“Half-Fast” Bitcoin Miner: Open-Source Bitcoin Mining with FPGA

Peter Xu (px2117)
Patrick Taylor (pat2138)
Ben Nappier (ben2113)
Matheus Candido (mcc2224)

Table of Contents

Introduction

Bitcoin is an open source peer-to-peer payment system introduced in 2009. The currency offers a reward to users who offer their computing power to verify and record transactions. This process is very extensive and a substantial amount of computational power is required. In this project we aimed to design a Bitcoin miner integrated to the main peer-to-peer network through a mining pool with FPGA hardware.

Bitcoin protocol

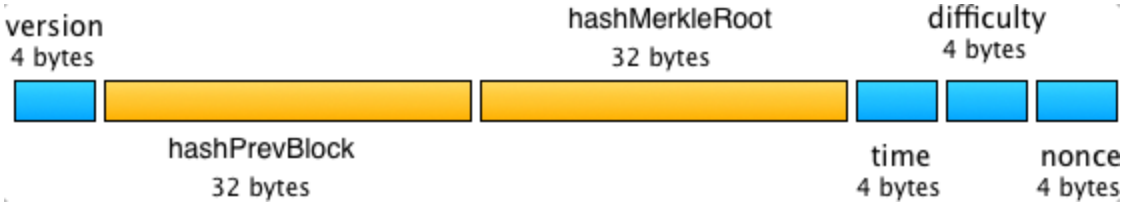
Users send bitcoins by broadcasting a digitally signed message to the network using whatever Bitcoin wallet software they have. Each person has a public and a private key, to respectively encrypt and decrypt data. The problem with digital transactions is that the payee can't verify if the previous owners did not double-spend a coin. The payee needs proof that by the time of that transaction, the majority of the nodes agreed that this is the first time the transaction is received. Instead of introducing a trusted central authority to check every transaction, to secure the network and overcome such problems, transactions are always verified by peers and stored in a public database called "block chain".

Transactions are sent to a selected number of people with computational power who will be rewarded for the interest in verifying transactions. These people are called miners, and before any approval, they must show a proof-of-work. A proof-of-work is a solution from running the algorithm SHA-256 on a set of transactions. The Bitcoin proof-of-work requires the hash of a block's header to have a certain number of zeros, known as the set difficulty. If the proof-of-work is below the difficulty, the solution is valid. In order to find a valid solution, SHA-256 is run on the data with a single number incremented until the solution is potentially found. This special number is known as the nonce value.

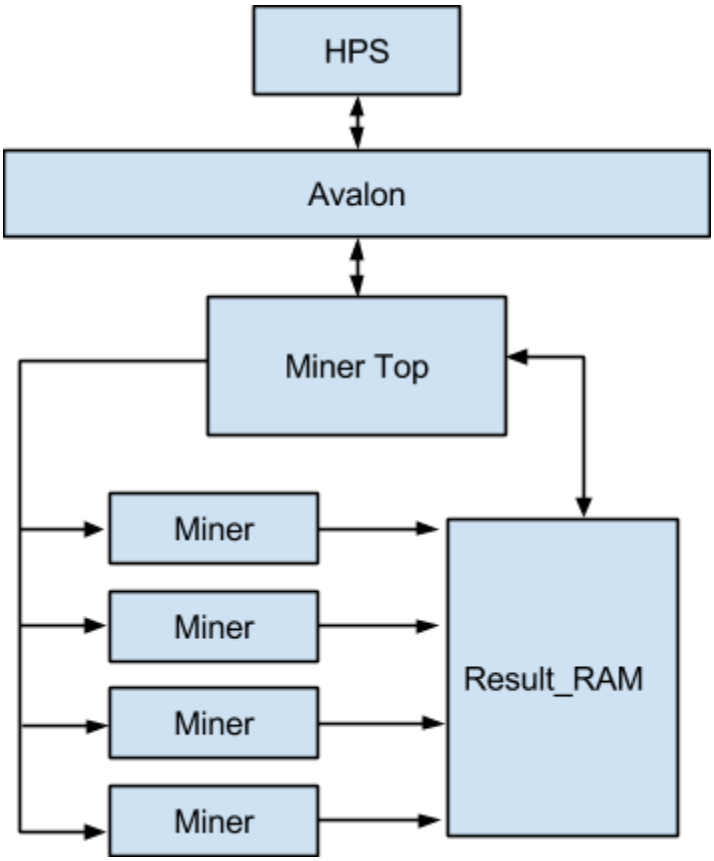
The FPGA will be doing the SHA-256 algorithm with different nonce values which is basically guessing encryptions for headers of Bitcoin blocks. These blocks have pending transactions and solving a puzzle validates the transactions. When a miner solves a block, it is rewarded with a fixed amount of bitcoins. In our case, the pool we participate will be rewarded if we find the solution.

Project design

Our "Half Fast" Bitcoin miner design utilizes SoCKit board to mine Bitcoins at a rate exceeding those of CPU/GPU-based systems. We use the getwork protocol to retrieve an 80 byte block header, consisting of a 4 byte version number, a 32 byte previous block header, a 32 byte Merkle Root of previous transactions, a 4 byte timestamp, a 4 byte difficulty, and a 4 byte nonce. Once this header is parsed from the Bitcoin network block retrieval process, the nonce-incremented header is sent through a double SHA-256 hash, and compared to the difficulty specified in the header data. Given our board's ability to run Linux, we were able to poll the Bitcoin network for blocks and mine the given blocks on the the FPGA, as well as run multiple iterations of the miner in parallel to increase the rate at which successful blocks are found.



High-level block diagram



The interface between software and hardware primarily sends work retrieved from the Bitcoin network to the Miner peripherals.The software will repeatedly probe the miner to see if it has solved the block. If it does, the software reads the solved nonce value and sends this result back to the network. If the Miner peripherals do not solve their work within a given time, they will be updated with additional work once the software receives a new block header from the network. The software parses and prepare the data to be hashed so that the only operations the hardware is concerned about are hashing and verifying the difficulty of the hashed data.

Each miner differs only in the initial nonce value at which it will start computing. Therefore, all of the miners share the same header data except for the nonce portion. When a new data block is available, the miner is informed and it enters the loading state, where it prepares to read in the new block header data. Once the peripheral receives the header data, it will keep working until it solves the block or it is told to work on new data. From the hardware to the software, the miner has a dedicated register which it sets if it has solved the block and obtained a valid nonce. The program

checks these registers regularly and if it detects that a miner has solved a block, it checks the corresponding registers that contain the valid nonce value.

Miner

The main function of the miner is to run the SHA256 algorithm on the 80 byte value of the Block header once and then hash that result again. Each time it will increment the nonce and do this again. In each iteration it will compare the new resulting 256 bit hash (or digest) with the difficulty specified in the block header. The difficulty is a minimum value the resulting hash must be set by the bitcoin network. This difficulty can be the thought of as the number of trailing zeros as mentioned before. If it is within the range of the difficulty, it sets the corresponding register set in “Results RAM” block (Fig. 1). The first 32 bits hold the solved nonce value and the last bit is a flag indicating that the corresponding miner has solved the block.

The miner will pad the value that is to be hashed such that it is a multiple of 512 bits per the SHA256 algorithm. Since the nonce is the only value altered in the header upon each iteration, the first 64 bytes of the header will remain the same and so only needs to be hashed once. The hash of this is called the “midstate” value and it is stored for reuse in subsequent iterations. The remaining 16 byte portion, which contains the nonce, changes on each iteration since different nonce values are tested. These 16 bytes will be hashed starting from the midstate value.

Hardware Implementation

Our miner has three states: Idle, Loading, and Stop. The FSM is shown below. In the Idle state, it waits for new header data input from the driver. This is triggered by asserting a “start” register which takes it to the loading state. In the Loading state, the software writes the block header information to the miner. Once it finishes loading the last byte, a “load_done” register is asserted to bring it back to the Idle state where it waits for the miner to solve for a nonce. When a miner has solved a nonce, it asserts a “ticket” register to bring it to the Stop state. In this state, the miner latches the nonce output from the miner so that the solved nonce remains available to be read until the new header data is read into the system.

