A SystemC model of a Bitcoin Miner

Zhongqi Cheng, Rainer Doemer

Technical Report CECS-16-04
September 7, 2016

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
(949) 824-8919

zhongqc@uci.edu
http://www.cecs.uci.edu
Abstract

Bitcoin is a new digital asset, and for years people are struggling to accelerate Bitcoin miners for more profit. In this document, we present an example of a parallel Bitcoin miner specification model as a case study for SystemC based Electronic System Level design. We recode a C++ reference code of a Bitcoin miner, implement a central controller and utilize polling to synchronize between the parallel computation blocks. The experiment demonstrates that SystemC is a powerful language for system design, and the results show the speed-up ratio grows linearly with the number of parallel worker modules, which in turn suggests our design to be effective on accelerating the Bitcoin miner.
## Contents

1 Introduction \hspace{10em} 1

2 Overview of Bitcoin Miner Algorithm \hspace{10em} 1

3 System Level Modeling of Bitcoin Miner \hspace{10em} 2
   3.1 Sequential Bitcoin Miner Design \hspace{10em} 2
   3.2 Parallel Bitcoin Miner Design \hspace{10em} 2
      3.2.1 Expected speed-up ratio of parallelization \hspace{10em} 2
      3.2.2 Implementation of Parallel Bitcoin Miner \hspace{10em} 3

4 Experiments and Results \hspace{10em} 4

5 Conclusion and Future Work \hspace{10em} 4

References \hspace{10em} 4

A Appendix \hspace{10em} 6
   A.1 Source Code of Parallel Bitcoin Miner in SystemC \hspace{10em} 6
   A.2 Makefile \hspace{10em} 28
List of Figures

1. Bitcoin miner flowchart .................................................. 2
2. scan and hash for sequential implementation .......................... 2
3. Expected speed-up ratio with increasing level of parallelism .......... 3
4. Structure of parallel Bitcoin miner ........................................ 3
5. Structure of main control block with two scan and hash blocks .... 3
6. Structure of scan and hash for parallel implementation .......... 3
List of Tables

1  Example of a Bitcoin block header ................................................................. 2
2  Performance with increasing Number of Workers ........................................... 4
3  Relationships between the Total Number of Communications and Computations 4
Abstract

Bitcoin is a new digital asset, and for years people are struggling to accelerate Bitcoin miners for more profit. In this document, we present an example of a parallel Bitcoin miner specification model as a case study for SystemC based Electronic System Level design. We recode a C++ reference code of a Bit- coin miner, implement a central controller and utilize polling to synchronize between the parallel computation blocks. The experiment demonstrates that SystemC is a powerful language for system design, and the results show the speed-up ratio grows linearly with the number of parallel worker modules, which in turn suggests our design to be effective on accelerating the Bitcoin miner.

1 Introduction

Bitcoin is a new peer-to-peer digital asset and a decentralized payment system introduced by Satoshi Nakamoto in 2009 [5]. New bitcoins are created from Bitcoin blocks, which must contain some transactions and a so-called proof-of-work. The proof-of-work requires users to find a nonce, such that when the block content is hashed using the SHA256 [7] along with the number, the result is numerically smaller than the network’s difficulty target.

The proof-of-work is easy for any node in the network to verify, but extremely time-consuming to generate, as for a secure cryptographic hash, users must try many different nonce values (usually the sequence of tested values is 0, 1, 2, 3, ...) before meeting the difficulty target [2]. As such, with a higher hash-rate, users are expected to earn more bitcoins.

In this paper, we propose a SystemC [6] based parallel Bitcoin miner model. Our Bitcoin miner is based on a C++ reference code [4], and using polling to synchronize between the parallel computation blocks. Timing for communications and computations are studied under different conditions using the single-thread SystemC 2.3.1-Accellera simulator.

2 Overview of Bitcoin Miner Algorithm

Mining today takes on two forms, solo mining and pooled mining [8]. The proposed Bitcoin miner mainly focuses on solo mining and is based on the reference implementation CPUminer [4]. The Bitcoin miner basically performs the following three steps, as illustrated in Figure[1]

I. get_work first requests a Bitcoin block template from the network or a local server, and based on that, it builds a block header which contains some information describing the Bitcoin block, as described in [1]. Then the 80-byte block header is sent to the computational part, called scan_and_hash in our project, along with a target threshold.

II. scan_and_hash iterates the nonce (a 4-byte value inside the block header) over a certain range, and repeatedly performs SHA256 on the whole block header to generate a corresponding hash. When the hash value is below the desired difficulty target, the proof-of-work is done and the successful block header is transferred to the submit_work module.

III. submit_work combines together the block header
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x123...fff</td>
<td>First 72 bytes which are fixed</td>
</tr>
<tr>
<td>0x30c31b18</td>
<td>nBit, encoded version of target</td>
</tr>
<tr>
<td>0x00000000</td>
<td>Initial nonce value</td>
</tr>
</tbody>
</table>

Table 1: Example of a Bitcoin block header

and some other information. Then it broadcasts the block to the network, or sends it back to the local server.

Note that the main computation of Bitcoin mining is performed in scan_and_hash. In other words, scan_and_hash carries by far the highest computational work.

3 System Level Modeling of Bitcoin Miner

In this section, we first introduce the implementation of a sequential Bitcoin miner, which serves both as a simple model of mining and as a reference for performance analysis. Then we propose our high level model of parallel Bitcoin miner in the SystemC system-level description language, and estimate its expected speed-up ratio through simulation.

3.1 Sequential Bitcoin Miner Design

Figure 2 demonstrates the mining steps of a sequential scan_and_hash block. As illustrated, there is a big while loop iterating the nonce and performing SHA256. For instance, assume that the block header passed to the scan_and_hash block is shown in Table 3. Then the goal of the scan_and_hash would be to find a nonce starting from the initial value (0x00000000 here) such that the hash of the 80-byte block header is less than the difficulty target. This target is derived from the nBit, and more details are explained in [3]. In this example, the target is $0x1bc330 \times 256^{0x18−3}$. We should note that the sequential miner is very time consuming.

![Figure 2: scan_and_hash for sequential implementation](image)

3.2 Parallel Bitcoin Miner Design

Based on the observation that the scan_and_hash function is the most complex and time-consuming part, we decide to optimize it first. Parallelization is the most intuitive approach to speed up the computation.

The parallel structure we propose is that to duplicate the sequential one and run them in parallel, with each assigned a disjoint nonce range, and synchronized through a central controller named main_control_block, as shown in Figure 4.

3.2.1 Expected speed-up ratio of parallelization

To ease qualitative analysis of the speed-up ratio, it is reasonable to assume that there is a one-to-one mapping between the nonce and the resulting hash, and thus we can easily calculate the probability of finding a successful nonce. In the above example in Table 3, this probability is roughly $2^{(0x18−3)\times 8+24−256} = 5.4^{-20}$. This is to say that if we can calculate 1 billion nonces per second, it will take around 588 years on average to find a successful block header. Furthermore, note that with SHA256 as the secure cryptographic hash function, the successful nonces have a uniform distribution across the entire range.

