0);
    write_arg(d->decode->addr_mode_op2, tmp, d->arg0);

    pc += d->decode->instr_size;
}
```

The function is passed in the decode table entry and adds the `clk_cycles` count to the master `cycle_count` time state. A local function "`fetch_arg()`" is used to get the values indicated by `op1` and `op2` operands, based on the passed in addressing mode fields from the decode table entry.

Testing

A set of assembler programs were developed and are provided for execution on `cpu8051`, that execute a range of self-tests to verify the model. These tests are all directed tests, but cover nearly all aspects of the model including all instructions and all exceptions. Each program lives in a solitary directory under the directory `test/<category>/` and each sub-directory has a single source file, '`<category>.asm`'. These tests are self-checking and return a value `0x99` in internal memory location `0x7f` if the test passes, or `0xbb` if it fails (if the program never terminates cleanly, then this value is undefined—but is unlikely to be the pass value).

Executing Tests

The tests are all run via a 'runtest.sh' script that lives in the test/ directory. Changing directory to 'test/' and running the 'runtest' script will execute all the tests, giving a pass/fail criteria for each. An easier way to execute the tests is to use the makefile. When building code, a command 'make test' will get the build up-to-date, and then run the test script. The tail end of the output should be something like that shown below:

```
.
.
jz      :  PASS
push    :  PASS
mov     :  PASS
movc    :  PASS
xch     :  PASS
reti    :  PASS
```

There are currently 27 tests that cover all instructions and an EXT0 exception test. Note that some of the tests (e.g. rl/) actually cover multiple instructions: A full list is given below.

- acall includes testing of lcall, ret
- rl includes testing of rlc, rr, rrc
- jb includes testing of jbc, jnb
- jc includes testing of jnc
- jz includes testing of jnz
- jmp includes testing of sjmp, ljmp, ajmp
- push includes testing of pop
- movc includes testing of movx
- xch includes testing of xchd, swap

All the others tests are for the single instructions specified by the name of the test directory:

- add
- addc
- subb
- anl
- orl
- xrl
- cjne
- clr
- setb
- cpl
- da
- dec
- inc
- div
- mul
- djnz
- mov
- reti

Further Reading

[1] C8051F52x/F53x Data Sheet, Rev 1.4, Silicon Laboratories, April 2012
[2] KEIL/ARM 8051 instruction manual website,
www.keil.com/support/man/docs/is51/is51_instructions.htm
[3] 8502.com tutorial: http://www.8052.com/tut8051
[4] “Using as, the GNU Assembler”, version 2.19.51, 2009