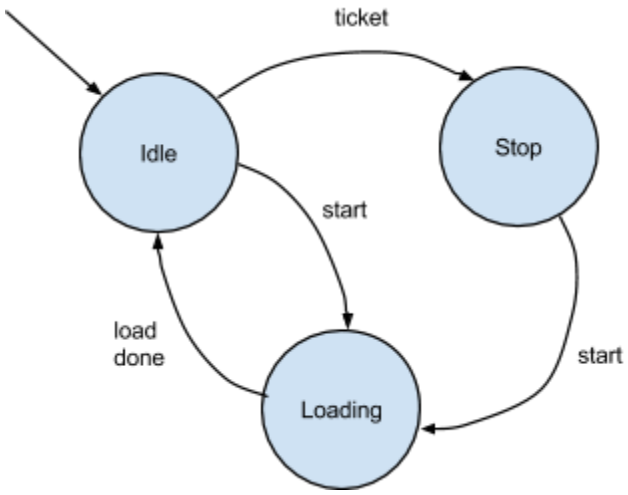


Fig. 1 FSM

Miner_Top (Fig. 2) is the top level module of the system. The driver writes the header block data into the a register set called “header_buffer.” Miner_Top instantiates multiple miner modules (fpgaminer_top) that work in parallel to solve the block. Each miner starts from a different nonce value which is determined by a certain range offset. Miner0 starts at the nonce supplied from the header data, Miner1 starts from nonce + range, Miner2 starts from nonce + 2*range, etc. These starting nonces are calculated prior to the start of the mining process and stored in “Nonce RAM” which is a register file. Each nonce from the Nonce RAM feeds the respective miner. Since the rest of the header input data is the same for all miners, header_buffer feeds into all of the miners.

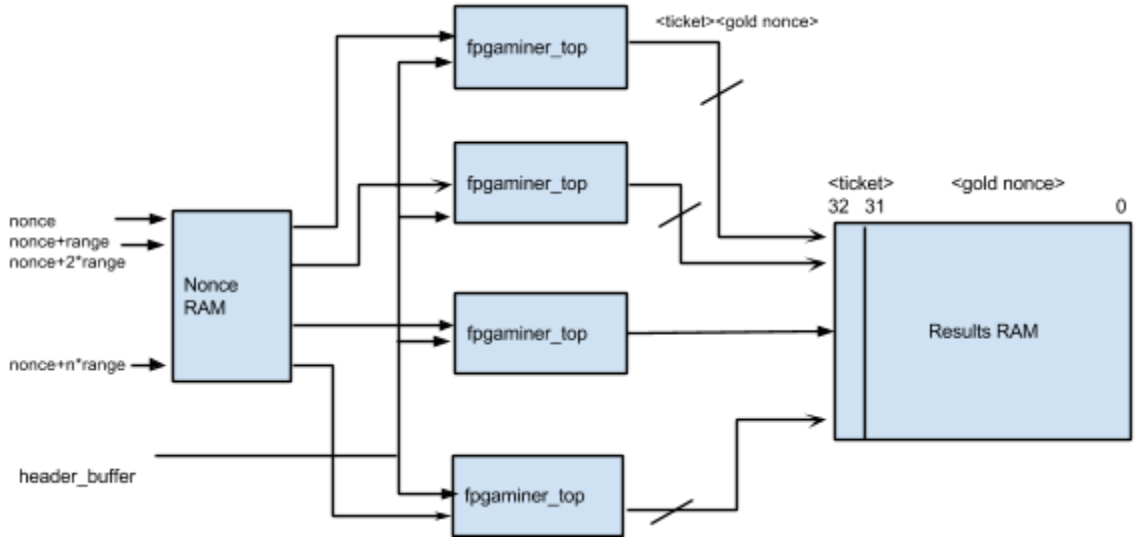


Fig. 2. Top level Miner_Top

The output of each miner is a 33-bit register set which feeds into a register file called “Results RAM.” The first 32 bits represent the nonce value solved by each respective miner and the last bit is a flag indicating that the miner has solved the block.

To load new header block data into the system, a “start” register is set by the driver which clears all of the values in header_buffer, Nonce RAM, and Results RAM and prepares the system

for loading. The start register was used over a reset signal because allowed the system to be reset without physically pushing the reset button and it was easier to write to a register than to send a reset signal from the driver.

The module below was the actual miner. We used an open source miner that was implemented on the SoCKit board [1]. Header data is passed from Miner_Top through header_buffer and is stored in state and data register sets. State holds the midstate initially and data holds the rest of the data input, including the nonce. Since data is only 16 bytes and it needs to be 64 bytes in order to be processed by SHA256, the miner pads it with the appropriate bits until it is 64 bytes. The miner consists of two serial SHA256 modules which performed the two SHA256 hashes as specified in the Mining algorithm. State and Data are passed to the first SHA256 blocks. While the hashes are being performed, the nonce is incremented and the new nonce replaces the previous nonce in Data. Thus the miner is repeatedly testing different nonce values until it finds one that solves the block. When the second SHA256 block completes, its hash result is compared with the difficulty and if the number of trailing zeros match, the corresponding nonce that computed that hash is latched to nonce_out. This is fed back to Miner_Top where it is stored in Result RAM in the corresponding location.

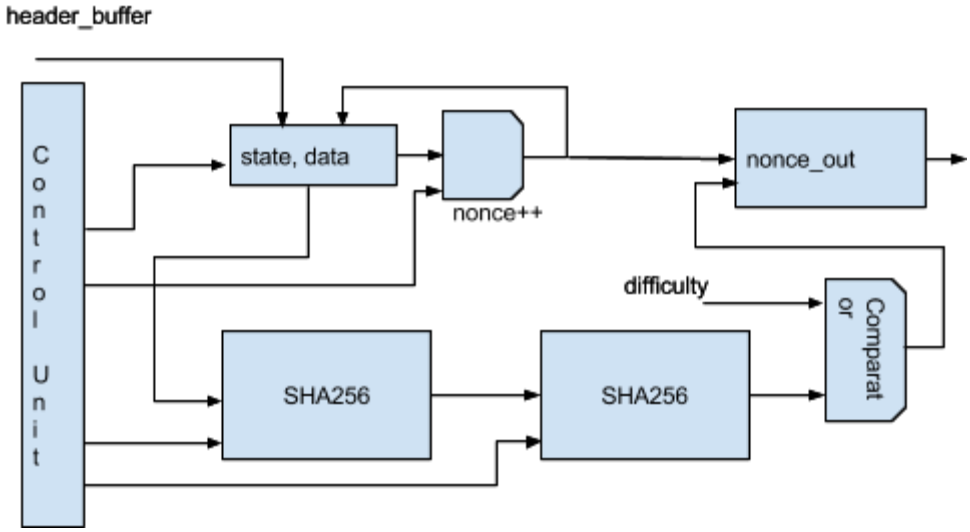


Fig. 3. FPGA Miner

The module on Fig.4 performs the SHA256 computation. SHA256 takes in a message input on which it computes. In this case, the message is state and data. K is a set of constants that are needed in the algorithm. The modules in the middle “e0, e1, ch, maj, s0, s1” are a set of bitwise operations that operate on a subset of state and data. They are used as part of a Compression function in SHA256, which loops through a 512-bit chunk and calculates a set of numbers that will eventually be combined to form the hash. The number of instantiations of these blocks is dependent on a LOOP parameter which determines how much Compression loop is unrolled. A large LOOP parameter partially unrolls the loop, utilizes less space, but executes the algorithm more slowly. Conversely, a small LOOP parameter further unrolls the loop at the cost of more space but faster execution.

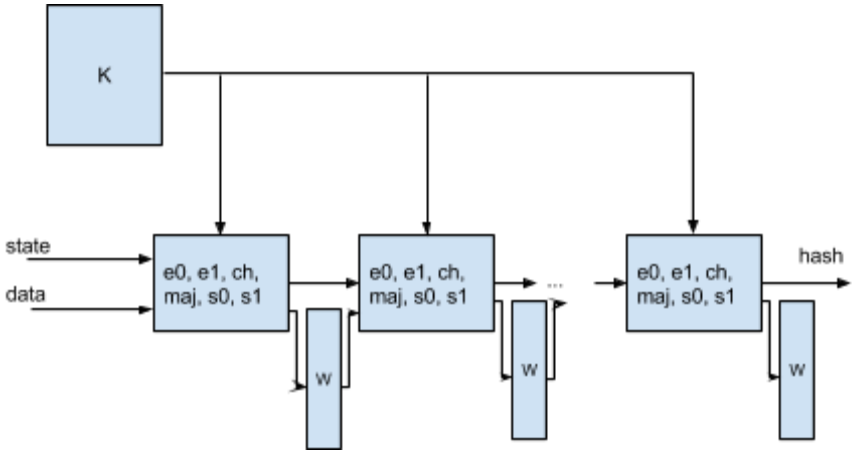


Fig. 4. SHA_Transform

The W blocks are a set of intermediate registers used in the Compression loop. Each subsequent iteration of the loop depends on the W values in the previous iteration. State is modified and the result of each “e0, e1, ch, maj, s0, s1” module is accrued in state during each intermediate step. In the final step, the values in state are concatenated to form the hash result.

