Assignment 4

Bitcoin Miner

Outline

- Introduction
- Sequential Code
- Grading

Outline

- Introduction
- Sequential Code
- Grading

Introduction - What is a Miner?

- A miner's job is to find out a hash value, such that the block is secured

- More specifically, the primary task of a miner is to find out a proper value called "**nonce**", which is part of a block's header

- The value of nonce has to satisfy some requirements, including:
  - It has to be 32-bit long
  - It has to be less than a pre-defined value called "**Target Difficulty**"

Introduction - What is a Block?

- A block consists of two parts: the **block header** and **block body**

- The block body contains a number of bitcoin transactions (or simply transactions) and coinbase transactions
  - Bitcoin transactions are the records of bitcoin transfers
  - Coinbase transactions records the bitcoins rewarded to the miner who finds the proper **nonce** for the block

- There are several fields included in the block header. We only cares about the following fields.
  - Nonce
  - The hash value of the previous block
  - Merkle root
  - **Target difficulty** (nBits)

Introduction - Block Layout

- The layout of a bitcoin block looks like the table presented below.
  - The **yellow part** corresponds to the block header
  - The **gray part** corresponds to the block body

- A secure hash value is generated (on the right side) for the block header

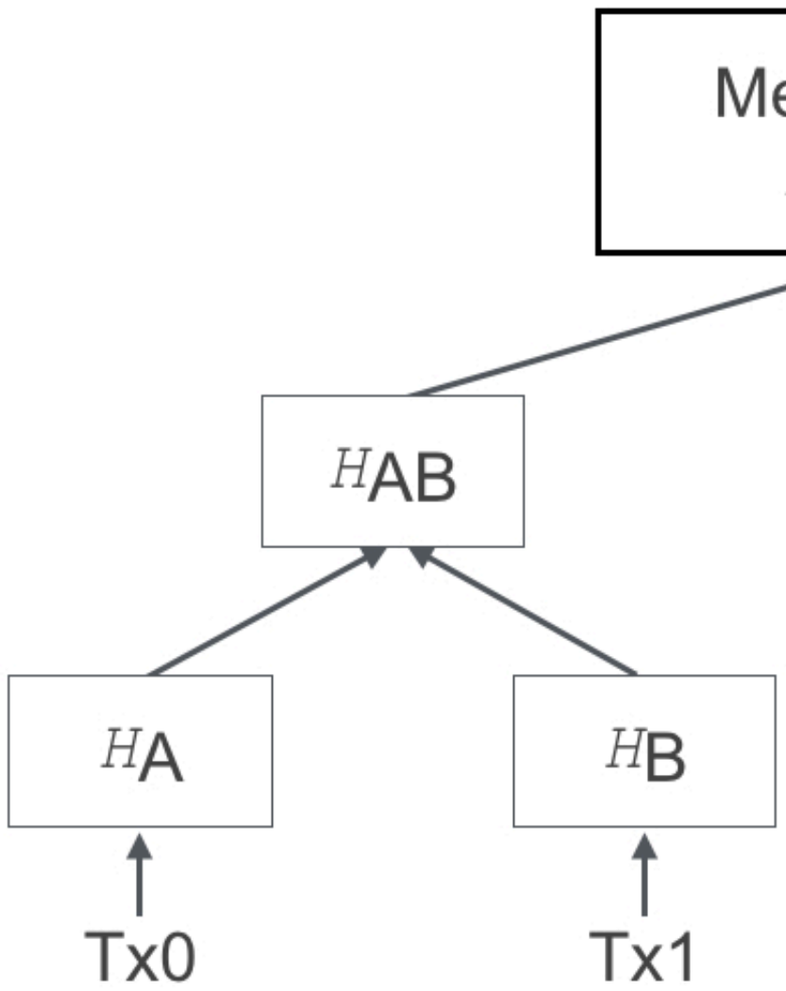
version	02000000
previous block hash (reversed)	17975b97c18ed1f7e255adf297599b55 330edab87803c81701000000000000000
Merkle root (reversed)	8a97295a2747b4f1a0b3948df3990344 c0e19fa6b2b92b3a19c8e6badc141787
timestamp	358b0553
bits	535f0119
nonce	48750833
transaction count	63
coinbase transaction	
transaction	
...	