Based on these two points, the speed-up ratio for our parallel design is expected to linearly grow with the increasing of parallelism, as illustrated through simulation in Figure 3. For instance, with the parallelism
of two, such that one scan_and_hash block iterates from 0x0000_0000 to 0x7fff_ffff and the other one from 0x8000_0000 to 0xffff_ffff, this two-worker Bitcoin miner is expected to spend half as much time as that of the sequential one to find a successful nonce.

Figure 3: Expected speed-up ratio with increasing level of parallelism

3.2.2 Implementation of Parallel Bitcoin Miner

In this section, we propose the implementation of our parallel Bitcoin miner. The flowchart is shown in Figure 4, which is mainly different from the sequential implementation in the place of an additional main_control_block.

Figure 5 shows the main_control_block, and for a simple illustrative purpose, this figure only contains two scan_and_hash blocks. The main_control_block serves to synchronize the parallel scan_and_hash blocks, such that when one of them has found a successful block header, others can stop as they do not need to work on the current proof-of-work any longer. And in the next few clock cycles, all the scan_and_hash blocks shall restart with a new block header. For now, the synchronization is implemented through polling.

Here we describe in more details the algorithm for the main_control_block (controller). First, the controller requests a 80-byte block header from the outside. Next, FOUND is sent to all the scan_and_hash blocks (workers), which indicates that the workers shall start (restart, if the worker is already running) on a new proof-of-work, and then block header is broadcasted. Afterwards, synchronization is performed within a polling loop. The controller begins to wait for flags from workers. If one or more FOUND is received, which means a successful nonce is found, then the controller sends the nonce out, and restarts itself. If the flags are all NOT FOUND, then NOT FOUND flag is broadcasted back, indicating that all workers shall keep hashing on the current proof-of-work.

Correspondingly, we also modify the algorithm for the scan_and_hash block. As shown in Figure 6, this block consists of a single loop. At the very beginning of the loop, a flag is received from the controller. If FOUND, then it reads in a block header, and increments the nonce by one and performs SHA256 for the proof-of-work. If NOT FOUND, the module simply skips the reading of the block header, and leaves the computation steps unchanged.
4 Experiments and Results

We have implemented the proposed Bitcoin miner in SystemC 2.3.1 and simulated the module under an Intel E3-1240 processor, with a main frequency of 3.40 GHz. The simulator we choose is the SystemC 2.3.1-Accellera.

Our main concern for now is the speed-up ratio of the proposed Bitcoin miner, and the analysis is performed in two parts.

In the first part, we focus on the timing of synchronization between the scan_and_hash blocks, and it is easily obtained by appending 

\[ \text{wait}(1, \text{SC\_NS}) \]

instructions to the communication behaviors, specifically, the write instructions. The final simulator time is recorded as \( N_{\text{comm}} \), which indicates the number of communications.

In the second part, the timing cost for computation is studied. This is done in a similar way by appending 

\[ \text{wait}(1, \text{SC\_NS}) \]

instructions to the computation behaviors. Note that in this step, the wait instructions for communications are removed, so the simulation time only increases with the computation behaviors. Similarly, the final simulator time is recorded as \( N_{\text{comp}} \), which indicates the number of computations.

Although we have set the cost of communication and computation both to one nano second, they are quite different in reality, and the reason we set them to the same value is merely to check out the relationship between the total number of communications and computations taking place in the simulation. And furthermore, for the complexity of communications, the time overhead of polling is not taken into account in the calculation of the speed-up ratio.

Table 2 evaluates the performance of the proposed Bitcoin miner, where \( P \) is the number of scan_and_hash blocks. The reference for the speed-up ratio is the sequential model, and all the experiments are conducted with a total of 500 proof-of-works. The results reflect the perfect parallelization of our design, which is in accord with the expected ones mentioned in section 3.2.1. Note that our current simulator only runs on a single thread, hence the run times are roughly the same under different number of workers. And since the proof-of-work is a matter of guessing a random number, the differences between the run times are reasonabale.

Table 3 shows the relationship between the total number of communications and computations. From the structure of our design shown in Figure 5, we can deduce that \( N_{\text{comm}} \) is expected to be \( 2 \times P \times N_{\text{comp}} \). This table illustrates that the experimental result agrees well with the theoretical one, which also in turn indicates that our design is very good in parallelism.

5 Conclusion and Future Work

In this document, we have created a system-level model of a bitcoin miner based on its C++ reference code. We improved the model by exploiting parallelization. The SystemC 2.3.1-Accellera simulator is used to find the speed-up ratio of our design, which grows almost linearly with the number of scan_and_hash blocks. In future work, we plan to improve the hash-rate by combining thread-level parallelism with data-level parallelism, as well as introducing some better synchronization methods. Researches about power and area may also be taken into consideration.

References

https://en.wikipedia.org/wiki/Bitcoin

[3] How to derive difficulty target from nBit.  

https://github.com/pooler/cpuminer


[6] Open SystemC Initiative,  


[8] solo mining and pooled mining.  
A Appendix

A.1 Source Code of Parallel Bitcoin Miner in SystemC

Listing 1: config.h

```c
#ifndef _CONFIG_H
#define _CONFIG_H

#define PAR 8
#define N_PAR
#define BUF_SIZE1 2048
#define SET_STACK_SIZE() set_stack_size(128*1024*1024)

#define OPT_N_THREADS 1;
#define OPT_SCANTIME 5;
#define OPT_QUIET true;
#define OPT_BENCHMARK true

#endif
```