Software Implementation

Our software consists of both a userspace that retrieves header information from the Bitcoin network and a driver that communicates said information to the hardware. The driver is a simple modification of the VGA driver provided in Lab 3 for use in our implementation; this includes a change in the number of addresses from 8 to 100, as well as modifying the limit on digit size from 8 to 2^8. The userspace is a C program entitled getwork.c that utilizes the ioctl, curl, and jansson code libraries, as well as functions included in the modified vga_led drivers, to make function calls to the network for header information. The program primarily consists of two threads, one waiting for new work from the network, and one listening to the hardware for a solved block data to send to a proof a work to the network.

Software and Hardware

Our miner operates in three stages. The first stage involves a software driver written in C communicating directly to the Bitcoin network through a series of JSON calls. These JSON calls retrieve data to be used in the miner: “data” is the 512-bit input containing the trailing 16-bytes of the 80-byte header padded as required by the Bitcoin algorithm, while “midstate” contains the 256-bit output of hashing the first 64-bytes of the header to be concatenated with the hashed “data” as input to the second SHA-256 hash (note: we also retrieve “target”, the difficulty specified by the network for the block, but this is currently hardcoded in our miner). Once “data” and “midstate” are parsed from these JSON calls, they are concatenated to become a 768-bit input that is passed to our miner.

During our second stage, we perform ninety-six 8-bit write calls, using a sequential addressing scheme, to pass all information to our miner. Once all information is safely written to hardware registers, we pass information to our four miners, operating in parallel only differing in the value used for the 32-bit nonce (incremented by 1 per miner and increased by a factor of 4 after every hash). After the double SHA-256 hash is performed, its result is compared to the target consisting of a series of leading hexadecimal zeros and trailing ones (for our data, 8 and 56

respectively). If the hashed output is less than the specified target, a “golden ticket” flag is raised and the 32-bit nonce used to produce the hash is written out to the software with a one-bit flag.

The third stage occurs in parallel with the second stage; while the mining is ongoing, the software is constantly performing a “long-poll” of the network to pass any changes in “data” or “midstate” to the hardware. If the software receives a new set of data from the mining pool, the miner is reset and begins working on the next block. This should occur approximately every ten minutes, as specified by the Bitcoin algorithm. In the rare instance that our miner is able to successfully pass a “golden ticket” within this ten-minute window, our software replaces the original nonce with the “golden nonce,” and performs additional JSON calls to send this information to the Bitcoin network and check for approval from the network. If successful, this will earn a share of Bitcoin profits in our wallet and a new block from the network.

Testing Methodology

We made extensive use of ModelSim and the System Console when testing our hardware implementation.

ModelSim

We used ModelSim to test the Open Source Miner and make sure that it would function properly. The code for it came along with sample block header data and the solved nonce value which we used as a reference. We used ModelSim to generate the values for the header data, write signals, etc. We also used scripts to automate the testing which made things convenient and easy for us. One thing we could have done better was write better and more dynamic scripts that would be flexible and easy to modify for inputting large data sets. For instance, we initially tested it by asserting the wires for header_buffer to the value we wanted to test. When we modified the interface so that we had to load header data as a sequence of bytes, it became infeasible to specify each byte value and the time at which it would appear manually. However, we did not want to spend too much time just learning ModelSim so we moved on once we verified the functionality of the miner.

System Console

We used the System Console to verify the correctness of our hardware once it was loaded into the board. This was especially useful for debugging and testing when the software portions were not yet complete. We created a series of scripts that would write both incorrect and correct data to the miner. This allowed us to verify that when incorrect data was sent to the miner, there would be no nonce values or flags indicating that the block has been solved. Conversely, when correct data was sent, we were able to verify that a miner had solved the block and output the solved nonce value. We created read scripts that would probe the peripheral’s registers and verify that it was behaving properly.

While the System Console was invaluable, it was not very practical for incremental debugging. Recompiling and retesting was very time consuming for small changes. Using something like SignalTap may have been more efficient but we did not have the background for it or the time to learn it.

Performance Comparison

Product	Hash rate [Mhash/s]
Avnet Spartan-6 LX150T Development Kit	**100**
Bitcoin Dominator X5000	**100**
BitForce SHA256 Single	**832**
Butterflylabs Mini Rig	**25,200**
Digilent Nexys 2 500K	**5**
Icarus	**380**
KnCMiner Mars	**6,000**
Lancelot	**400**
ModMiner Quad	**800**
Terasic DE2-115	**80**
X6500 FPGA Miner	**400**
ZTEX USB-FPGA Module 1.15b	**90**
ZTEX USB-FPGA Module 1.15x	**215**

ZTEX USB-FPGA Module 1.15y	**860**
Half-Fast	**22.69**

Source: https://en.bitcoin.it/wiki/Mining_hardware_comparison#FPGA

Our miner Half-Fast performs 22.69 MHashes/sec. We determined this by timing exactly how long it took for a miner to begin from a nonce value of 0 to the solved value of 0x0e33337a (this was the correct nonce value for our specific reference sample header data). This took 1 minute 24 seconds and dividing the total nonces (0x0e33337a = 238236538) by the time difference, we get the hash rate for one miner. Then we multiplied this by 8 since we had 8 miners and got our approximate result of 22.69 MHashes/sec. In the table above, we compare our performance to other FPGA Miner performance benchmarks from the Bitcoin Wiki site. Our miner is slower than most of the benchmarks, but it should be noted that our mining system is not optimized and has made the full utilization of the SoCKit board resources.

Milestones

Milestone 1	**- Create the peer to peer software** Connecting to the bitcoin network via software and obtain/parse block header information **- Implement SHA256 in software to understand the algorithm** **- Simulate and verify behavior of the Open Source Miner**	**Accomplished**
Milestone 2	**- Implement and verify functionality of the Miner system on the board**	**Accomplished**
Milestone 3	**- Add support for Bitcoin Mining pools** **- Integrate the two systems** Integrate software and hardware. Pass the block header information to the miner and send the results to the bitcoin network/mining pool **- Implement multiple miners in parallel**	**Accomplished**

Lessons Learned

- Learn how to use ModelSim and the System Console.
- Start from Lab3 skeleton code.
- Underestimated the initial planning/design process. Come up with a good and well thought out preliminary hardware design. Identify what are the requirements, what signal inputs are needed, and consult Professor Edwards or adviser directly before implementing. Define and decide on the Memory Map Interface early on and consult with the advisers whether they will meet your requirement.
- Work on hardware and software in parallel

Contributions

- Peter
    - Hardware Implementation
    - Parallelization
    - Simulation and testing hardware
- Matheus
    - Understanding Bitcoin protocol
    - Software Implementation
- Patrick
    - Hardware Implementation
    - Software Implementation
    - Simulation and Testing Hardware
- Ben
    - Software Implementation
    - Network setup
    - General IO testing

References

[1] Open Source Bitcoin Miner. https://github.com/gardintrapp/Open-Source-FPGA-Bitcoin-Miner

[2] Nielsen, Michael. How the Bitcoin Protocol actually works. Retrieved on May 09, 2014 from http://goo.gl/9TWn6G.

[3] Nakamoto, Satoshi. Bitcoin: A Peer-to-Peer Electronic Cash System. Retrieved on May 09, 2014 from https://bitcoin.org/bitcoin.pdf.

[4] Pacia, Chris. Bitcoin Mining Explained Like You're Five: Part 2 – Mechanics. Retrieved on May 09, 2014 from http://goo.gl/0kcFtt.

