Detailed explanation of i8051 ISS and analysis of variables overflow and different solutions

- Pietro Alfonsi
- Edgardo Vittoria
- Giannantonio Frittella
- Andrea Rofini

8051 Instruction Set Simulator

General description

8051 Instruction Set Simulator created by Dalton Project allows to simulate simple programs for the microcontroller providing, furthermore, stats and measures of performance. Source code is divided in:

- main.cc : where the process is launched
- i8051.h : header file that describes the class which implements the simulator procedures
- i8051.cc : actual implementation of the simulation procedures

How the process is launched

In the main.cc file two main actions are performed:

- check if number of command line arguments is correct
- launch the simulation

The first one is:

```
1
2       if( argc > 3 ) {
3       cerr << "usage: 8051sim <HEX File> <Trace File>" << endl;
4       exit(0);
5       }
6       else {
7       signal(SIGINT, SigHandler);
8
```

If number of arguments is higher than three, the correct usage is shown as a output message and execution is ended. Line 7 handles the manual interruption (SIGINT or CTRL+c in a terminal) by calling the function SigHandler, which calls the Stop function of the i8051 class, terminating the execution:

```
1   void SigHandler(int signal)
2   {
3       i8051.Stop();
4   }
```

After that, the process is launched by calling Simulate(), a function of class i8051:

```
    i8051.Simulate(argc>=2 ? argv[1] : "out.hex",
                   argc==3 ? argv[2] : "output.txt");
        }
```

If two arguments are given, then the ISS will print a trace of the execution in a `out.hex` file, if a third argument is given then it will also be outputted in a .txt file.

We will see later in detail what happens after this call.

i8051 Header file

The i8051.h file defines the variables, flags and functions of the i8051 class, that performs the simulation.

First of all, some flags for output are defined:

```
    #define DEBUG
    #define DEBUG_PC
    #define DETAIL
    #define PORTS
```

`DEBUG` will output a trace of all instructions executed, with `DEBUG_PC` also the program counter will be outputted, while `DETAIL` adds extra information for every instruction. Finally, `PORTS` prints output whenever any value of the ports is changed.

Next, we can define the condition for which the program is terminated:

```
    //#define PROGRAM_COMPLETION ((unsigned char)RAM[P1] == 0x7F)
    #define PROGRAM_COMPLETION ( ((unsigned char)RAM[P0] == 0x0) && ((unsigned
    char)RAM[P1] == 0x0) && ((unsigned char)RAM[P2] == 0x0) && ((unsigned
    char)RAM[P3] == 0x0))
```

`PROGRAM_COMPLETION` defines a condition to be met for the execution to stop. The first (naive) one is when port P1 reaches value 127, which is not optimal and can lead to unwanted premature termination, if wanted output is indeed 127. The second one will stop the program when all ports have value 0, which is a more robust solution, as we can manually set the values to 0 when we want to stop the execution.

Then the physical size of RAM and ROM are defined:

```
    static const unsigned int RomSize = 65536;
    static const unsigned int RamSize = 128;
    static const unsigned int XRamSize = 65536;
```

Going ahead the name of all the instructions of the set are defined:

```
enum Opcode {
    ACALL, ADD1,  ADD2,  ADD3,  ADD4,  ADDC1, ADDC2, ADDC3, ADDC4,
    AJMP,  ANL1,  ANL2,  ANL3,  ANL4,  ANL5,  ANL6,  ANL7,  ANL8,
    CJNE1, CJNE2, CJNE3, CJNE4, CLR1,  CLR2,  CLR3,  CPL1,  CPL2,
    CPL3,  DA,    DEC1,  DEC2,  DEC3,  DEC4,  DIV,   DJNZ1, DJNZ2,
    INC1,  INC2,  INC3,  INC4,  INC5,  JB,    JBC,   JC,    JMP,
    JNB,   JNC,   JNZ,   JZ,    LCALL, LJMP,  MOV1,  MOV2,  MOV3,
    MOV4,  MOV5,  MOV6,  MOV7,  MOV8,  MOV9,  MOV10, MOV11, MOV12,
    MOV13, MOV14, MOV15, MOV16, MOV17, MOV18, MOVC1, MOVC2, MOVX1,
    MOVX2, MOVX3, MOVX4, MUL,   NOP,   ORL1,  ORL2,  ORL3,  ORL4,
    ORL5,  ORL6,  ORL7,  ORL8,  POP,   PUSH,  RET,   RETI,  RL,
    RLC,   RR,    RRC,   SETB1, SETB2, SJMP,  SUBB1, SUBB2, SUBB3,
    SUBB4, SWAP,  XCH1,  XCH2,  XCH3,  XCHD,  XRL1,  XRL2,  XRL3,
    XRL4,  XRL5,  XRL6
};
```