Listing 2: types.h

```c
#ifndef _TYPES
#define _TYPES

#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <sys/time.h>
#include <time.h>

#include "systemc.h"
#include "config.h"

#define bswap_32(x) ((((x) << 24) & 0xff000000u) | (((x) << 8) & 0x00ff0000u) |
| (((x) >> 8) & 0x0000ff00u) | (((x) >> 24) & 0x000000ffu))

typedef struct work {
    uint32_t data[32];
    uint32_t target[8];

    /*
     * int height;
     * char* txs;
     * char* workid;
     * char* job_id;
     * size_t xnonce2_len;
     * unsigned char* xnonce2;
     */

    work(void)
    {
        for (int i = 0; i < 32; i++)
            data[i] = 0;
        for (int i = 0; i < 8; i++)
            target[i] = 0;
    }
}
```
work\& \texttt{operator\{const work\& copy\}\{}
\texttt{\quad for (int i = 0; i < 32; i++)}
\texttt{\quad data[i] = copy.data[i];}
\texttt{\quad for (int i = 0; i < 8; i++)}
\texttt{\quad target[i] = copy.target[i];}
\texttt{\quad return \&\&\& this;}
\texttt{\}}
\texttt{\}}
\texttt{uint32\_t\& \texttt{operator\[\](const int index)}
\texttt{\{ return data[index]; \}}
\texttt{\}}
\texttt{operator uint32\_t\*()}
\texttt{\{ return data; \}}
\texttt{\}}\ \texttt{WORK;}
\texttt{typedef struct scan\_work \{}
\texttt{\quad uint32\_t data[32];}
\texttt{\quad uint32\_t target[8];}
\texttt{\quad uint32\_t max\_nonce;}
\texttt{\quad unsigned long hashes\_done;}
\texttt{\} scan\_work \{ void\}
\texttt{\{ for (int i = 0; i < 32; i++)}
\texttt{\quad data[i] = 0;}
\texttt{\quad for (int i = 0; i < 8; i++)}
\texttt{\quad target[i] = 0;}
\texttt{\quad hashes\_done=0;}
\texttt{\}}
\texttt{\}}
\texttt{scan\_work\& \texttt{operator\{const scan\_work\& copy\}\{}
\texttt{\quad for (int i = 0; i < 32; i++)}
\texttt{\quad data[i] = copy.data[i];}
\texttt{\quad for (int i = 0; i < 8; i++)}
\texttt{\quad target[i] = copy.target[i];}
\texttt{\quad hashes\_done=copy.hashes\_done;}
\texttt{\quad return \&\&\& this;}
\texttt{\}}
\texttt{\}}
\texttt{uint32\_t\& \texttt{operator\[\](const int index)}
\texttt{\{ return data[index]; \}}
\texttt{\}}
\texttt{operator uint32\_t\*()}
\texttt{\{ return data; \}}
\texttt{\}}\ \texttt{SCAN\_WORK;}
\texttt{typedef struct config\_scan\_work \{}
\texttt{\quad uint32\_t start\_point;}
\texttt{\quad int index;}
\texttt{\} config\_scan\_work \{ void\}
\texttt{\}}


```cpp
113  
114  {
115      start_point=0;
116      index=0;
117  }
118  
119  config_scan_work operator=(const config_scan_work& copy) {
120      start_point=copy.start_point;
121      index=copy.index;
122      return *this;
123  }
124  
125  uint32_t& operator[](const int index) {
126      return start_point;
127  }
128  
129  operator uint32_t*() {
130      return &start_point;
131  }
132  
133  } CONFIG_SCAN_WORK;
134  
135  
136  template<class T> class sc_T_sender: virtual public sc_interface {
137  public:
138      virtual void write(T) = 0;
139  };
140  
141  template<class T> class sc_T_receiver: virtual public sc_interface {
142  public:
143      virtual void read(T&) = 0;
144  };
145  
146  template<class T> class sc_T_queue: public sc_channel,
147         public sc_T_sender<T>, public sc_T_receiver<T> {
148  public:
149      sc_T_queue(sc_module_name name, int size_ = 4): sc_channel(name) {
150          size = size_;
151          buf = (T*)malloc(sizeof(T)*size);
152          reset();
153      }
154      ~sc_T_queue() {
155          delete [] buf;
156      }
157      
158      int num_available() {
159          return size - free_slots;
160      }
161      
162      int num_free() {
163          return free_slots;
164      }
165      
166      void write(T X) {
167          if (num_free() == 0)
168              wait(data_read_event);
169          buf[wi] = X;
170      }
```
void write(T Y)
{
    if (num_available() == 0)
    wait(data_written_event);
    Y = buf[ri];
    ri = (ri+1) % size;
    free_slots ++;
    data_read_event.notify(SC_ZERO_TIME);
}

void reset()
{
    free_slots = size;
    ri = 0;
    wi = 0;
}

private:
    int size;    // size
    T *buf;      // circular buffer
    int free_slots; // free space
    int ri;      // read index
    int wi;      // write index
    sc_event data_read_event;
    sc_event data_written_event;
};

typedef sc_T_sender< WORK> i_work_sender;
typedef sc_T_receiver< WORK> i_work_receiver;
typedef sc_T_queue< WORK> c_work_queue;
typedef sc_T_sender<unsigned> i_uint_sender;
typedef sc_T_receiver<unsigned> i_uint_receiver;
typedef sc_T_queue<unsigned> c_uint_queue;
typedef sc_T_sender<SCAN_WORK> i_scan_work_sender;
typedef sc_T_receiver<SCAN_WORK> i_scan_work_receiver;
typedef sc_T_queue<SCAN_WORK> c_scan_work_queue;
typedef sc_T_sender<int> i_num_sender;
typedef sc_T_receiver<int> i_num_receiver;
typedef sc_T_queue<int> c_num_queue;
typedef sc_T_sender<CONFIG_SCAN_WORK> i_config_scan_work_sender;
typedef sc_T_receiver<CONFIG_SCAN_WORK> i_config_scan_work_receiver;
typedef sc_T_queue<CONFIG_SCAN_WORK> c_config_scan_work_queue;

static inline uint32_t swab32(uint32_t v) //0x12345678 => 0x78563412
{
    return bswap_32(v);
}

#endif

Listing 3: getwork.h

#ifndef _GET_WORK

#endif

9
```cpp
#define _GET_WORK

#include "systemc.h"
#include "types.h"

class Get_Work : public sc_module
{
public:
    sc_port<i_work_sender> WorkOut;
    struct work g_work; // this should be shared between many scanhash blocks
    // CONSTRUCTOR
    SC_HAS_PROCESS(Get_Work);
    Get_Work(sc_module_name name);
    // METHODS
    bool workio_get_work(uint32_t p, uint32_t t);
    void main();
};

#include "getwork.h"

Get_Work::Get_Work(sc_module_name name)
    : sc_module(name)
{
    SC_THREAD(main);
    SET_STACK_SIZE();
}

bool Get_Work::workio_get_work(uint32_t p, uint32_t t)
{
    int failures = 0;
    memset(g_work.data, 0x55, 76);
    g_work.data[17] = swab32(time(NULL)) + t * 10; // include here
    memset(g_work.data + 19, 0x00, 52); // 19-31 uint32_t
    g_work.data[20] = 0x80000000;
    g_work.data[31] = 0x00000280;
    memset(g_work.target, 0x00, sizeof(g_work.target)); // 256 bit
    for (int j = 0; j < 7; j++)
    {
        g_work.target[j] = 0xffffffff;
        g_work.target[7] = p;
    }
    return true;
}

void Get_Work::main()
{
    bool ok = true;
    int i = 0;
}
```

Listing 4: getwork.cc
```c
uint32_t n=0x0000ffff;
while(i<500){
    ok = workio_get_work(n,i);
    i=i+1;
    WorkOut->write(g_work);
}
```