Source code

*We used Verilog because the Open Source Miner was initially implemented in Verilog and so we wanted to stick to the same language for consistency.

miner_top.v

```
`define IDX32(x) (((x)+1)*(32)-1):((x)*(32))
`define IDX8(x) (((x)+1)*(8)-1):((x)*(8))
`define NONCEIDX 383:352

module miner_top(
        input clk,
        input reset,
        input write,
        input read,
        input chipselect,
        input [7:0] writedata,
        input [7:0] address,

        output reg [7:0] readdata);

        reg [767:0] header_buffer = 768'd0;

        // State variables
        reg loading = 1'b0;
        reg load_done = 1'b0;
        reg start = 1'b0;
        reg stop = 1'b0;
        reg read_gold_nonce = 1'b1;
        reg ticket = 1'b0;

        // Parallelization
        parameter mctr = 8;
                        //number of miners
        localparam range = 32'd500000000;           //the range of nonce values for
each miner

        // Wires to connect miner outputs to the result ram
        wire [32:0] nonce_out_a0;
        wire [32:0] nonce_out_a1;
        wire [32:0] nonce_out_a2;
        wire [32:0] nonce_out_a3;
        wire [32:0] nonce_out_a4;
        wire [32:0] nonce_out_a5;
        wire [32:0] nonce_out_a6;
        wire [32:0] nonce_out_a7;

        // Ram for holding the nonce input and nonce results for each miner
        reg [31:0] nonce_ram[0:7];
        reg [32:0] result_ram[0:7];

        // Instantiate multiple miners
        fpgaminer_top miner0 (clk, {header_buffer[767:384], nonce_ram[0],
header_buffer[351:0]}, load_done, nonce_out_a0);
```

```
	fpgaminer_top miner1 (clk, {header_buffer[767:384], nonce_ram[1],
header_buffer[351:0]}, load_done, nonce_out_a1);
	fpgaminer_top miner2 (clk, {header_buffer[767:384], nonce_ram[2],
header_buffer[351:0]}, load_done, nonce_out_a2);
	fpgaminer_top miner3 (clk, {header_buffer[767:384], nonce_ram[3],
header_buffer[351:0]}, load_done, nonce_out_a3);
	fpgaminer_top miner4 (clk, {header_buffer[767:384], nonce_ram[4],
header_buffer[351:0]}, load_done, nonce_out_a4);
	fpgaminer_top miner5 (clk, {header_buffer[767:384], nonce_ram[5],
header_buffer[351:0]}, load_done, nonce_out_a5);
	fpgaminer_top miner6 (clk, {header_buffer[767:384], nonce_ram[6],
header_buffer[351:0]}, load_done, nonce_out_a6);
	fpgaminer_top miner7 (clk, {header_buffer[767:384], nonce_ram[7],
header_buffer[351:0]}, load_done, nonce_out_a7);

	always @(posedge clk) begin

		// Establish the nonce values from which each miner will start
		nonce_ram[0] <= header_buffer[`NONCEIDX];
		nonce_ram[1] <= header_buffer[`NONCEIDX] + range;
		nonce_ram[2] <= header_buffer[`NONCEIDX] + 2*range;
		nonce_ram[3] <= header_buffer[`NONCEIDX] + 3*range;
		nonce_ram[4] <= header_buffer[`NONCEIDX] + 4*range;
		nonce_ram[5] <= header_buffer[`NONCEIDX] + 5*range;
		nonce_ram[6] <= header_buffer[`NONCEIDX] + 6*range;
		nonce_ram[7] <= header_buffer[`NONCEIDX] + 7*range;


		if(start && !loading) begin
			loading <= 1'b1;
			load_done <= 1'b0;
			start <= 1'b0;
			header_buffer <= 768'd0; //Reset all including the nonce
and golden nonce

			result_ram[0] <= 33'd0;
			nonce_ram[0] <= 32'd0;

			result_ram[1] <= 33'd0;
			nonce_ram[1] <= 32'd0;

			result_ram[2] <= 33'd0;
			nonce_ram[2] <= 32'd0;

			result_ram[3] <= 33'd0;
			nonce_ram[3] <= 32'd0;

			result_ram[4] <= 33'd0;
			nonce_ram[4] <= 32'd0;

			result_ram[5] <= 33'd0;
			nonce_ram[5] <= 32'd0;

			result_ram[6] <= 33'd0;
			nonce_ram[6] <= 32'd0;

			result_ram[7] <= 33'd0;
			nonce_ram[7] <= 32'd0;
```

```
                read_gold_nonce = 1'b1;
                stop <= 1'b0;
        end
        else if(write && !loading && chipselect) begin
                if(address == 7'd102)
                        start <= writedata[0];
        end
        else if(write && loading && chipselect) begin
                case(address)
                        8'd0: header_buffer[7:0] <= writedata;
                        8'd1: header_buffer[15:8] <= writedata;
                        8'd2: header_buffer[23:16] <= writedata;
                        8'd3: header_buffer[31:24] <= writedata;
                        8'd4: header_buffer[39:32] <= writedata;
                        8'd5: header_buffer[47:40] <= writedata;
                        8'd6: header_buffer[55:48] <= writedata;
                        8'd7: header_buffer[63:56] <= writedata;
                        8'd8: header_buffer[71:64] <= writedata;
                        8'd9: header_buffer[79:72] <= writedata;
                        8'd10: header_buffer[87:80] <= writedata;
                        8'd11: header_buffer[95:88] <= writedata;
                        8'd12: header_buffer[103:96] <= writedata;
                        8'd13: header_buffer[111:104] <= writedata;
                        8'd14: header_buffer[119:112] <= writedata;
                        8'd15: header_buffer[127:120] <= writedata;
                        8'd16: header_buffer[135:128] <= writedata;
                        8'd17: header_buffer[143:136] <= writedata;
                        8'd18: header_buffer[151:144] <= writedata;
                        8'd19: header_buffer[159:152] <= writedata;
                        8'd20: header_buffer[167:160] <= writedata;
                        8'd21: header_buffer[175:168] <= writedata;
                        8'd22: header_buffer[183:176] <= writedata;
                        8'd23: header_buffer[191:184] <= writedata;
                        8'd24: header_buffer[199:192] <= writedata;
                        8'd25: header_buffer[207:200] <= writedata;
                        8'd26: header_buffer[215:208] <= writedata;
                        8'd27: header_buffer[223:216] <= writedata;
                        8'd28: header_buffer[231:224] <= writedata;
                        8'd29: header_buffer[239:232] <= writedata;
                        8'd30: header_buffer[247:240] <= writedata;

                        8'd31: header_buffer[255:248] <= writedata;
                        8'd32: header_buffer[263:256] <= writedata;
                        8'd33: header_buffer[271:264] <= writedata;
                        8'd34: header_buffer[279:272] <= writedata;
                        8'd35: header_buffer[287:280] <= writedata;
                        8'd36: header_buffer[295:288] <= writedata;
                        8'd37: header_buffer[303:296] <= writedata;
                        8'd38: header_buffer[311:304] <= writedata;
                        8'd39: header_buffer[319:312] <= writedata;
                        8'd40: header_buffer[327:320] <= writedata;
                        8'd41: header_buffer[335:328] <= writedata;
                        8'd42: header_buffer[343:336] <= writedata;
                        8'd43: header_buffer[351:344] <= writedata;
                        8'd44: header_buffer[359:352] <= writedata;
                        8'd45: header_buffer[367:360] <= writedata;
                        8'd46: header_buffer[375:368] <= writedata;
                        8'd47: header_buffer[383:376] <= writedata;
                        8'd48: header_buffer[391:384] <= writedata;
                        8'd49: header_buffer[399:392] <= writedata;
```

```
                        8'd50: header_buffer[407:400] <= writedata;
                        8'd51: header_buffer[415:408] <= writedata;
                        8'd52: header_buffer[423:416] <= writedata;
                        8'd53: header_buffer[431:424] <= writedata;
                        8'd54: header_buffer[439:432] <= writedata;
                        8'd55: header_buffer[447:440] <= writedata;
                        8'd56: header_buffer[455:448] <= writedata;
                        8'd57: header_buffer[463:456] <= writedata;
                        8'd58: header_buffer[471:464] <= writedata;
                        8'd59: header_buffer[479:472] <= writedata;
                        8'd60: header_buffer[487:480] <= writedata;

                        8'd61: header_buffer[495:488] <= writedata;
                        8'd62: header_buffer[503:496] <= writedata;
                        8'd63: header_buffer[511:504] <= writedata;
                        8'd64: header_buffer[519:512] <= writedata;
                        8'd65: header_buffer[527:520] <= writedata;
                        8'd66: header_buffer[535:528] <= writedata;
                        8'd67: header_buffer[543:536] <= writedata;
                        8'd68: header_buffer[551:544] <= writedata;
                        8'd69: header_buffer[559:552] <= writedata;
                        8'd70: header_buffer[567:560] <= writedata;
                        8'd71: header_buffer[575:568] <= writedata;
                        8'd72: header_buffer[583:576] <= writedata;
                        8'd73: header_buffer[591:584] <= writedata;
                        8'd74: header_buffer[599:592] <= writedata;
                        8'd75: header_buffer[607:600] <= writedata;
                        8'd76: header_buffer[615:608] <= writedata;
                        8'd77: header_buffer[623:616] <= writedata;
                        8'd78: header_buffer[631:624] <= writedata;
                        8'd79: header_buffer[639:632] <= writedata;
                        8'd80: header_buffer[647:640] <= writedata;
                        8'd81: header_buffer[655:648] <= writedata;
                        8'd82: header_buffer[663:656] <= writedata;
                        8'd83: header_buffer[671:664] <= writedata;
                        8'd84: header_buffer[679:672] <= writedata;
                        8'd85: header_buffer[687:680] <= writedata;
                        8'd86: header_buffer[695:688] <= writedata;
                        8'd87: header_buffer[703:696] <= writedata;
                        8'd88: header_buffer[711:704] <= writedata;
                        8'd89: header_buffer[719:712] <= writedata;
                        8'd90: header_buffer[727:720] <= writedata;
                        8'd91: header_buffer[735:728] <= writedata;
                        8'd92: header_buffer[743:736] <= writedata;
                        8'd93: header_buffer[751:744] <= writedata;
                        8'd94: header_buffer[759:752] <= writedata;
                        8'd95: begin
                                                        header_buffer[767:760] <=
writedata;
                                                        loading <= 1'b0;
                                                        load_done <= 1'b1;
                                                end
                endcase
        end     // end of if(write && loading)
        else if (read && chipselect && !loading) begin
                case (address)
                        8'd0: readdata <= header_buffer[7:0];
                        8'd1: readdata <= header_buffer[15:8];
                        8'd2: readdata <= header_buffer[23:16];
                        8'd3: readdata <= header_buffer[31:24];
```

```
                8'd4: readdata <= header_buffer[39:32];
                8'd5: readdata <= header_buffer[47:40];
                8'd6: readdata <= header_buffer[55:48];
                8'd7: readdata <= header_buffer[63:56];
                8'd8: readdata <= header_buffer[71:64];
                8'd9: readdata <= header_buffer[79:72];
                8'd10: readdata <= header_buffer[87:80];
                8'd11: readdata <= header_buffer[95:88];
                8'd12: readdata <= header_buffer[103:96];
                8'd13: readdata <= header_buffer[111:104];
                8'd14: readdata <= header_buffer[119:112];
                8'd15: readdata <= header_buffer[127:120];
                8'd16: readdata <= header_buffer[135:128];
                8'd17: readdata <= header_buffer[143:136];
                8'd18: readdata <= header_buffer[151:144];
                8'd19: readdata <= header_buffer[159:152];
                8'd20: readdata <= header_buffer[167:160];
                8'd21: readdata <= header_buffer[175:168];
                8'd22: readdata <= header_buffer[183:176];
                8'd23: readdata <= header_buffer[191:184];
                8'd24: readdata <= header_buffer[199:192];
                8'd25: readdata <= header_buffer[207:200];
                8'd26: readdata <= header_buffer[215:208];
                8'd27: readdata <= header_buffer[223:216];
                8'd28: readdata <= header_buffer[231:224];
                8'd29: readdata <= header_buffer[239:232];
                8'd30: readdata <= header_buffer[247:240];

                8'd31: readdata <= header_buffer[255:248];
                8'd32: readdata <= header_buffer[263:256];
                8'd33: readdata <= header_buffer[271:264];
                8'd34: readdata <= header_buffer[279:272];
                8'd35: readdata <= header_buffer[287:280];
                8'd36: readdata <= header_buffer[295:288];
                8'd37: readdata <= header_buffer[303:296];
                8'd38: readdata <= header_buffer[311:304];
                8'd39: readdata <= header_buffer[319:312];
                8'd40: readdata <= header_buffer[327:320];
                8'd41: readdata <= header_buffer[335:328];
                8'd42: readdata <= header_buffer[343:336];
                8'd43: readdata <= header_buffer[351:344];
                8'd44: readdata <= header_buffer[359:352];
                8'd45: readdata <= header_buffer[367:360];
                8'd46: readdata <= header_buffer[375:368];
                8'd47: readdata <= header_buffer[383:376];
                8'd48: readdata <= header_buffer[391:384];
                8'd49: readdata <= header_buffer[399:392];
                8'd50: readdata <= header_buffer[407:400];
                8'd51: readdata <= header_buffer[415:408];
                8'd52: readdata <= header_buffer[423:416];
                8'd53: readdata <= header_buffer[431:424];
                8'd54: readdata <= header_buffer[439:432];
                8'd55: readdata <= header_buffer[447:440];
                8'd56: readdata <= header_buffer[455:448];
                8'd57: readdata <= header_buffer[463:456];
                8'd58: readdata <= header_buffer[471:464];
                8'd59: readdata <= header_buffer[479:472];