After all these global variables have been described, the class that defines the simulated microcontroller is illustrated, with the defininiton of `class I8051`. The first batch of variables are named after the Special Function Registers, a set of special purpose registers inside the microcontroller, like the accumulator, stack pointer ecc. (see more at https://www.cs.ucr.edu/~ehwang/courses/cs120b/8051.pdf, page 3):

```
private:
    const unsigned short ACC;
    const unsigned short PSW;
    const unsigned short B;
    const unsigned short SP;
    const unsigned short P0;
    const unsigned short P1;
    const unsigned short P2;
    const unsigned short P3;
    const unsigned short DPL;
    const unsigned short DPH;
```

Then we have variables for representation of physical hardware on board, as well as counters like Instruction Register and Program Counter:

```
private:
    char ROM[RomSize];
    char RAM[RamSize];
    char XRAM[XRamSize];
    unsigned short PC;
    unsigned long instrCount;
    unsigned long cycleCount;
    unsigned char IR;
    unsigned short tempDPTR;
    bool progEnd;
```

ProgEnd is used to control the end of the execution, with suited functions that we will see later.

Now all the methods are defined. First we have methods that manage the execution flow(start, stop ecc.):

```
public:
    bool Simulate(const char* fileName, const char* outFile = "output.txt");
    bool ProgramCompletion();
    void Stop();
    void Init8051();
    Opcode Decode(const unsigned char IR);
```

Next we have utility functions that handle bit manipulation:

```
 private:
    void SetBit(char &thisByte, unsigned char thisBit);
    void ClearBit(char &thisByte, unsigned char thisBit);
    unsigned char GetBit(char thisByte, unsigned char thisBit);
    unsigned char GetRegisterBank();
```

Also there is output formatting:

```
private:
    void PrintHex(unsigned char byte, ostream* os = &cout);
    void PrintPorts();
```

And finally we have methods to handle the input HEX file and load data into memory:

```
 private:
    bool Hex2Short(const char* buf, unsigned &val);
    bool Load(const char* buf, unsigned char* rom, unsigned& prgSize);
    bool LoadHex(const char *filename);
```

All the variables and functions that are needed for simulation are defined. We will now see the implementation of these functions.

i8051.cc file

Constructor

First of all, the constructor is defined:

```
I8051::I8051() : InvalidData(11),
                 LineLength(80),
                 RecordTypeLength(3),
                 ACC(0x0160),
                 PSW(0x0150),
                 B(0x0170),
                 SP(0x0101),
                 P0(0x0100),
                 P1(0x0110),
                 P2(0x0120),
                 P3(0x0130),
                 DPL(0x0102),
                 DPH(0x0103),
                 PC(0),
                 instrCount(0),
                 cycleCount(0),
                 progEnd(false)
```

This will initialize specific values for the aforementioned variables.

Simulate

This is the main function in this file, defined as:

```
bool
I8051::Simulate(const char* inFile, const char* outFile)
```

This will execute the simulation using all the other functions previously defined, and is the only one called in the `main.cc` file:

```
i8051.Simulate(argc>=2 ? argv[1] : "out.hex",
               argc==3 ? argv[2] : "output.txt");
```

This method defines some variables:

```
// file handler
    ofstream os;

    clock_t begin, end;

```

```
6       unsigned short tempProduct = 0;
7       unsigned short jumpAddr;
8       short tempAdd;
9       unsigned char directAddr = 0;
10      unsigned char regNum;
11      unsigned char rotateBit;
12      unsigned char lowerNibble;
13      unsigned char tempACC;
14      char temp;
15      char popData;
16
17      bool carry3;
18      bool carry6;
19      bool carry7;
20      bool borrow3;
21      bool borrow6;
22      bool borrow7;
23
24  #ifdef PORTS
25      char lastP0 = 0xFF;
26      char lastP1 = 0xFF;
27      char lastP2 = 0xFF;
28      char lastP3 = 0xFF;
29  #endif
```

The most important ones are `begin` and `end` that will be used to calculate execution time and `lastPx` will hold the last used values for the four ports (P0, P1, P2 and P3), to be used to then print them. Other variables are used for holding data and perform calculations and conversions.