Block hash

0000000000000000 e067a478024addfe cdc93628978aa52d 91fabd4292982a50

Introduction - Merkle Tree Nodes

- We store the transactions in a data structure called “Merkle Tree”
  - The tree is constructed in a binary tree fashion
  - The root and intermediate nodes of the tree are generated from their children



H_A = SHA256(SHA256(Tx0))

H_B = SAH256(SHA256(Tx1))

H_{AB} = SAH256(SHA256(H_A + H_B))

Introduction - Double SHA-256

```
HA = SHA256( SHA256(Tx0) )
```

```
HB = SAH256( SHA256(Tx1) )
```

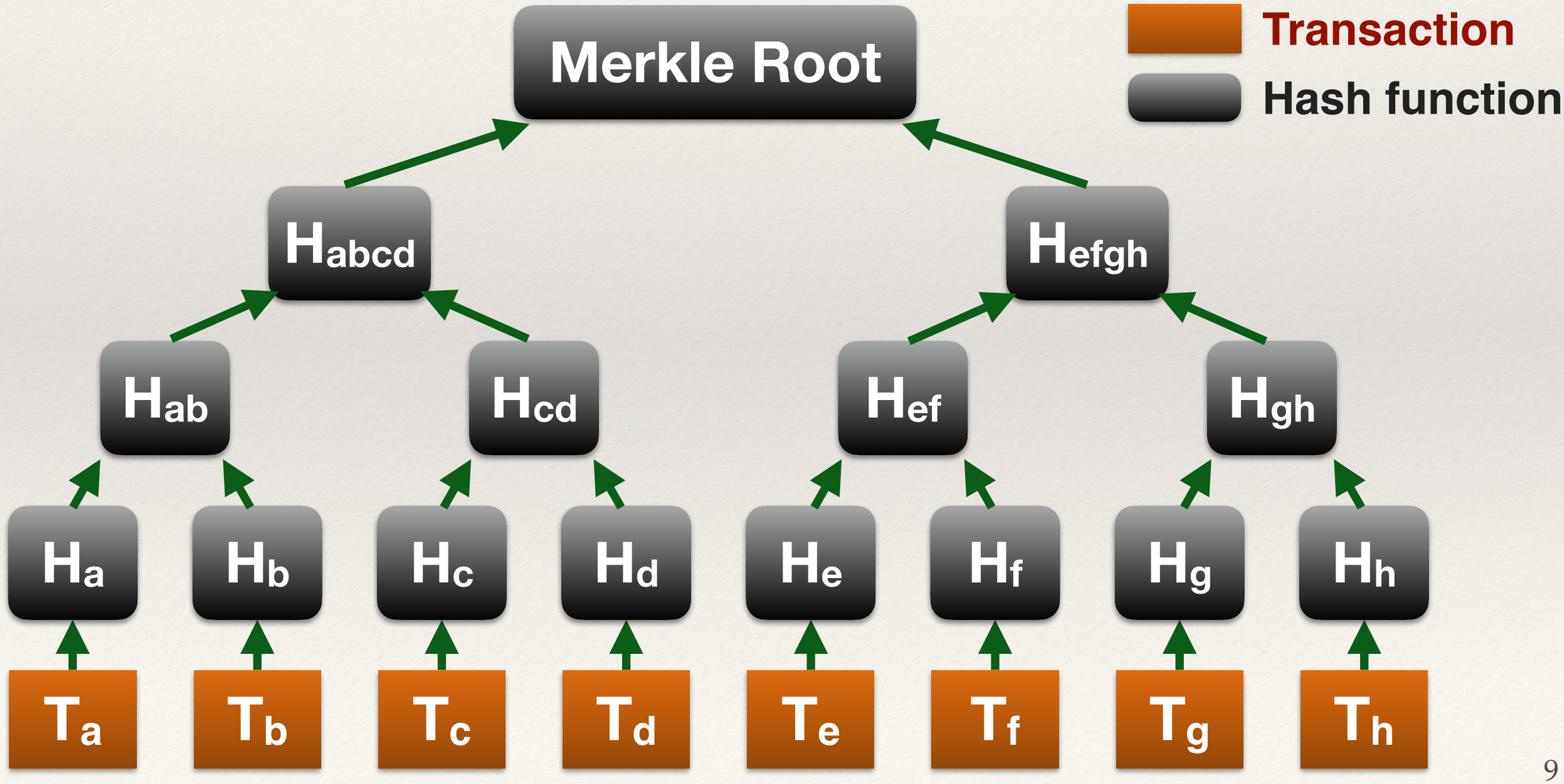
```
HAB = SAH256( SHA256( HA + HB ) )
```