Listing 5: miner.h

```c
#ifndef _Miner
#define _Miner
#include "systemc.h"
#include "types.h"
#include "miner_thread.h"
#include "scan_hash.h"

#define QUEUES_AND_SCANNER_CTOR(n) \
c_num_queue Flag_to_scanner_q##n; \ 
c_num_queue Flag_from_scanner_q##n; \ 
c_scan_work_queue Work_to_scanner_q##n; \ 
c_scan_work_queue Work_from_scanner_q##n; \ 
c_config_scan_work_queue Config_q##n; \ 
Scan_Hash scan_hash##n; \ 

#define QUEUES_AND_SCANNER_INIT(n) \ 
, Flag_to_scanner_q##n (*Flag_to_scanner_q" #n, BUF_SIZE1) \ 
, Flag_from_scanner_q##n (*Flag_from_scanner_q" #n, BUF_SIZE1) \ 
, Work_to_scanner_q##n (*Work_to_scanner_q" #n, BUF_SIZE1) \ 
, Work_from_scanner_q##n (*Work_from_scanner_q" #n, BUF_SIZE1) \ 
, Config_q##n (*Config_q" #n, BUF_SIZE1) \ 
, scan_hash##n (*scan_hash" #n) \ 

#define QUEUES_AND_SCANNER_CONNECTION(n) \ 
miner_thread.flag_to_scanner_##n (Flag_to_scanner_q##n); \ 
miner_thread.flag_from_scanner_##n (Flag_from_scanner_q##n); \ 
miner_thread.work_to_scanner_##n (Work_to_scanner_q##n); \ 
miner_thread.work_from_scanner_##n (Work_from_scanner_q##n); \ 
miner_thread.config_to_scanner_##n (Config_q##n); \ 
scan_hash##n.flag_to_main (Flag_from_scanner_q##n); \ 
scan_hash##n.flag_from_main (Flag_to_scanner_q##n); \ 
scan_hash##n.work_to_main (Work_from_scanner_q##n); \ 
scan_hash##n.work_from_main (Work_to_scanner_q##n); \ 
scan_hash##n.config_from_main (Config_q##n); \ 

class Miner : public sc_module
{

public:

    QUEUES_AND_SCANNER_CTOR(0)
#if PAR > 1
    QUEUES_AND_SCANNER_CTOR(1)
#endif
#if PAR > 2
    QUEUES_AND_SCANNER_CTOR(2)
    QUEUES_AND_SCANNER_CTOR(3)
#endif
```
```cpp
#include "miner.h"

Miner::Miner(sc_module_name name)
    : sc_module(name)
    , miner_thread("miner_thread")
    QUEUES_AND_SCANNER_INIT(0)
#if PAR > 1
    QUEUES_AND_SCANNER_INIT(1)
#endif
#if PAR > 2
    QUEUES_AND_SCANNER_INIT(2)
    QUEUES_AND_SCANNER_INIT(3)
#endif
#if PAR > 4
    QUEUES_AND_SCANNER_INIT(4)
    QUEUES_AND_SCANNER_INIT(5)
    QUEUES_AND_SCANNER_INIT(6)
    QUEUES_AND_SCANNER_INIT(7)
#endif
{
    miner_thread.WorkIn(WorkIn); //get work from getwork
    miner_thread.WorkOut(WorkOut); //send result to submitwork
    QUEUES_AND_SCANNER_CONNECTION(0)
#if PAR > 1
    QUEUES_AND_SCANNER_CONNECTION(1)
#endif
#if PAR > 2
    QUEUES_AND_SCANNER_CONNECTION(2)
    QUEUES_AND_SCANNER_CONNECTION(3)
#endif
#if PAR > 4
    QUEUES_AND_SCANNER_CONNECTION(4)
    QUEUES_AND_SCANNER_CONNECTION(5)
    QUEUES_AND_SCANNER_CONNECTION(6)
    QUEUES_AND_SCANNER_CONNECTION(7)
#endif
}
```

Listing 6: miner.cc
Listing 7: miner_thread.h

```cpp
#ifndef _MINER_THREAD
#define _MINER_THREAD

#include "systemc.h"
#include "types.h"

#define PORT_AND_CONFIG_CTOR(n) \
    sc_port<i_num_sender> flag_to_scanner_##n; \
    sc_port<i_num_receiver> flag_from_scanner_##n; \
    sc_port<i_scan_work_sender> work_to_scanner_##n; \
    sc_port<i_scan_work_receiver> work_from_scanner_##n; \
    sc_port<i_config_scan_work_sender> config_to_scanner_##n; \
    struct config_scan_work config##n;

#define SEND_START_MSG(n) \
    int flag##n; \
    flag_to_scanner_##n->write(1); \
    wait(1,SC_NS); \

#define SEND_CONFIG_AND_WORK(n) \
    config##n.start_point = max_nonce/N*n; \
    config##n.index = n; \
    work_to_scanner_##n->write(scan_work); \
    wait(1,SC_NS); \
    config_to_scanner_##n->write(config##n); \
    wait(1,SC_NS); \

#define POLLING(n) \
    flag_from_scanner_##n->read(flag##n); \
    wait(1,SC_NS); \
    if(flag##n>0) { \
        flag_glob=1; \
        work_from_scanner_##n->read(scan_work); \
        wait(1,SC_NS); \
        printf("result from scanner %d\n",n); \
    }

class Miner_Thread : public sc_module
{

public:

    PORT_AND_CONFIG_CTOR(0)
    PORT_AND_CONFIG_CTOR(1)
    PORT_AND_CONFIG_CTOR(2)
    PORT_AND_CONFIG_CTOR(3)
    PORT_AND_CONFIG_CTOR(4)
    PORT_AND_CONFIG_CTOR(5)
    PORT_AND_CONFIG_CTOR(6)
    PORT_AND_CONFIG_CTOR(7)

    sc_port<i_work_receiver> WorkIn;
    sc_port<i_work_sender> WorkOut;

    struct work g_work;  //this should be shared between many scanhash blocks
```
struct scan_work scan_work;

double thr_hashrate;

//CONSTRUCTOR
SC_HAS_PROCESS(Miner_Thread);
Miner_Thread(sc_module_name name);