First action is to check the correct loading of the input HEX file with the use of LoadHex function (LoadHex function):

```
1   if( !LoadHex(inFile) ) {
2           return false;
3       }
```

Then the class is initialized and the clock count is started (Init8051 function):

```
1   Init8051();
2   begin = clock();
```

Next the `simulate` function will perform the following actions until the execution stops:

```
1   while(!progEnd) {
2           if( PC >= RomSize ) PC = 0;
3           IR = ROM[PC++];
4           instrCount++;
5           switch(Decode(IR)) {
```

Line 2 checks if Program Counter has exceeded ROM size, so it can be reset to 0, line 3 will load the next instruction into the Instruction Register and line 4 will increment the number of instructions executed. Line 5 will decode the loaded instruction (Decode function).

All the instructions in the Set are manually implemented as cases in the switch statement previously mentioned (Ex. MOV1, you can check the code for all of them):

```
1   // A <- Rn
2           case MOV1:
3               #ifdef DEBUG
4               os << "MOV 1" << endl;
5               #ifdef DETAIL
6               os << "\t" << "A <- R" << (int)(IR & 0x07) << endl;
7               #endif
8               #endif
9               regNum = IR & 0x07;
10              RAM[ACC] = RAM[GetRegisterBank()+regNum];
11              cycleCount += 12;
12              break;
```

Eventually, the values in the ports are outputted and the stop condition is checked (PrintPorts function, ProgramCompletion function):

```
1   #ifdef PORTS
2           // Set LastP* to last used value
3           if( (lastP0 != RAM[P0]) || (lastP1 != RAM[P1]) ||
4               (lastP2 != RAM[P2]) || (lastP3 != RAM[P3]) ) {
5               lastP0 = RAM[P0];
6               lastP1 = RAM[P1];
7               lastP2 = RAM[P2];
8               lastP3 = RAM[P3];
9               PrintPorts();
10          }
11          #endif
12
13          // Checks if the condition of interruption of the program is met
14          if( ProgramCompletion() ) progEnd = true;
```

The if statement will print the output if values of ports has changed and set the `lastPx` variables to the new value.

Finally, simulation metrics are calculated and outputted:

```
1       // clock tick count at time of completion
2       end = clock();
3       // calculate execution time as end - begin
4       double time = (double)(end - begin)/(double)(CLOCKS_PER_SEC);
5       // Print metrics of execution
6       cout << setiosflags(ios::left);
7       cout << endl;
8       cout << setw(45) << "Instructions Executed: "
9            << instrCount << endl;
10      cout << setw(45) << "Execution Time(seconds): "
11           << time << endl;
12      cout << setw(45) << "Average Instructions/second: "
13           << instrCount/time << endl;
14      cout << endl;
15      cout << setw(45) << "Clock Cycles Required for 8051: "
16           << cycleCount << endl;
17      cout << setw(45) << "Execution Time for 8051(12 MHz)(seconds): "
18           << cycleCount/12e6 << endl;
19      cout << setw(45) << "Average Instructions/second for 8051: "
20           << instrCount/(cycleCount/12e6) << endl;
21      cout << endl;
22      return true;
```

Line 2 calculates the time at the end of execution, to be used in line 4 to get time of execution as end - begin. The C++ macro CLOCKS_PER_SEC represents the number of clock ticks per second, so dividing a count of clock ticks (`end - begin`) by this expression yields the number of seconds.

The simulation is now finished.

• Decode

The Decode function just maps an hexadecimal value passed in memory from the input HEX file to the corresponding instruction in the set (see code for all the statements):

```
1   Opcode
2   I8051::Decode(const unsigned char IR)
3   {
4       switch( IR ) {
5       case 0x25:
6           return ADD2;
7       case 0x24:
8           return ADD4;
9       ...
```

Init8051

This function initializes the values for the Special Function Registers:

```
void
I8051::Init8051() {
    for(unsigned int i=0;i<RamSize;i++) RAM[i] = 0x00;
    // Accumulator
    RAM[ACC] = 0x00;
    // Program Status Word
    RAM[PSW] = 0x00;
    // Ports
    RAM[P0] = 0xFF;
    RAM[P1] = 0xFF;
    RAM[P2] = 0xFF;
    RAM[P3] = 0xFF;
    // B Register to multiply and divide
    RAM[B] = 0x00;
    // Stack Pointer
    RAM[SP] = 0x07;
    // Data Pointers
    RAM[DPH] = 0x00;
    RAM[DPL] = 0x00;
}
```