- Each parent node is generated by a double SHA-256 function
  - Each child node is encrypted by SHA-256 **TWICE**
  - The encrypted results (**HA** and **HB**) are summed together
  - The sum is then encrypted **AGAIN** by SHA-256 **TWICE**

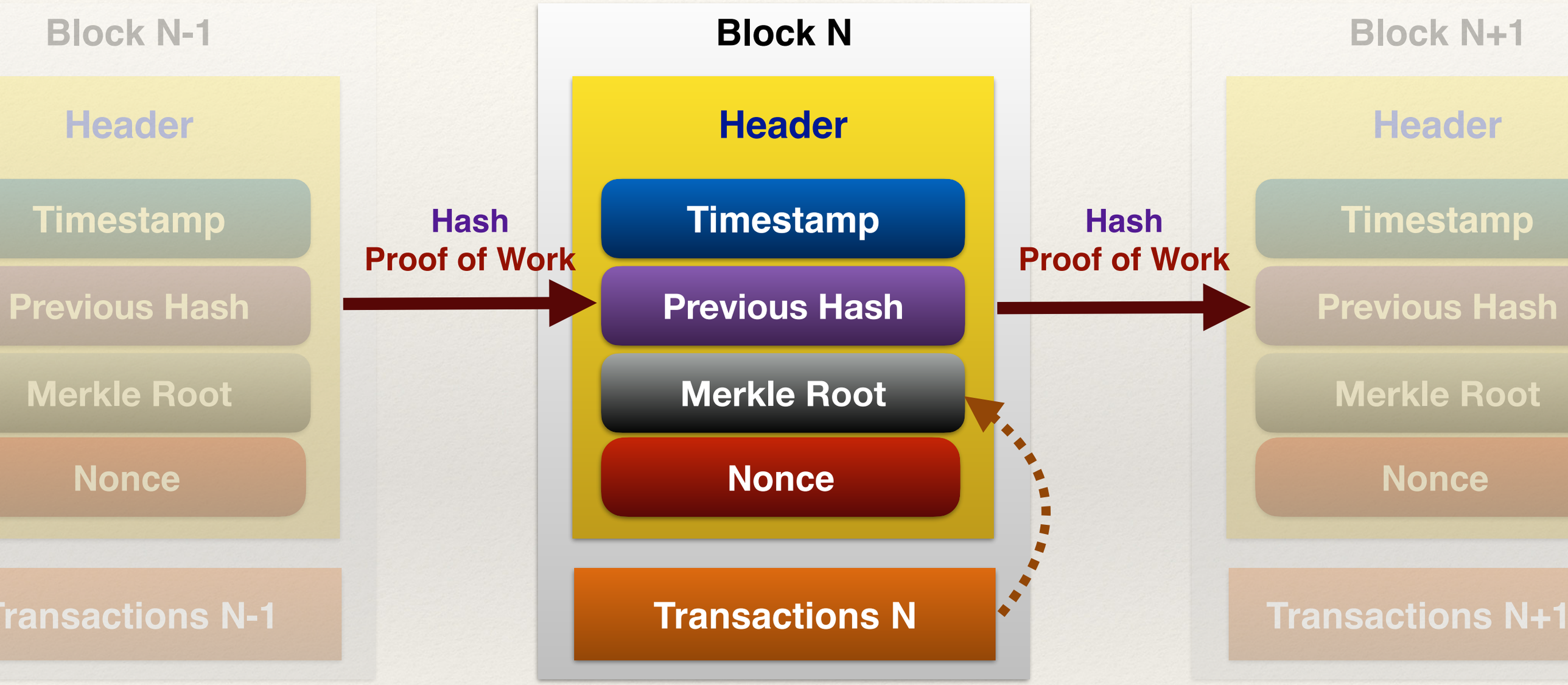
- The procedure is repeated again and again, until the root node is reached
  - The root node is called the **Merkle Root**