//METHODS
void miner_thread();
void main();
#if PAR > 4
SEND_START_MSG(4)
SEND_START_MSG(5)
SEND_START_MSG(6)
SEND_START_MSG(7)
#endif

int simulation_loop=0;

while(1){
    simulation_loop++;
    WorkIn->read(g_work);
    int flag_glb=0;
    printf("target=
");
    for(int m=0;m<8;m++)
        printf("%08x",g_work.target[m]);
    printf("\n");
    /* preprocessing before send work */
    work_copy(&work, &g_work);
    work.data[19] = 0;
    max_nonce = end_nonce;
    hashes_done = 0; // hashes_done is the number of hashes scanned
    gettimeofday(&tv_start, NULL);
    for(int j=0;j<32;j++)
        scan_work.data[j]=work.data[j];
    for(int j=0;j<8;j++)
        scan_work.target[j]=work.target[j];
    scan_work.max_nonce=max_nonce;
    scan_work.hashes_done=hashes_done;
    SEND_CONFIG_AND_WORK(0)
    #if PAR > 1
    SEND_CONFIG_AND_WORK(1)
    #endif
    #if PAR > 2
    SEND_CONFIG_AND_WORK(2)
    SEND_CONFIG_AND_WORK(3)
    #endif
    #if PAR > 4
    SEND_CONFIG_AND_WORK(4)
    SEND_CONFIG_AND_WORK(5)
    SEND_CONFIG_AND_WORK(6)
    SEND_CONFIG_AND_WORK(7)
    #endif

    while(true){
        POLLING(0)
        #if PAR > 1
        POLLING(1)
        #endif
        #if PAR > 2
        POLLING(2)
        POLLING(3)
        #endif
        #if PAR > 4
        POLLING(4)
        POLLING(5)
        POLLING(6)
        POLLING(7)
        #endif
if(flag_glb>0) {
    flag_to_scanner_0->write(1); wait(1,SC_NS);
#if PAR > 1
    flag_to_scanner_1->write(1); wait(1,SC_NS);
#endif
    flag_to_scanner_2->write(1); wait(1,SC_NS);
    flag_to_scanner_3->write(1);
    wait(1,SC_NS);
#if PAR > 2
    flag_to_scanner_4->write(1); wait(1,SC_NS);
    flag_to_scanner_5->write(1); wait(1,SC_NS);
    flag_to_scanner_6->write(1); wait(1,SC_NS);
    flag_to_scanner_7->write(1); wait(1,SC_NS);
#endif
    break;
}
flag_to_scanner_0->write(0); wait(1,SC_NS);
#if PAR > 1
    flag_to_scanner_1->write(0); wait(1,SC_NS);
#endif
#if PAR > 2
    flag_to_scanner_2->write(0); wait(1,SC_NS);
    flag_to_scanner_3->write(0); wait(1,SC_NS);
#endif
#if PAR > 4
    flag_to_scanner_4->write(0); wait(1,SC_NS);
    flag_to_scanner_5->write(0); wait(1,SC_NS);
    flag_to_scanner_6->write(0); wait(1,SC_NS);
    flag_to_scanner_7->write(0); wait(1,SC_NS);
#endif
}

//postprocessing after result received
hashes_done=scan_work.hashes_done;
//printf("from miner_thread: hash done \d\n", hashes_done);

/* record scanhash elapsed time */
timeval_subtract(&diff, &tv_end, &tv_start);
if(diff.tv_usec || diff.tv_sec) {
    thr_hashrate = hashes_done / (diff.tv_sec + 1e-6 * diff.tv_usec);
}
if(OPT_BENCHMARK) {
    double hashrate = 0;
    hashrate += thr_hashrate;
    sprintf(s, hashrate >= 1e6 ? "%lf" : "%lf", 1e-3 * hashrate*N);
    printf("Total: %s khash/s
", s);
}
for(int j=0;j<32;j++)
g_work.data[j]=scan_work.data[j];
WorkOut->write(g_work); wait(1,SC_NS);
#ifndef PLATFORM
#define PLATFORM

#include "systemc.h"
#include "types.h"
#include "getwork.h"
#include "submitwork.h"
#include "miner.h"

class Platform : public sc_module
{
    //get work from outside
    //proceed by miner_thread
    //send to scan_hash
    //back to miner_thread
    //send to outside

public:
    c_work_queue Work_q1;
    c_work_queue Work_q2;

    /* miner_thread send work to scan_hash
    scan_hash hashes and send back hash to miner_thread */
    Miner miner;
    Get_Work get_work;
    Submit_Work submit_work;

    SC_HAS_PROCESS(Platform);
    //CONSTRUCTOR
    Platform(sc_module_name name);
    //METHODS
};
#endif

#include "platform.h"
Platform::Platform(sc_module_name name)
    : sc_module(name)
    , miner("miner")
Listing 11: scan_hash.h

```c
#ifndef __SCAN_HASH
#define __SCAN_HASH

#include "systemc.h"
#include "types.h"

class Scan_Hash : public sc_module {
public:
  sc_port<i_num_sender> flag_to_main;
  sc_port<i_scan_work_sender> work_to_main;
  sc_port<i_num_receiver> flag_from_main;
  sc_port<i_scan_work_receiver> work_from_main;
  sc_port<i_config_scan_work_receiver> config_from_main;
  int flag;

  struct scan_work scan_work;
  struct config_scan_work config_work;

  //CONSTRUCTOR
  SC_HAS_PROCESS(Scan_Hash);
  Scan_Hash(sc_module_name name);

  //METHODS
  void main();
};
#endif
```

Listing 12: scan_hash.cc

```c
#include "scan_hash.h"
#include "sha2.h"
#include "util.h"

Scan_Hash::Scan_Hash(sc_module_name name)
  : sc_module(name)
    { SC_THREAD(main);
      SET_STACK_SIZE();
    }

void Scan_Hash::main()
{
```