ProgramCompletion

Checks the stop condition if defined in header file:

```
bool
I8051::ProgramCompletion()
{
    #ifdef PROGRAM_COMPLETION
    if( PROGRAM_COMPLETION ) return true;
    #endif
    return false;
    #ifdef PORTS
    cout << "P0\tP1\tP2\tP3" << endl;
    PrintPorts();
    #endif
}
```

PrintHex

This function is responsible for printing in hexadecimal format the byte passed as parameter:

```
1   void
2   I8051::PrintHex(unsigned char byte, ostream* os)
3   {
4       // Takes first 4 bit of byte (AND 11110000)
5       unsigned char upperNibble = (byte & 0xF0) >> 4;
6       // Takes last 4 bit of byte (AND 00001111)
7       unsigned char lowerNibble = byte & 0x0F;
8
9       // Checks if a carry is necessary
10      upperNibble = (upperNibble > 9) ? 'A'+(upperNibble-10) : '0'+upperNibble;
11      lowerNibble = (lowerNibble > 9) ? 'A'+(lowerNibble-10) : '0'+lowerNibble;
12      // Prints the value
13      *os << "0x" << upperNibble << lowerNibble;
14  }
```

Line 5 and 7 perform bit-to-bit AND in order to divide the input byte in two blocks of 4 bits (upper and lower). Then line 10 and 11 convert each block of bits in hexadecimal, performing a carry if needed. Finally it outputs the value as 0x *Value* to the `os` variable.

PrintPorts

PrintPorts outputs the current values in P0, P1, P2 and P3, using the PrintHex function:

```
1   void
2   I8051::PrintPorts()
3   {
4       PrintHex(RAM[P0]); cout << "\t";
5       PrintHex(RAM[P1]); cout << "\t";
6       PrintHex(RAM[P2]); cout << "\t";
7       PrintHex(RAM[P3]); cout << "\n";
8   }
```

LoadHex

This function takes the input HEX file and loads its content into memory:

```
1   bool
2   I8051::LoadHex(const char *filename)
3   {
4       unsigned prgSize = 0;
5       char buf[256];
6       ifstream ifs(filename);
```

```
7       memset(ROM, 0, sizeof(char)*RomSize);
8       if( ifs.bad() ) {
9           cerr << "Error: invalid HEX file." << endl;
10          return false;
11      }
12      while( !ifs.eof() ) {
13          ifs.getline(buf, sizeof(buf));
14          if( Load(buf, (unsigned char*)ROM, prgSize) ) break;
15          }
16      return true;
17  }
```

Lines 4 to 7 initialize variables including the ROM, which is filled with zeros. Line 8 checks for errors in the input file. If there are none, line 12 gets every line of the file and loads it into memory with the Load function, line-by-line.

Load

This method checks if input line is a correct Hex format and loads its data field into ROM:

```
1   bool
2   I8051::Load(const char* buf, unsigned char* rom, unsigned& prgSize)
3   {
4       unsigned temp;
5       /* len = record length
6          base = data address
7          type = data type */
8       unsigned len, base, type;
9       unsigned char checksum = 0;
10      char hex[16];
```

First of all, some variables are defined: `len`, `base` and `type` refer to the *Byte Count*, *Address* and *Record Type* fields in the Hex format (see full format at https://en.wikipedia.org/wiki/Intel_HEX).

Then, it will check if the first value is `:`, as in the HEX standard:

```
1   if( buf[0] != ':' ) {
2
3           cerr << "Error: hex file error." << endl;
4           return false;
5       }
```

Next it controls if the *Byte Count* field is present (mandatory, 2 bytes) using the Hex2Short function:

```
1   hex[0] = buf[1];
2   hex[1] = buf[2];
3   hex[2] = 0; // termination character
4   if( !Hex2Short(hex, len) ) return false;
```

Same for *Address* (4 bytes):

```
1   hex[0] = buf[3];
2   hex[1] = buf[4];
3   hex[2] = buf[5];
4   hex[3] = buf[6];
5   hex[4] = 0;
6   if( !Hex2Short(hex, base) ) return false;
```

And for *Record Type* (2 bytes):

```
1   hex[0] = buf[7];
2   hex[1] = buf[8];
3   hex[2] = 0;
4   if( !Hex2Short(hex, type) ) return false;
```

In all three snippets, the string is manually terminated with 0, the ASCII code for string termination.