                8'd60: readdata <= header_buffer[487:480];

                8'd61: readdata <= header_buffer[495:488];
```

```
                            8'd62: readdata <= header_buffer[503:496];
                            8'd63: readdata <= header_buffer[511:504];
                            8'd64: readdata <= header_buffer[519:512];
                            8'd65: readdata <= header_buffer[527:520];
                            8'd66: readdata <= header_buffer[535:528];
                            8'd67: readdata <= header_buffer[543:536];
                            8'd68: readdata <= header_buffer[551:544];
                            8'd69: readdata <= header_buffer[559:552];
                            8'd70: readdata <= header_buffer[567:560];
                            8'd71: readdata <= header_buffer[575:568];
                            8'd72: readdata <= header_buffer[583:576];
                            8'd73: readdata <= header_buffer[591:584];
                            8'd74: readdata <= header_buffer[599:592];
                            8'd75: readdata <= header_buffer[607:600];
                            8'd76: readdata <= header_buffer[615:608];
                            8'd77: readdata <= header_buffer[623:616];
                            8'd78: readdata <= header_buffer[631:624];
                            8'd79: readdata <= header_buffer[639:632];
                            8'd80: readdata <= header_buffer[647:640];
                            8'd81: readdata <= header_buffer[655:648];
                            8'd82: readdata <= header_buffer[663:656];
                            8'd83: readdata <= header_buffer[671:664];
                            8'd84: readdata <= header_buffer[679:672];
                            8'd85: readdata <= header_buffer[687:680];
                            8'd86: readdata <= header_buffer[695:688];
                            8'd87: readdata <= header_buffer[703:696];
                            8'd88: readdata <= header_buffer[711:704];
                            8'd89: readdata <= header_buffer[719:712];
                            8'd90: readdata <= header_buffer[727:720];
                            8'd91: readdata <= header_buffer[735:728];
                            8'd92: readdata <= header_buffer[743:736];
                            8'd93: readdata <= header_buffer[751:744];
                            8'd94: readdata <= header_buffer[759:752];
                            8'd95: readdata <= header_buffer[767:760];

                            //load state
                            8'd101: begin
                                                readdata[0] <= loading;
                                                readdata[1] <= load_done;
                                                readdata[7:2] <=
5'b000000;
                                            end
                            //start state
                            8'd102: begin
                                                readdata[0] <= start;
                                                readdata[1] <= stop;
                                                readdata[7:2] <=
5'b000000;
                                            end
                            //read_gold_nonce
                            8'd103: begin
                                                readdata[0] <=
read_gold_nonce;
                                            end

                            //nonce_ram
                            8'd104: readdata <= nonce_ram[0][`IDX8(0)];
                            8'd105: readdata <= nonce_ram[0][`IDX8(1)];
                            8'd106: readdata <= nonce_ram[0][`IDX8(2)];
                            8'd107: readdata <= nonce_ram[0][`IDX8(3)];
```

```
                        8'd108: readdata <= nonce_ram[1][`IDX8(0)];
                        8'd109: readdata <= nonce_ram[1][`IDX8(1)];
                        8'd110: readdata <= nonce_ram[1][`IDX8(2)];
                        8'd111: readdata <= nonce_ram[1][`IDX8(3)];


                        8'd112: readdata <= nonce_ram[2][`IDX8(0)];
                        8'd113: readdata <= nonce_ram[2][`IDX8(1)];
                        8'd114: readdata <= nonce_ram[2][`IDX8(2)];
                        8'd115: readdata <= nonce_ram[2][`IDX8(3)];

                        8'd116: readdata <= nonce_ram[3][`IDX8(0)];
                        8'd117: readdata <= nonce_ram[3][`IDX8(1)];
                        8'd118: readdata <= nonce_ram[3][`IDX8(2)];
                        8'd119: readdata <= nonce_ram[3][`IDX8(3)];

                        8'd120: readdata <= nonce_ram[4][`IDX8(0)];
                        8'd121: readdata <= nonce_ram[4][`IDX8(1)];
                        8'd122: readdata <= nonce_ram[4][`IDX8(2)];
                        8'd123: readdata <= nonce_ram[4][`IDX8(3)];

                        8'd124: readdata <= nonce_ram[5][`IDX8(0)];
                        8'd125: readdata <= nonce_ram[5][`IDX8(1)];
                        8'd126: readdata <= nonce_ram[5][`IDX8(2)];
                        8'd127: readdata <= nonce_ram[5][`IDX8(3)];

                        8'd128: readdata <= nonce_ram[6][`IDX8(0)];
                        8'd129: readdata <= nonce_ram[6][`IDX8(1)];
                        8'd130: readdata <= nonce_ram[6][`IDX8(2)];
                        8'd131: readdata <= nonce_ram[6][`IDX8(3)];


                        8'd132: readdata <= nonce_ram[7][`IDX8(0)];
                        8'd133: readdata <= nonce_ram[7][`IDX8(1)];
                        8'd134: readdata <= nonce_ram[7][`IDX8(2)];
                        8'd135: readdata <= nonce_ram[7][`IDX8(3)];



                        //result_ram
                        8'd148: readdata <= result_ram[0][`IDX8(0)];
                        8'd149: readdata <= result_ram[0][`IDX8(1)];
                        8'd150: readdata <= result_ram[0][`IDX8(2)];
                        8'd151: readdata <= result_ram[0][`IDX8(3)];
                        8'd152: begin
                                                readdata[7:1] <=
6'b000000;
                                            readdata[0] <=
result_ram[0][32];
                                                end

                        8'd153: readdata <= result_ram[1][`IDX8(0)];
                        8'd154: readdata <= result_ram[1][`IDX8(1)];
                        8'd155: readdata <= result_ram[1][`IDX8(2)];
                        8'd156: readdata <= result_ram[1][`IDX8(3)];
                        8'd157: begin
                                                readdata[7:1] <=
6'b000000;
```

```
                                        readdata[0] <=
result_ram[1][32];
                                        end

                            8'd158: readdata <= result_ram[2][`IDX8(0)];
                            8'd159: readdata <= result_ram[2][`IDX8(1)];
                            8'd160: readdata <= result_ram[2][`IDX8(2)];
                            8'd161: readdata <= result_ram[2][`IDX8(3)];
                            8'd162: begin
                                            readdata[7:1] <=
6'b000000;
                                        readdata[0] <=
result_ram[2][32];
                                        end

                            8'd163: readdata <= result_ram[3][`IDX8(0)];
                            8'd164: readdata <= result_ram[3][`IDX8(1)];
                            8'd165: readdata <= result_ram[3][`IDX8(2)];
                            8'd166: readdata <= result_ram[3][`IDX8(3)];
                            8'd167: begin
                                            readdata[7:1] <=
6'b000000;
                                        readdata[0] <=
result_ram[3][32];
                                        end

                            8'd168: readdata <= result_ram[4][`IDX8(0)];
                            8'd169: readdata <= result_ram[4][`IDX8(1)];
                            8'd170: readdata <= result_ram[4][`IDX8(2)];
                            8'd171: readdata <= result_ram[4][`IDX8(3)];
                            8'd172: begin
                                            readdata[7:1] <=
6'b000000;
                                        readdata[0] <=
result_ram[4][32];
                                        end

                            8'd173: readdata <= result_ram[5][`IDX8(0)];
                            8'd174: readdata <= result_ram[5][`IDX8(1)];
                            8'd175: readdata <= result_ram[5][`IDX8(2)];
                            8'd176: readdata <= result_ram[5][`IDX8(3)];
                            8'd177: begin
                                            readdata[7:1] <=
6'b000000;
                                        readdata[0] <=
result_ram[5][32];
                                        end

                            8'd178: readdata <= result_ram[6][`IDX8(0)];
                            8'd179: readdata <= result_ram[6][`IDX8(1)];
                            8'd180: readdata <= result_ram[6][`IDX8(2)];
                            8'd181: readdata <= result_ram[6][`IDX8(3)];
                            8'd182: begin
                                            readdata[7:1] <=
6'b000000;
                                        readdata[0] <=
result_ram[6][32];
                                        end

                            8'd183: readdata <= result_ram[7][`IDX8(0)];
```

```
                                8'd184: readdata <= result_ram[7][`IDX8(1)];
                                8'd185: readdata <= result_ram[7][`IDX8(2)];
                                8'd186: readdata <= result_ram[7][`IDX8(3)];
                                8'd187: begin
                                                                readdata[7:1] <=
6'b000000;
                                                        readdata[0] <=
result_ram[7][32];
                                                        end



                        endcase
                end
                else if(stop) begin
                                // Stop state. When a miner has solved the block and
nonce

                                // Continuously loads the nonce output of the miners
to result ram
                                result_ram[0][31:0] <= nonce_out_a0[`IDX32(0)];
                                result_ram[1][31:0] <= nonce_out_a1[`IDX32(0)];
                                result_ram[2][31:0] <= nonce_out_a2[`IDX32(0)];
                                result_ram[3][31:0] <= nonce_out_a3[`IDX32(0)];
                                result_ram[4][31:0] <= nonce_out_a4[`IDX32(0)];
                                result_ram[5][31:0] <= nonce_out_a5[`IDX32(0)];
                                result_ram[6][31:0] <= nonce_out_a6[`IDX32(0)];
                                result_ram[7][31:0] <= nonce_out_a7[`IDX32(0)];


                                result_ram[0][32] <= nonce_out_a0[32];
                                result_ram[1][32] <= nonce_out_a1[32];
                                result_ram[2][32] <= nonce_out_a2[32];
                                result_ram[3][32] <= nonce_out_a3[32];
                                result_ram[4][32] <= nonce_out_a4[32];

                                result_ram[5][32] <= nonce_out_a5[32];
                                result_ram[6][32] <= nonce_out_a6[32];
                                result_ram[7][32] <= nonce_out_a7[32];

                end
                else begin
                        load_done <= 1'b0;

                        // check if any of the miners has solved the nonce by
checking their tickets
                        ticket = nonce_out_a0[32] ||
                                                        nonce_out_a1[32] ||
                                                        nonce_out_a2[32] ||
                                                        nonce_out_a3[32] ||
                                                        nonce_out_a4[32] ||
                                                        nonce_out_a5[32] ||
                                                        nonce_out_a6[32] ||
                                                        nonce_out_a7[32];



                        // Condition to enter Stop state
                        if(read_gold_nonce && ticket && !loading) begin
```

```
                                result_ram[0][31:0] <= nonce_out_a0[`IDX32(0)];
                                result_ram[1][31:0] <= nonce_out_a1[`IDX32(0)];
                                result_ram[2][31:0] <= nonce_out_a2[`IDX32(0)];
                                result_ram[3][31:0] <= nonce_out_a3[`IDX32(0)];
                                result_ram[4][31:0] <= nonce_out_a4[`IDX32(0)];
                                result_ram[5][31:0] <= nonce_out_a5[`IDX32(0)];
                                result_ram[6][31:0] <= nonce_out_a6[`IDX32(0)];
                                result_ram[7][31:0] <= nonce_out_a7[`IDX32(0)];


                                result_ram[0][32] <= nonce_out_a0[32];
                                result_ram[1][32] <= nonce_out_a1[32];

                                result_ram[2][32] <= nonce_out_a2[32];
                                result_ram[3][32] <= nonce_out_a3[32];
                                result_ram[4][32] <= nonce_out_a4[32];
                                result_ram[5][32] <= nonce_out_a5[32];
                                result_ram[6][32] <= nonce_out_a6[32];
                                result_ram[7][32] <= nonce_out_a7[32];


                                read_gold_nonce = 1'b0;
                                stop <= 1'b1;
                        end
                        else begin
                                result_ram[0] <= 33'd0;
                                result_ram[1] <= 33'd0;
                                result_ram[2] <= 33'd0;
                                result_ram[3] <= 33'd0;
                                result_ram[4] <= 33'd0;
                                result_ram[5] <= 33'd0;
                                result_ram[6] <= 33'd0;
                                result_ram[7] <= 33'd0;

                        end

                end
        end //end of always @ (posedge clk)

endmodule
```