1, get_work("get_work")
2, submit_work("submit_work")
3, Work_q1("Work_q1", BUF_SIZE1)
4, Work_q2("Work_q2", BUF_SIZE1)
5 {
6    get_work.WorkOut(Work_q1);
7    miner.WorkIn(Work_q1);
8
9    miner.WorkOut(Work_q2);
10    submit_work.WorkIn(Work_q2);
11 }

```
bool rc;

uint32_t data[64];
uint32_t hash[8];
uint32_t midstate[8];
uint32_t prehash[8];
uint32_t n;
uint32_t first_nonce;
uint32_t Htarg;
int loop_id;
int index;

while(true){
    flag_from_main->read(flag);
    if(flag>0){
        work_from_main->read(scan_work);
        config_from_main->read(config_work);
        index=config_work.index;
        n = scan_work.data[19]+config_work.start_point - 1;
        first_nonce = scan_work.data[19]+config_work.start_point;
        Htarg = scan_work.target[7];
        memcpy(data, scan_work.data + 16, 64);
        sha256d_preextend(data);
        sha256_init(midstate);
        sha256_transform(midstate, scan_work.data, 0);
        memcpy(prehash, midstate, 32);
        loop_id=0;
        printf("hashing block:");
        printf("%d",index);
        /*for(int m=0;m<8;m++)
        printf("%08x",scan_work.target[m]);*/
        printf("nonce starting from %ld\n",first_nonce);
    }
    data[3] = ++n;
    sha256d_ms(hash, data, midstate, prehash);
    if (swab32(hash[7]) <= Htarg) {
        scan_work.data[19] = data[3];
        sha256d_80_swap(hash, scan_work.data);
        if (fulltest(hash, scan_work.target)) {
            flag_to_main->write(1);
            scan_work.hashes_done = n - first_nonce + 1;
            scan_work.data[19] = n;
            printf("\nhash found from %d\nhash done=%d\nhash: %08x\n",index,n,scan_work.
hashes_done);
            for(int m=0;m<8;m++)
                printf("%08x",hash[m]);
            printf("\n");
            work_to_main->write(scan_work);
            continue;
        }
    }
    else{
        printf("\nhash found from %d\nhash done=%d\nhash: %08x\n",index,n,scan_work.
hashes_done);
        for(int m=0;m<8;m++)
            printf("%08x",hash[m]);
        printf("\n");
        work_to_main->write(scan_work);
        continue;
    }
}
flag_to_main->write(0);
}
/*
if(((loop_id)%10000)==0){
printf("\nloop= %ld n= %ld index %d
hash :
",loop_id,n,index);
for(int m=0;m<8;m++)
printf("%08x",hash[m]);
}*/
loop_id++;
//wait(1,SC_NS);
}
return ;

Listing 13: sha2.h

#ifndef _SHA2
#define _SHA2
#include "types.h"

const uint32_t sha256d_hash1[16] = {
0x00000000, 0x00000000, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x00000000, 0x00000000,
0x80000000, 0x00000000, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x00000000, 0x00000000,
};

const uint32_t sha256_h[8] = {
//each is 32 bits, total 256 bits
0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19
};

const uint32_t sha256_k[64] = {
0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
0x3956c25b, 0x8eb6db3a, 0x6b901122, 0xd62f1d30,
0x40e0390f, 0x57101f07, 0x1f88d1c7, 0x5488eb7c,
0x9b05688c, 0x1f83d9ab, 0x5be0cd19, 0x682e6ff3,
0x041e358c, 0x14292967, 0x27b48352, 0x362fa46f,
0x0f721eda, 0x0960d91c, 0x110a62c8, 0x2f48b7e0,
0x34be4b59, 0x391c0cb3, 0x4ed8a4d0, 0x3bde0equiv,
0x243185be, 0x0c54c4f6, 0x38be3dec, 0x0dd2a6df,
0x72be5d74, 0x83fbc9e2, 0x08a2b703, 0x064f5395,
0x521f2189, 0x5a517e15, 0x5be0cd19, 0x682e6ff3,
0x041e358c, 0x14292967, 0x27b48352, 0x362fa46f,
0x0f721eda, 0x0960d91c, 0x110a62c8, 0x2f48b7e0,
0x34be4b59, 0x391c0cb3, 0x4ed8a4d0, 0x3bde0equiv,
0x243185be, 0x0c54c4f6, 0x38be3dec, 0x0dd2a6df,
0x72be5d74, 0x83fbc9e2, 0x08a2b703, 0x064f5395,
0x521f2189, 0x5a517e15, 0x5be0cd19, 0x682e6ff3,
0x041e358c, 0x14292967, 0x27b48352, 0x362fa46f,
0x0f721eda, 0x0960d91c, 0x110a62c8, 0x2f48b7e0,
0x34be4b59, 0x391c0cb3, 0x4ed8a4d0, 0x3bde0equiv,
0x243185be, 0x0c54c4f6, 0x38be3dec, 0x0dd2a6df,
0x72be5d74, 0x83fbc9e2, 0x08a2b703, 0x064f5395,
0x521f2189, 0x5a517e15, 0x5be0cd19, 0x682e6ff3,
0x041e358c, 0x14292967, 0x27b48352, 0x362fa46f,
0x0f721eda, 0x0960d91c, 0x110a62c8, 0x2f48b7e0,
0x34be4b59, 0x391c0cb3, 0x4ed8a4d0, 0x3bde0equiv,
0x243185be, 0x0c54c4f6, 0x38be3dec, 0x0dd2a6df,
0x72be5d74, 0x83fbc9e2, 0x08a2b703, 0x064f5395,
0x521f2189, 0x5a517e15, 0x5be0cd19, 0x682e6ff3,
0x041e358c, 0x14292967, 0x27b48352, 0x362fa46f,
0x0f721eda, 0x0960d91c, 0x110a62c8, 0x2f48b7e0,
0x34be4b59, 0x391c0cb3, 0x4ed8a4d0, 0x3bde0equiv,
```c
#define RND(a, b, c, d, e, f, g, h, k) \
  do { \
    t0 = h + S1(e) + Ch(e, f, g) + k; \
    t1 = S0(a) + Maj(a, b, c); \
    d += t0; \
    h = t0 + t1; \
  } while (0)

#define RNDr(S, W, i) \
  RND(S[(64 - i) % 8], S[(65 - i) % 8], \n  S[(66 - i) % 8], S[(67 - i) % 8], \n  S[(68 - i) % 8], S[(69 - i) % 8], \n  S[(70 - i) % 8], S[(71 - i) % 8], \n  W[i] + sha256_k[i])

void sha256d_prehash(uint32_t *S, const uint32_t *W);
void sha256d_ms(uint32_t *hash, uint32_t *W, 
  const uint32_t *midstate, const uint32_t *prehash);
void sha256d_80_swap(uint32_t *hash, const uint32_t *data);
void sha256d_preextend(uint32_t *W);
void sha256d_init(uint32_t *state);
void sha256d_transform(uint32_t *state, const uint32_t *block, int swap);
```

Listing 14: sha2.cc

```c
#include "sha2.h"
#include "util.h"

/* Elementary functions used by SHA256 */
#define Ch(x, y, z) ((x & (y ^ z)) ^ z)
#define Maj(x, y, z) ((x & (y | z)) | (y & z))
#define ROTR(x, n) ((x >> n) | (x << (32 - n)))
#define S0(x) (ROTR(x, 2) ^ ROTR(x, 13) ^ ROTR(x, 22))
#define S1(x) (ROTR(x, 6) ^ ROTR(x, 11) ^ ROTR(x, 25))
#define s0(x) (ROTR(x, 7) ^ ROTR(x, 18) ^ (x >> 3))
#define s1(x) (ROTR(x, 17) ^ ROTR(x, 19) ^ (x >> 10))

void sha256d_prehash(uint32_t *S, const uint32_t *W) 
{
  uint32_t t0, t1;
  RNDr(S, W, 0);
  RNDr(S, W, 1);
  RNDr(S, W, 2);
}