Introduction - Merkle Tree Structure

- A typically Merkle tree has the structure similar to the figure depicted below
  - Please note that the hash functions are **double SHA-256**



Introduction - Blockchain



- A blockchain is essentially a sequence of verified blocks
  - Each block contains the has value of its previous block's header
  - If something is modified, it will be immediately discovered

Introduction - Verification of Transactions

- **Who has to verify these transactions?**
  - The nodes in the bitcoin network
- **How do they verify the transactions?**
  - The transactions from each node are broadcast to the entire network
  - Each node collects transactions by some algorithms defined by the bitcoin protocol.
  - The transactions are collected for block encryption
- **How is the encrypted block added to the blockchain?**
  - The newly encrypted block is broadcast to the entire network
  - The miners in the network verifies if the new block is correct and conforms to the bitcoin protocol
    (e.g., Whether **block header's hash** < **Target Difficulty** or not)

Introduction - Target Difficulty

- The **Target Difficulty** is a value to determine how difficult it is to find out a proper hash value for the block header

- The **Target Difficult** conforms to the following rules
  - It is a 256-bit number, encoded by a field called "**nbits**" (in 4 bytes)
  - It is determined by the bitcoin network
  - It has **n** leading zeros (currently, **n** equals **72**)

- The **Target Difficulty** is adjusted such that a new block is computed every 10 minutes (on average)
  - The value of n is determined by the computing power of the entire bitcoin network

Introduction - Goal

- The primary goal of assignment 4 is to find out a proper hash value for a given block header
  - The hash value has to be less than the **Target Difficulty**
  - The **time you spent** on finding the hash value has to be accelerated
  - Use **any techniques** that you've learned in this class!

Outline

- Introduction
- Sequential Code
- Grading

Sequential Code

- **Procedure**

  I. Read input

  II. Calculate a merkle root from transactions

  III. Decode find out the **Target Difficulty**

  IV. Find out a proper **nonce**, such that the **hash value** of the block header satisfies the requirement

```
for nonce = 0X00000000 to 0Xffffffff
        calculate a hash value
        if the hash value < Target Difficulty
            done
```

Sequential Code - Header Extraction

- **Step 1** : Read input
- There are several fields in the block header that you have to extract
- Please familiar yourself with the fields extracted from the header
- An example is provided in the figure:
  - E.g., **example.cpp**

```
// **** read data *****
char version[9];
char prevhash[65];
char ntime[9];
char nbits[9];
int tx;
char *raw_merkle_branch;
char **merkle_branch;

getline(version, 9, fin);
getline(prevhash, 65, fin);
getline(ntime, 9, fin);
getline(nbits, 9, fin);
fscanf(fin, "%d\n", &tx);

raw_merkle_branch = new char [tx * 65];
merkle_branch = new char *[tx];
for(int i=0;i<tx;++i)
{
    merkle_branch[i] = raw_merkle_branch + i * 65;
    getline(merkle_branch[i], 65, fin);
    merkle_branch[i][64] = '\0';
}
```