At this point, `len`, `base` and `type` contain the values of the related fields and code goes on with:

```
1   if( type == 1 ) return true;
2   if ( (base+len) > (prgSize) ) {
3           (prgSize) = base + len + 2;
4       }
5   if( (prgSize) >= RomSize || (base+len) >= RomSize ) {
6           printf("program too large\n");
7           exit(1);
8       }
```

Line 1 checks for an End Of File record, which is value 1 in HEX format. If not, size of program read thus far is calculated and checked if greater than ROM size.

Then bytes in *Data* field are loaded into ROM:

```
1   for(unsigned i=0; i<len; i++) {
2           hex[0] = buf[ 9 + i * 2];
3           hex[1] = buf[10 + i * 2];
4       if( !Hex2Short(hex, temp) ) return false;
5           rom[base + i] = (unsigned char)temp;
6       }
```

The number of data bytes (and the position of them) is identified by the `len` variable. If a correct hexadecimal value is present in each position in the loop, it is put into the `temp` variable and then loaded into ROM (with respect of the offset defined by `base`).

Finally the checksum is calculated and confronted with the value in the last two bytes of the HEX line:

```
1   for(unsigned i=0; buf[i * 2 + 1] && buf[i * 2 + 2]; i++) {
2           hex[0] = buf[i * 2 + 1];
3           hex[1] = buf[i * 2 + 2];
4           unsigned temp;
5           if( !Hex2Short(hex, temp) ) return false;
6               checksum += (unsigned char)temp;
7           }
8   if( checksum != 0 ) {
9       cerr << "Error: checksum failed." << endl;
10      return false;
11  }
12  return false;
```

Checksum is defined as two's complement of the least significant byte of the sum of all the values in the HEX record, and can be computed as the additive inverse, summing all the byte values (including checksum bytes) and verifying that the sum is zero (line 10)