void sha256d_ms(uint32_t *hash, uint32_t *W, 
  const uint32_t *midstate, const uint32_t *prehash) 
{
  uint32_t S[64];
  uint32_t t0, t1;
  int i;
  S[18] = W[18];
  S[19] = W[19];
  S[20] = W[20];
  S[22] = W[22];
  S[23] = W[23];
```

21
S[31] = W[31];
W[18] += s0(W[3]);
W[19] += W[3];
W[20] += s1(W[18]);
W[21] = s1(W[19]);
W[22] += s1(W[20]);
W[23] += s1(W[21]);
W[24] += s1(W[22]);
W[25] = s1(W[23]) + W[18];
W[26] = s1(W[24]) + W[19];
W[27] = s1(W[25]) + W[20];
W[28] = s1(W[26]) + W[21];
W[29] = s1(W[27]) + W[22];
W[30] += s1(W[28]) + W[23];
W[31] += s1(W[29]) + W[24];
for (i = 32; i < 64; i += 2) {
    W[i] = s1(W[i - 2]) + W[i - 7] + s0(W[i - 15]) + W[i - 16];
    W[i + 1] = s1(W[i - 1]) + W[i - 6] + s0(W[i - 14]) + W[i - 15];
}
memcpy(S, prehash, 32);
RNDr(S, W, 3);
RNDr(S, W, 4);
RNDr(S, W, 5);
RNDr(S, W, 6);
RNDr(S, W, 7);
RNDr(S, W, 8);
RNDr(S, W, 9);
RNDr(S, W, 10);
RNDr(S, W, 11);
RNDr(S, W, 12);
RNDr(S, W, 13);
RNDr(S, W, 14);
RNDr(S, W, 15);
RNDr(S, W, 16);
RNDr(S, W, 17);
RNDr(S, W, 18);
RNDr(S, W, 19);
RNDr(S, W, 20);
RNDr(S, W, 21);
RNDr(S, W, 22);
RNDr(S, W, 23);
RNDr(S, W, 24);
RNDr(S, W, 25);
RNDr(S, W, 26);
RNDr(S, W, 27);
RNDr(S, W, 28);
RNDr(S, W, 29);
RNDr(S, W, 30);
RNDr(S, W, 31);
RNDr(S, W, 32);
RNDr(S, W, 33);
RNDr(S, W, 34);
RNDr(S, W, 35);
RNDr(S, W, 36);
RNDr(S, W, 37);
RNDr(S, W, 38);
RNDr(S, W, 39);
RNDr(S, W, 40);
RNDr(S, W, 41);
RNDr(S, W, 42);
RNDr(S, W, 43);
RNDr(S, W, 44);
RNDr(S, W, 45);
RNDr(S, W, 46);
RNDr(S, W, 47);
RNDr(S, W, 48);
RNDr(S, W, 49);
for (i = 0; i < 8; i++)
    S[i] += midstate[i];

W[18] = S[18];
W[19] = S[19];
W[20] = S[20];
W[22] = S[22];
W[23] = S[23];
W[24] = S[24];
W[30] = S[30];
W[31] = S[31];

memcpy(S + 8, sha256d_hash1 + 8, 32);
S[16] = s1(sha256d_hash1[14]) + sha256d_hash1[9] + s0(S[1]) + S[0];
S[17] = s1(sha256d_hash1[15]) + sha256d_hash1[10] + s0(S[2]) + S[1];
S[18] = s1(S[16]) + sha256d_hash1[11] + s0(S[3]) + S[2];
S[19] = s1(S[17]) + sha256d_hash1[12] + s0(S[4]) + S[3];
S[20] = s1(S[18]) + sha256d_hash1[13] + s0(S[5]) + S[4];
S[21] = s1(S[19]) + sha256d_hash1[14] + s0(S[6]) + S[5];
S[22] = s1(S[20]) + sha256d_hash1[15] + s0(S[7]) + S[6];
S[23] = s1(S[21]) + S[16] + s0(sha256d_hash1[8]) + S[7];
S[24] = s1(S[22]) + S[17] + s0(sha256d_hash1[9]) + sha256d_hash1[8];
S[25] = s1(S[23]) + S[18] + s0(sha256d_hash1[10]) + sha256d_hash1[9];
S[26] = s1(S[24]) + S[19] + s0(sha256d_hash1[11]) + sha256d_hash1[10];
S[27] = s1(S[25]) + S[20] + s0(sha256d_hash1[12]) + sha256d_hash1[11];
S[28] = s1(S[26]) + S[21] + s0(sha256d_hash1[13]) + sha256d_hash1[12];
S[29] = s1(S[27]) + S[22] + s0(sha256d_hash1[14]) + sha256d_hash1[13];
S[30] = s1(S[28]) + S[23] + s0(sha256d_hash1[15]) + sha256d_hash1[14];
S[31] = s1(S[29]) + S[24] + s0(S[16]) + sha256d_hash1[15];

for (i = 32; i < 60; i += 2) {
    S[i] = s1(S[i - 2]) + S[i - 7] + s0(S[i - 15]) + S[i - 16];
    S[i+1] = s1(S[i - 1]) + S[i - 6] + s0(S[i - 14]) + S[i - 15];
}
S[60] = s1(S[58]) + S[53] + s0(S[45]) + S[44];
RNDr(hash, S, 18);
RNDr(hash, S, 19);
RNDr(hash, S, 20);
RNDr(hash, S, 21);
RNDr(hash, S, 22);
RNDr(hash, S, 23);
RNDr(hash, S, 24);
RNDr(hash, S, 25);
RNDr(hash, S, 26);
RNDr(hash, S, 27);
RNDr(hash, S, 28);
RNDr(hash, S, 29);
RNDr(hash, S, 30);
RNDr(hash, S, 31);
RNDr(hash, S, 32);
RNDr(hash, S, 33);
RNDr(hash, S, 34);
RNDr(hash, S, 35);
RNDr(hash, S, 36);
RNDr(hash, S, 37);
RNDr(hash, S, 38);
RNDr(hash, S, 39);
RNDr(hash, S, 40);
RNDr(hash, S, 41);
RNDr(hash, S, 42);
RNDr(hash, S, 43);
RNDr(hash, S, 44);
RNDr(hash, S, 45);
RNDr(hash, S, 46);
RNDr(hash, S, 47);
RNDr(hash, S, 48);
RNDr(hash, S, 49);
RNDr(hash, S, 50);
RNDr(hash, S, 51);
RNDr(hash, S, 52);
RNDr(hash, S, 53);
RNDr(hash, S, 54);
RNDr(hash, S, 55);
RNDr(hash, S, 56);

  + S[57] + sha256_k[57];
  + S[58] + sha256_k[58];
hash[0] += hash[4] + S1(hash[1]) + Ch(hash[1], hash[2], hash[3])
  + S[59] + sha256_k[59];
hash[7] += hash[3] + S1(hash[0]) + Ch(hash[0], hash[1], hash[2])
  + S[60] + sha256_k[60]
  + sha256_h[7];

void sha256d_80_swap(uint32_t *hash, const uint32_t *data)
{
  uint32_t S[16];
  int i;

  //first sha256, performed to data, stored into S
  sha256_init(S);
  //data is 1024 bits, so to calculate the sha256, we need to perform the algo twice
  sha256_transform(S, data, 0);
  sha256_transform(S, data + 16, 0);
  memcpy(S + 8, sha256d_hash1 + 8, 32);

  //second sha256, performed to S, stored into hash
  sha256_init(hash);
  sha256_transform(hash, S, 0);
  for (i = 0; i < 8; i++)
    hash[i] = swab32(hash[i]);
//W[i] = s1(W[i - 2]) + W[i - 7] + s0(W[i - 15]) + W[i - 16];
W[16] = s1(W[14]) + W[9] + s0(W[1]) + W[0];
W[17] = s1(W[15]) + W[10] + s0(W[2]) + W[1];