Sequential Code - Header Fields

- A representative header fields of a block is depicted below
  - Please have a comparison with pages 6, 10, and 16
  - These information has to be extracted first before we carrying out the calculation

```
1                                                                  number of blocks
20000000                                                           version
0000000000000000000e2969067f2304a25004a2462e1114a262bc1740b722ff   prehash
5ad3327e                                                           time
1749500d                                                           nBits ( difficulty )
15                                                                 number of transactions
44763fed906f1a0e8b82e7497ac83856baa4b1c5621affdce04108a212018c4e
243e01b55beffbd3d62e80c5984670427bee29ea3951ac8d22cc30b5ad5f719d
cb85fee177c7a52cec1fe429b9d859dc33a6940c59ee2762376faef7462ed692
879e6462607890a15592055440e35cc7f6f2baa3d739526620ef56d08c1d99e6
3b16d0e9eea507dbe2e4e53df55fb7f3b58362a2ec2b9c5881430492cebe356d
ab06a72f0fce1cf5f2abae51d1dfb0e7a1704ed4110e6449161616898e54a855
14e839969b13e37da8e04c1e29a2a188a43326a2f5ee161b13a464885b97c44b
9182548ea7e756983ecfabec95042c4fdc3b023ae0919bb98f92dad9bf519acf   transactions
1cb1f779cc48c1510218eff73ac2f0cf8055adad700c240c1a82abac7246452c
d11b7b029d5590db2f59af768d1407c4688a1d667e87ae024147035c7cd05d4b
73039db4ad7e8f19d7e04a2e85fcb7149e4caf07199a14eb7905a22de84658af
0ba19a830c8e3673513176cc8ec7b020bc516f18db3b8adc8ee5389a6949bf0e
1d41e9e2688c3537566b32e810b20de1b20de0fc14911a1a11bf09670ebe5023
105dbf4799c27c8bef0871c228a682efb579fa652c4d793a4741390b97af413e
64186ef6f7343a53eb3bc100a0a8c920c39980e37c50c4870367dcc056d10941
```

Sequential Code - Merkle Root

- **Step 2**：Calculate a merkle root from transactions
  - The merle root is calculated by a function provided by the TA
  - You are encouraged to take a look of its implementation

- In **example.cpp**, you can find out a merkle root function as below:

```
unsigned char merkle_root[32];
calc_merkle_root(merkle_root, tx, merkle_branch);

printf("merkle root(little): ");
print_hex(merkle_root, 32);
printf("\n");

printf("merkle root(big):    ");
print_hex_inverse(merkle_root, 32);
printf("\n");
```

Sequential Code - Target Difficulty

- Step 3：Decode find out the **Target Difficulty**
  - The is decoded and calculated from a number called bits
  - The decode algorithm is already implemented by us (shown below)
  - If you are interested in that algorithm, please take a look of the following website: https://en.bitcoin.it/wiki/Difficulty

- Please refer to **example.cpp** and see how it is implemented

```
unsigned int exp = block.nbits >> 24;
unsigned int mant = block.nbits & 0xffffff;
unsigned char target_hex[32] = {};

unsigned int shift = 8 * (exp - 3);
unsigned int sb = shift / 8;
unsigned int rb = shift % 8;
```

Sequential Code - Nonce

- **Step 4** : Find out a proper **nonce**, such that the **hash value** of the block header satisfies the requirement
  - Try if you can figure out a way to parallelize and accelerate the code
- The implementation can be found in **example.cpp**

```
for(block.nonce=0x00000000; block.nonce<=0xffffffff;++block.nonce)
{
    //sha256d
    double_sha256(&sha256_ctx, (unsigned char*)&block, sizeof(block));
    if(block.nonce % 1000000 == 0)
    {
        printf("hash #%10u (big): ", block.nonce);
        print_hex_inverse(sha256_ctx.b, 32);
        printf("\n");
    }

    if(little_endian_bit_comparison(sha256_ctx.b, target_hex, 32) < 0)  // sha256_ctx < target_hex
    {
        printf("Found Solution!!\n");
        printf("hash #%10u (big): ", block.nonce);
        print_hex_inverse(sha256_ctx.b, 32);
        printf("\n\n");

        break;
    }
}
```