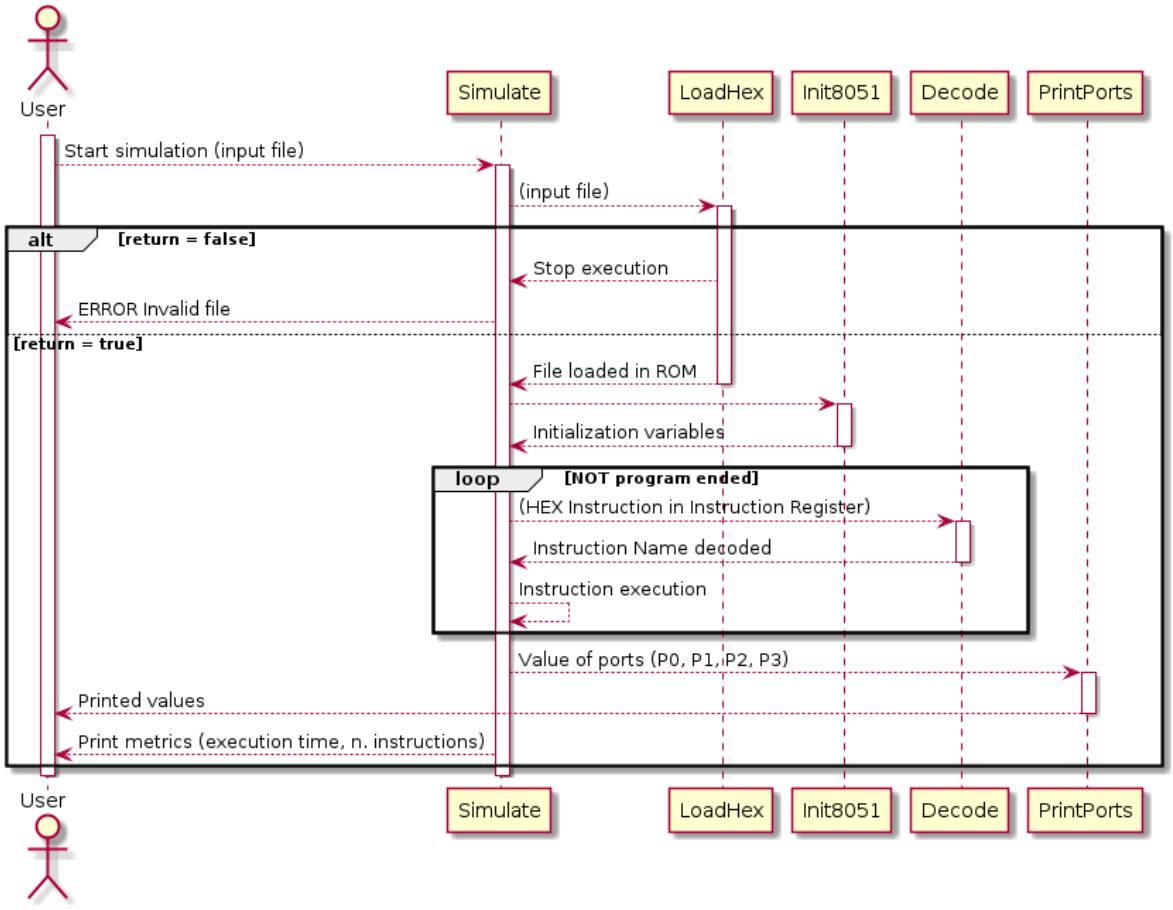
• Hex2Short

This function takes an array of characters and a reference variable as input and checks if a correct hexadecimal value is represented in the array and puts it into the referenced input (sscanf reference or type `man sscanf` in a Linux terminal).

```
bool
I8051::Hex2Short(const char* buf, unsigned &val)
{
    int i;
    if( sscanf(buf, "%x", &i) != 1 ) {
        cerr << "Error: hex file error." << endl;
        return false;
    }
    val = i;
    return true;
}
```

Execution flow

Here is represented the flow of execution as a sequence diagram, with calls between the functions that compose the i8051 class:



Variables overflow

We saw that the simulation prints some metrics related to execution time and number of instructions on the machine that hosts the simulation and, also, metrics that calculate the time and number of instructions on the simulated 8051 microcontroller. In the header file, these variables (`instrCount` and `cycleCount`) are defined as `unsigned long` , meaning that 32 bits are reserved for them. It may happen, if the

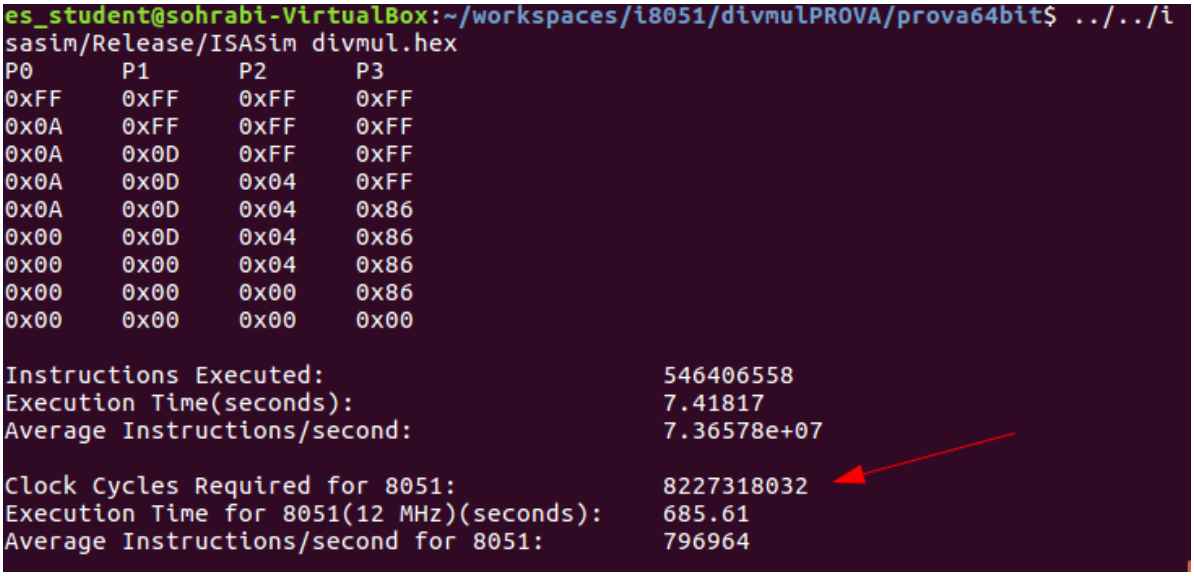
program is heavy, that these variables overflow the 2^32 value limit. We observed that the count will restart from 0 every time the limit is reached, invalidating all the other metrics derived from the instruction count and cycle count.