W[19] = s1(W[17]) + W[12] + s0(W[4]);
W[22] = W[15] + s0(W[7]) + W[6];
W[23] = W[16] + s0(W[8]) + W[7];
W[25] = s0(W[10]) + W[9];
W[26] = s0(W[11]) + W[10];
W[27] = s0(W[12]) + W[11];
W[28] = s0(W[13]) + W[12];
W[29] = s0(W[14]) + W[13];
W[30] = s0(W[15]) + W[14];
W[31] = s0(W[16]) + W[15];
}

void sha256d_preextend(uint32_t *W)
{
//W[i] = s1(W[i - 2]) + W[i - 7] + s0(W[i - 15]) + W[i - 16];
W[16] = s1(W[14]) + W[9] + s0(W[1]) + W[0];
W[17] = s1(W[15]) + W[10] + s0(W[2]) + W[1];
W[19] = s1(W[17]) + W[12] + s0(W[4]);
W[22] = W[15] + s0(W[7]) + W[6];
W[23] = W[16] + s0(W[8]) + W[7];
W[25] = s0(W[10]) + W[9];
W[26] = s0(W[11]) + W[10];
W[27] = s0(W[12]) + W[11];
W[28] = s0(W[13]) + W[12];
W[29] = s0(W[14]) + W[13];
W[30] = s0(W[15]) + W[14];
W[31] = s0(W[16]) + W[15];
}

void sha256_init(uint32_t *state)
{
memcpy(state, sha256_h, 32);
}

/* SHA256 block compression function. The 256-bit state is transformed via the 512-bit input block to produce a new state. */

void sha256_transform(uint32_t *state, const uint32_t *block, int swap)
{
    uint32_t W[64];
    uint32_t S[8];
    uint32_t t0, t1;
    int i;

    /* 1. Prepare message schedule W. */
    if (swap) {
        for (i = 0; i < 16; i++)
            W[i] = swab32(block[i]);
    } else
        memcpy(W, block, 64);

    /* 2. Initialize working variables. */
    memcpy(S, state, 32);

    /* 3. Mix. */
    //64 loops
    RNDr(S, W, 0);
    RNDr(S, W, 1);
    RNDr(S, W, 2);
    RNDr(S, W, 3);
    RNDr(S, W, 4);
    RNDr(S, W, 5);
    RNDr(S, W, 6);
    RNDr(S, W, 7);
    RNDr(S, W, 8);
    RNDr(S, W, 9);
    RNDr(S, W, 10);

    /* 4. Compress block. */
    */
#ifndef _SUBMIT_WORK
#define _SUBMIT_WORK

#include "systemc.h"
#include "types.h"

class Submit_Work : public sc_module
{
    // 4. Mix local working variables into global state */
    for (i = 0; i < 8; i++)
        state[i] += S[i];
);

Listing 15: submitwork.h
public:
  sc_port<i_work_receiver> WorkIn;

  struct work g_work; //this should be shared between many scanhash blocks

  //CONSTRUCTOR
  SC_HAS_PROCESS(Submit_Work);
  Submit_Work(sc_module_name name);

  //METHODS
  bool workio_submit_work();

  void main();
};

Listing 16: submitwork.cc

#include "submitwork.h"

Submit_Work::Submit_Work(sc_module_name name)
  : sc_module(name)
  {
    SC_THREAD(main);
    SET_STACK_SIZE();
  }

bool Submit_Work::workio_submit_work()
  {
    int failures = 0;
    return true;
  }

void Submit_Work::main()
  {
    int n=1;
    while(true){
      WorkIn->read(g_work);
      workio_submit_work();
      n=n+1;
    }
  }

Listing 17: util.h

#ifndef _UTIL
#define _UTIL

#include "types.h"

bool fulltest(const uint32_t *hash, const uint32_t *target);
int timeval_subtract(struct timeval *result, struct timeval *x,
                        struct timeval *y);

#endif
#include "util.h"

bool fulltest(const uint32_t *hash, const uint32_t *target)
{
    int i;
    bool rc = true;
    for (i = 7; i >= 0; i--)
    {
        if (hash[i] > target[i])
            rc = false;
        break;
        if (hash[i] < target[i])
            rc = true;
        break;
    }
    return rc;
}

/* Subtract the 'struct timeval' values X and Y,
storing the result in RESULT.
Return 1 if the difference is negative, otherwise 0. */
int timeval_subtract(struct timeval *result, struct timeval *x, struct timeval *y)
{
    /* Perform the carry for the later subtraction by updating Y. */
    if (x->tv_usec < y->tv_usec)
    {
        int nsec = (y->tv_usec - x->tv_usec) / 1000000 + 1;
        y->tv_usec -= 1000000 * nsec;
        y->tv_sec += nsec;
    }
    if (x->tv_usec - y->tv_usec > 1000000)
    {
        int nsec = (x->tv_usec - y->tv_usec) / 1000000;
        y->tv_usec += 1000000 * nsec;
        y->tv_usec -= nsec;
    }
    /* Compute the time remaining to wait.
    * 'tv_usec' is certainly positive. */
    result->tv_usec = x->tv_usec - y->tv_usec;
    result->tv_usec = x->tv_usec - y->tv_usec;
    /* Return 1 if result is negative. */
    return x->tv_usec < y->tv_usec;
}

A.2 Makefile

Listing 19: Makefile

SYSTEMC = /opt/pkg/systemc-2.3.1
INCLUDE = -I -I$(SYSTEMC)/include
LIBRARY = $(SYSTEMC)/lib-linux64
CFLAG = $INCLUDE -L$(LIBRARY) -lXlinker -R -lXlinker $(LIBRARY) -lsystemc -O3
CC = g++
RM = rm -f
TARGETS = cpuminer
OFILES = Main.o \\
    util.o \\
    sha2.o \\
    getwork.o \\
    miner.o \\
    miner_thread.o \\
    platform.o \\
    scan_hash.o \\
    submitwork.o \\

HFILES = types.h config.h

all: $(TARGETS)

clean:
    $(RM) *.o $(TARGETS) *.gch

cpuminer: $(OFILES)
    $(CC) $(CFLAG) -o $@

%.o : %.cc $(HFILES)
    $(CC) -c $(INCLUDE) -O3