fpgaminer_top.v

```
/*
*
* Copyright (c) 2011-2012 fpgaminer@bitcoin-mining.com
*
*
*
* This program is free software: you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation, either version 3 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program.  If not, see <http://www.gnu.org/licenses/>.
*
*/


`timescale 1ns/1ps

//module fpgaminer_top (osc_clk, midstate_buf_in, data_in);
module fpgaminer_top (clk, header_data_input, load_done, nonce_out);

	// The LOOP_LOG2 parameter determines how unrolled the SHA-256
	// calculations are. For example, a setting of 0 will completely
	// unroll the calculations, resulting in 128 rounds and a large, but
	// fast design.
	//
	// A setting of 1 will result in 64 rounds, with half the size and
	// half the speed. 2 will be 32 rounds, with 1/4th the size and speed.
	// And so on.
	//
	// Valid range: [0, 5]
`ifdef CONFIG_LOOP_LOG2
	parameter LOOP_LOG2 = `CONFIG_LOOP_LOG2;
`else
	parameter LOOP_LOG2 = 5;
`endif


	// No need to adjust these parameters
	localparam [5:0] LOOP = (6'd1 << LOOP_LOG2);
	// The nonce will always be larger at the time we discover a valid
	// hash. This is its offset from the nonce that gave rise to the valid
	// hash (except when LOOP_LOG2 == 0 or 1, where the offset is 131 or
	// 66 respectively).
	localparam [31:0] GOLDEN_NONCE_OFFSET = (32'd1 << (7 - LOOP_LOG2)) +
32'd1;

	input clk;
	input [767:0] header_data_input;
	input load_done;
	output [32:0] nonce_out;
	reg [32:0] nonce_out = 33'd0;

	////
	reg [255:0] state = 0;
	reg [511:0] data = 0;
	reg [31:0] nonce = 32'h00000000;


	//// PLL
	wire hash_clk;

  assign hash_clk = clk;

	//// Hashers
	wire [255:0] hash, hash2;
	reg [5:0] cnt = 6'd0;
```

```
	reg feedback = 1'b0;

	sha256_transform #(.LOOP(LOOP)) uut (
		.clk(hash_clk),
		.feedback(feedback),
		.cnt(cnt),
		.rx_state(state),
		.rx_input(data),
		.tx_hash(hash)
	);
	sha256_transform #(.LOOP(LOOP)) uut2 (
		.clk(hash_clk),
		.feedback(feedback),
		.cnt(cnt),

.rx_state(256'h5be0cd191f83d9ab9b05688c510e527fa54ff53a3c6ef372bb67ae856a09e667)
,	//H7,...,H0

.rx_input({256'h0000010000000000000000000000000000000000000000000000000080000000
, hash}),
		.tx_hash(hash2)
	);


	reg [255:0] midstate_buf = 0, data_buf = 0;
	wire [255:0] midstate_vw, data2_vw;
	reg [31:0] golden_nonce = 0;
  reg start = 1'b0;


	//// Control Unit
	reg is_golden_ticket = 1'b0;
	reg feedback_d1 = 1'b1;
	wire [5:0] cnt_next;
	wire [31:0] nonce_next;
	wire feedback_next;
	`ifndef SIM
		wire reset;
		assign reset = 1'b0;
	`else
		reg reset = 1'b0;	// NOTE: Reset is not currently used in the
actual FPGA; for simulation only.
	`endif

	assign cnt_next =  reset ? 6'd0 : (LOOP == 1) ? 6'd0 : (cnt + 6'd1) &
(LOOP-1);
	// On the first count (cnt==0), load data from previous stage (no
feedback)
	// on 1..LOOP-1, take feedback from current stage
	// This reduces the throughput by a factor of (LOOP), but also reduces
the design size by the same amount
	assign feedback_next = (LOOP == 1) ? 1'b0 : (cnt_next !=
{(LOOP_LOG2){1'b0}});
	assign nonce_next =
		reset ? 32'd0 :
		feedback_next ? nonce : (nonce + 32'd1);

	always @ (posedge hash_clk)
	begin
		`ifdef SIM
```

```
                        midstate_buf <=
256'h2b3f81261b3cfd001db436cfdsim:/fpgaminer_top/midstate_buf4c8f3f9c7450c9a0d04
9bee71cba0ea2619c0b5;
                        data_buf <=
256'h00000000000000000000000080000000_00000000_39f3001b6b7b8d4dc14bfc31;
                        nonce <= 30411740;
                `else
                        if(load_done)
                                start <= 1'b1;

                        if(start == 1'b1) begin
                           midstate_buf <= header_data_input[255:0];
                                data_buf <= header_data_input[767:256];
                                is_golden_ticket <= 1'b0;
                                golden_nonce <= 32'd0;
                                nonce_out <= 33'd0;
                        end
                `endif

    if(is_golden_ticket) begin
      nonce_out[32] <= is_golden_ticket;
    end
    nonce_out[31:0] <= golden_nonce;

                cnt <= cnt_next;
                feedback <= feedback_next;
                feedback_d1 <= feedback;


                // Give new data to the hasher
                state <= midstate_buf;
                data <=
{384'h000002800000000000000000000000000000000000000000000000000000000000000000000000
00000000000080000000, nonce_next, data_buf[95:0]};
          if(start == 1'b0)
             nonce <= nonce_next;
          else begin
             nonce <= header_data_input[383:352];
             start <= 1'b0;
          end

                // Check to see if the last hash generated is valid.
                is_golden_ticket <= (hash2[255:224] == 32'h00000000) &&
!feedback_d1;
                if(is_golden_ticket)
                begin
                        // TODO: Find a more compact calculation for this
                        if (LOOP == 1)
                                golden_nonce <= nonce - 32'd131;
                        else if (LOOP == 2)
                                golden_nonce <= nonce - 32'd66;
                        else
                                golden_nonce <= nonce - GOLDEN_NONCE_OFFSET;
                end

        end

endmodule
```