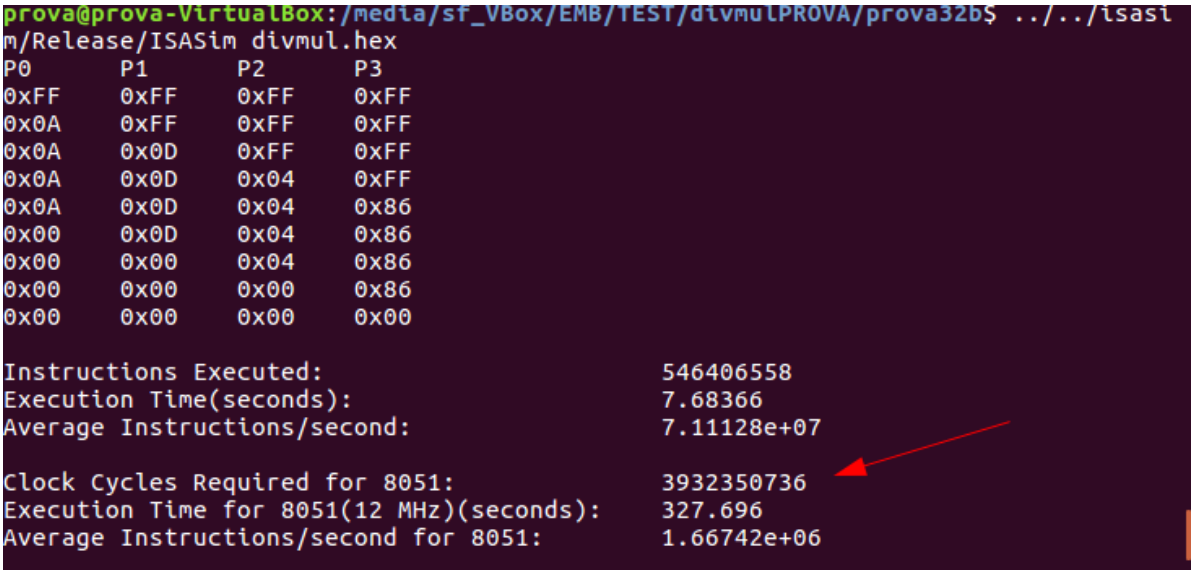
We provoked this behavior by executing the *divmul* program thousands of times in a cycle, doubling every time the number of iterations (to immediately check if the values are not falsified by potential overflows), until we observed overflow.

We found three solution for this problem:

- **Use a 64-bit system**

We observed that the overflow happens only in 32-bit systems. If the host machine supports 64-bit variables, it will automatically adjust the size to display the right value, even if the source code is compiled to be 32-bit.





As you can see, in the first image (64-bit system), the value for *Clock Cycles Required for 8051* is more than 8 billions, way over the 4.3 (sort of) billions limit given by the 32 bits. In the second image (32-bit system), the same value is bogus, and is actually the correct value minus the 32 bit limit. This error affects the other two related values as well.

Use 64-bit variables

The second solution is to change the declaration of `instrCount` and `cycleCount` to `unsigned long long`, a 64-bit variable. This will fix the issue even on 32-bit systems, where there will be two memory allocations per-variable to compensate the limits of the system. You can see that the results are the same as the 64-bit system:

```
prova@prova-VirtualBox:/media/sf_VBox/EMB/TEST/divmulPROVA/prova32b$ ../../isasi
m/Release/ISASim divmul.hex
P0      P1      P2      P3
0xFF    0xFF    0xFF    0xFF
0x0A    0xFF    0xFF    0xFF
0x0A    0x0D    0xFF    0xFF
0x0A    0x0D    0x04    0xFF
0x0A    0x0D    0x04    0x86
0x00    0x0D    0x04    0x86
0x00    0x00    0x04    0x86
0x00    0x00    0x00    0x86
0x00    0x00    0x00    0x00

Instructions Executed:                      546406558
Execution Time(seconds):                    7.05793
Average Instructions/second:                7.74174e+07

Clock Cycles Required for 8051:             8227318032
Execution Time for 8051(12 MHz)(seconds):   685.61
Average Instructions/second for 8051:       796964
```

Fix in code

The last solution is to fix the source code to keep track of the value overflow, if it takes place. We will implement this by checking if the value has re-started from 0 at every cycle, comparing it with the value in the previous cycle. If the new value is smaller (it works for count of instruction, which is incremented by 1 every cycle and count of cycles, incremented by 16, 32 or 64 depending on the instruction executed), a variable containing the number of times the overflow has happened is incremented by 1.