Sequential Code - SHA-256

- SHA-256 implementation for the sequential version can be found at:
  - sha256.h
  - sha256.c

- If you are interested in the implantation details, please take a look at the following website
  - https://en.wikipedia.org/wiki/SHA-2

Sequential Code - Location

- The sequential code can be found at:
  - **/home/ipc20/ta/hw4**

- How to run your program
  - **make**
  - srun <srun flag> **./example** [inputfile] [outputfile]

- We provide several testcases for you to test your program
  - The execution time of the testcases are given below:
    - **testcase/00/case00.in** : about 1 hr 15 min
    - **testcase/01/case01.in** : about 1 hr 15 min
    - **testcase/02/case02.in** : about 30 min
    - **testcase/03/case03.in** : about 11 min
  - Please keep in mind that your goal is to parallelize the mining algorithm and accelerating it
  - It has to be faster than the sequential version

Sequential Code - I/O Files

- Testcases can be found at
  - **/home/ipc20/ta/hw4/testcase**
- Input arguments can be found at:
  - **./testcase/00/case00.in**
  - To run the test files:
    srun <srun flag> **./example ./testcase/00/case00.in myout**
- The output file contains the **nonce** of the block
  - Please ensure that the generated hash value satisfies the requirement
    hash value < **Target Difficulty**
  - Please output your **nonce** in **little endian hex, lowercase**

Sequential Code - Optimization

- We encourage you to trace the source codes and optimize them
  - You are allowed to use **ANY TECHNIQUES** mentioned in the class
  - Please keep in mind that the GPU server only runs for 30 minutes
  - As a result, you have to figure out some methods to accelerate your codes
- If you don’t have any idea, you are very welcome to post questions or send emails to us
  - We will be more than happy to provide assistance

Outline

- Introduction
- Sequential Code
- Grading

Grading

- **Correctness (40%)**
- **Performance (40%)**
- **Report (20%)**

Grading Criteria

- **Correctness**
  - Please make sure to use **GPU** and verify your answer
- **Performance**
  - We will grade the performance of your assignments against the other students in the class
- **Report**
  - Please describe, in detail, how you parallelize your codes as well as your optimization methodology
- **Advanced CUDA skills**
  - You are welcome to use streaming, page-lock memory, asynchronous memory copy, or any other advanced skills
- **Others**
  - You will get credits if you are able to optimize the other parts of the source codes. Please justify your solutions and provide a detailed comparison **in your report**

Grading - Report

- In your report, please include the following parts
  - Your implementation
  - The parallelization and optimization techniques you used in your solution
  - Experiments of **various combinations of the number of blocks & threads** (at least **8** combinations) and **plot them with the figures**
  - Describe the **details** if you use advanced CUDA skills
  - If you optimize the other parts of your source codes, please demonstrate your **experimental results**. We **REQUIRE** you to justify your solutions so that we can give you credits.

Submission

- Please submit your hw4 to ILMS
  - cuda_miner.cu
  - sha256.cu
  - sha256.h
  - report.pdf
  - build.ninja
- **Please do not package them, directly upload the files to ILMS**

- Please make sure your **build.ninja** works properly and your program can run before submitting your assignment 4

Reminder

- Because we are doing hash, there may be a situation multiple nonce can satisfies the requirement
  hash value < **Target Difficulty**
- We accept all nonce satisfies the requirement, so don’t worry if your nonce isn’t same as our provided solutions
- We ensure that all the testcases have more than one solution
- Your build.ninja should build executable binary **cuda_miner**

Deadline

- The deadline of assignment 4 is **5/18 (Mon), 23:59 pm**
- Everyone is welcomed to ask questions on ILMS