sha256_transform.v

```
/*
*
* Copyright (c) 2011 fpgaminer@bitcoin-mining.com
*
*
*
* This program is free software: you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation, either version 3 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program.  If not, see <http://www.gnu.org/licenses/>.
*
*/


`timescale 1ns/1ps

// A quick define to help index 32-bit words inside a larger register.
`define IDX(x) (((x)+1)*(32)-1):((x)*(32))


// Perform a SHA-256 transformation on the given 512-bit data, and 256-bit
// initial state,
// Outputs one 256-bit hash every LOOP cycle(s).
//
// The LOOP parameter determines both the size and speed of this module.
// A value of 1 implies a fully unrolled SHA-256 calculation spanning 64 round
// modules and calculating a full SHA-256 hash every clock cycle. A value of
// 2 implies a half-unrolled loop, with 32 round modules and calculating
// a full hash in 2 clock cycles. And so forth.
module sha256_transform #(
	parameter LOOP = 6'd4
) (
	input clk,
	input feedback,
	input [5:0] cnt,
	input [255:0] rx_state,
	input [511:0] rx_input,
	output reg [255:0] tx_hash
);

	// Constants defined by the SHA-2 standard.
	localparam Ks = {
		32'h428a2f98, 32'h71374491, 32'hb5c0fbcf, 32'he9b5dba5,
		32'h3956c25b, 32'h59f111f1, 32'h923f82a4, 32'hab1c5ed5,
		32'hd807aa98, 32'h12835b01, 32'h243185be, 32'h550c7dc3,
		32'h72be5d74, 32'h80deb1fe, 32'h9bdc06a7, 32'hc19bf174,
		32'he49b69c1, 32'hefbe4786, 32'h0fc19dc6, 32'h240ca1cc,
		32'h2de92c6f, 32'h4a7484aa, 32'h5cb0a9dc, 32'h76f988da,
```

```
                32'h983e5152, 32'ha831c66d, 32'hb00327c8, 32'hbf597fc7,
                32'hc6e00bf3, 32'hd5a79147, 32'h06ca6351, 32'h14292967,
                32'h27b70a85, 32'h2e1b2138, 32'h4d2c6dfc, 32'h53380d13,
                32'h650a7354, 32'h766a0abb, 32'h81c2c92e, 32'h92722c85,
                32'ha2bfe8a1, 32'ha81a664b, 32'hc24b8b70, 32'hc76c51a3,
                32'hd192e819, 32'hd6990624, 32'hf40e3585, 32'h106aa070,
                32'h19a4c116, 32'h1e376c08, 32'h2748774c, 32'h34b0bcb5,
                32'h391c0cb3, 32'h4ed8aa4a, 32'h5b9cca4f, 32'h682e6ff3,
                32'h748f82ee, 32'h78a5636f, 32'h84c87814, 32'h8cc70208,
                32'h90befffa, 32'ha4506ceb, 32'hbef9a3f7, 32'hc67178f2};


        genvar i;

        generate

                for (i = 0; i < 64/LOOP; i = i + 1) begin : HASHERS
                        wire [511:0] W;
                        wire [255:0] state;

                        if(i == 0)
                                sha256_digester U (
                                        .clk(clk),
                                        .k(Ks[32*(63-cnt) +: 32]),
                                        .rx_w(feedback ? W : rx_input),
                                        .rx_state(feedback ? state : rx_state),
                                        .tx_w(W),
                                        .tx_state(state)
                                );
                        else
                                sha256_digester U (
                                        .clk(clk),
                                        .k(Ks[32*(63-LOOP*i-cnt) +: 32]),
                                        .rx_w(feedback ? W : HASHERS[i-1].W),
                                        .rx_state(feedback ? state :
HASHERS[i-1].state),
                                        .tx_w(W),
                                        .tx_state(state)
                                );
                end

        endgenerate

        always @ (posedge clk)
        begin
                if (!feedback)
                begin
                        tx_hash[`IDX(0)] <= rx_state[`IDX(0)] +
HASHERS[64/LOOP-6'd1].state[`IDX(0)];
                        tx_hash[`IDX(1)] <= rx_state[`IDX(1)] +
HASHERS[64/LOOP-6'd1].state[`IDX(1)];
                        tx_hash[`IDX(2)] <= rx_state[`IDX(2)] +
HASHERS[64/LOOP-6'd1].state[`IDX(2)];
                        tx_hash[`IDX(3)] <= rx_state[`IDX(3)] +
HASHERS[64/LOOP-6'd1].state[`IDX(3)];
                        tx_hash[`IDX(4)] <= rx_state[`IDX(4)] +
HASHERS[64/LOOP-6'd1].state[`IDX(4)];
                        tx_hash[`IDX(5)] <= rx_state[`IDX(5)] +
HASHERS[64/LOOP-6'd1].state[`IDX(5)];
```

```
                        tx_hash[`IDX(6)] <= rx_state[`IDX(6)] +
HASHERS[64/LOOP-6'd1].state[`IDX(6)];
                        tx_hash[`IDX(7)] <= rx_state[`IDX(7)] +
HASHERS[64/LOOP-6'd1].state[`IDX(7)];
                end
        end


endmodule


module sha256_digester (clk, k, rx_w, rx_state, tx_w, tx_state);

        input clk;
        input [31:0] k;
        input [511:0] rx_w;
        input [255:0] rx_state;

        output reg [511:0] tx_w;
        output reg [255:0] tx_state;


        wire [31:0] e0_w, e1_w, ch_w, maj_w, s0_w, s1_w;


        e0      e0_blk  (rx_state[`IDX(0)], e0_w);
        e1      e1_blk  (rx_state[`IDX(4)], e1_w);
        ch      ch_blk  (rx_state[`IDX(4)], rx_state[`IDX(5)], rx_state[`IDX(6)],
ch_w);
        maj     maj_blk         (rx_state[`IDX(0)], rx_state[`IDX(1)],
rx_state[`IDX(2)], maj_w);
        s0      s0_blk  (rx_w[63:32], s0_w);
        s1      s1_blk  (rx_w[479:448], s1_w);

        wire [31:0] t1 = rx_state[`IDX(7)] + e1_w + ch_w + rx_w[31:0] + k;
        wire [31:0] t2 = e0_w + maj_w;
        wire [31:0] new_w = s1_w + rx_w[319:288] + s0_w + rx_w[31:0];


        always @ (posedge clk)
        begin
                tx_w[511:480] <= new_w;
                tx_w[479:0] <= rx_w[511:32];

                tx_state[`IDX(7)] <= rx_state[`IDX(6)];
                tx_state[`IDX(6)] <= rx_state[`IDX(5)];
                tx_state[`IDX(5)] <= rx_state[`IDX(4)];
                tx_state[`IDX(4)] <= rx_state[`IDX(3)] + t1;
                tx_state[`IDX(3)] <= rx_state[`IDX(2)];
                tx_state[`IDX(2)] <= rx_state[`IDX(1)];
                tx_state[`IDX(1)] <= rx_state[`IDX(0)];
                tx_state[`IDX(0)] <= t1 + t2;
        end

endmodule
```