We define these variables in *i8051.h* :

```
1   unsigned short overflowInst;
2   unsigned long lastInstrCount;
3   unsigned short overflowCycle;
4   unsigned long lastCycleCount;
5   unsigned long overflowExecTime;
```

Every new iteration of the simulation, we save the last value of instruction count and cycle count (line 4 and 6):

```
while(!progEnd) {
        if( PC >= RomSize ) PC = 0;
        IR = ROM[PC++];
        lastInstrCount = instrCount;
        instrCount++;
        lastCycleCount = cycleCount;
        switch(Decode(IR)) {
```

So we can check, at the end of the single iteration, if the value has overflown.

If it has, we increment the overflow count:

```
if (lastCycleCount > cycleCount) {
    overflowCycle +=1;
        }

if (lastInstrCount > instrCount) {
    overflowInst +=1;
        }
```

We can now print the values in a formatted way:

```
cout << setw(45) << "Instructions Executed: "
     << instrCount << "+" << overflowInst << "*2^32" <<  endl;
...
cout << setw(45) << "Clock Cycles Required for 8051: "
     << cycleCount << "+" << overflowCycle << "*2^32" << endl;
```

For the other metrics, we have to adjust the way they are calculated. For the *Average instructions per second* we obtain:

```
double avgInstr = ((pow(2,28)/time)*16*overflowInst) + (instrCount/time);
...
cout << setw(45) << "Average Instructions/second: "
     << avgInstr << endl;
```

`instrCount/time` is the original value, but it doesn't count any possible overflow. We account for that, by multiplying 2^32 divided by the value of *execution time* (written as 2^28 first and the multiplied by 2^4 to avoid the risk of having other overflows) multiplied the count of overflows of the *Instruction count* .

For the metric of *Execution time for 8051* we have to adjust for cycle count overflows:

```
1   overflowExecTime  = (pow(2,31)/12e6)* 2 * overflowCycle;
2   double timeSim = (cycleCount/12e6) + overflowExecTime;
3   ...
4   cout << setw(45) << "Execution Time for 8051(12 MHz)(seconds): "
5        << timeSim  << endl;
```

Line 1 applies the same logic we saw above to compensate for the number of time the value has overflown. Simulated time, then, is calculated with the value obtained with the overflow values.

Finally, *Average Instructions per second for 8051* is adjusted by keeping track of overflow in number of instructions and in number of cycles:

```
1   double avgInstrSim = instrCount/(timeSim) + (pow(2,28)/(timeSim))*16 *
    overflowInst;
2   ...
3   cout << setw(45) << "Average Instructions/second for 8051: "
4        << avgInstrSim << endl;
```

These are the results (second image), compared with the ones of a 64-bit system (first image) using the original source code:

```
es_student@sohrabi-VirtualBox:~/workspaces/i8051/divmulPROVA/prova64bit$ ../../isasim/Release/ISASim divmul.hex
P0      P1      P2      P3
0xFF    0xFF    0xFF    0xFF
0x0A    0xFF    0xFF    0xFF
0x0A    0x0D    0xFF    0xFF
0x0A    0x0D    0x04    0xFF
0x0A    0x0D    0x04    0x86
0x00    0x0D    0x04    0x86
0x00    0x00    0x04    0x86
0x00    0x00    0x00    0x86
0x00    0x00    0x00    0x00

Instructions Executed:                      2185625380
Execution Time(seconds):                    29.8524
Average Instructions/second:                7.32144e+07

Clock Cycles Required for 8051:             32909256828
Execution Time for 8051(12 MHz)(seconds):   2742.44
Average Instructions/second for 8051:       796964
```

```
prova@prova-VirtualBox:/media/sf_VBox/EMB/TEST/divmulPROVA/prova32b$ ../../isasim/Release/ISASim divmul.hex
P0      P1      P2      P3
0xFF    0xFF    0xFF    0xFF
0x0A    0xFF    0xFF    0xFF
0x0A    0x0D    0xFF    0xFF
0x0A    0x0D    0x04    0xFF
0x0A    0x0D    0x04    0x86
0x00    0x0D    0x04    0x86
0x00    0x00    0x04    0x86
0x00    0x00    0x00    0x86
0x00    0x00    0x00    0x00

Instructions Executed:                      2185625380+0*2^32
Execution Time(seconds):                    44.4584
Average Instructions/second:                4.91611e+07

Clock Cycles Required for 8051:             2844485756+ 7*2^32
Execution Time for 8051(12 MHz)(seconds):   2742.04
Average Instructions/second for 8051:       797080
```

While some values are very similar (number of instructions, execution time for 8051 and average instructions), other are different. *Execution time* , for example, is very dependent on the host machine, number of background processes and it compensates, also, the calculations we added and it affects also the *Average Instructions per second* metric.