sha-256-functions.v

```
/*
*
* Copyright (c) 2011 fpgaminer@bitcoin-mining.com
*
*
*
* This program is free software: you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation, either version 3 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program.  If not, see <http://www.gnu.org/licenses/>.
*
*/


`timescale 1ns/1ps

module e0 (x, y);

	input [31:0] x;
	output [31:0] y;

	assign y = {x[1:0],x[31:2]} ^ {x[12:0],x[31:13]} ^ {x[21:0],x[31:22]};

endmodule


module e1 (x, y);

	input [31:0] x;
	output [31:0] y;

	assign y = {x[5:0],x[31:6]} ^ {x[10:0],x[31:11]} ^ {x[24:0],x[31:25]};

endmodule


module ch (x, y, z, o);

	input [31:0] x, y, z;
	output [31:0] o;

	assign o = z ^ (x & (y ^ z));

endmodule


module maj (x, y, z, o);

	input [31:0] x, y, z;
	output [31:0] o;
```

```
        assign o = (x & y) | (z & (x | y));

endmodule


module s0 (x, y);

        input [31:0] x;
        output [31:0] y;

        assign y[31:29] = x[6:4] ^ x[17:15];
        assign y[28:0] = {x[3:0], x[31:7]} ^ {x[14:0],x[31:18]} ^ x[31:3];

endmodule


module s1 (x, y);

        input [31:0] x;
        output [31:0] y;

        assign y[31:22] = x[16:7] ^ x[18:9];
        assign y[21:0] = {x[6:0],x[31:17]} ^ {x[8:0],x[31:19]} ^ x[31:10];

endmodule
```

getwork.c

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <sys/ioctl.h>
#include <curl/curl.h>
#include <jansson.h>
#include <pthread.h>
#include <fcntl.h>
#include "vga_led.h"
#include <unistd.h>
#include <sys/time.h>

#define BUFFER_SIZE  (256 * 1024)  /* 256 KB */

#define HEADER_BUFFER_SIZE 96
#define MIDSTATE_SIZE 64
#define DATA_SIZE 32

#define CHARS_TO_DATA 188
#define DATA_LENGTH 256

#define IP_OFFSET 7
#define IP_LENGTH 14

#define SIGNAL_OFFSET 7
#define NONCE_CHECK_START 152
#define NONCE_CHECK_FINISH 188
#define DATA_START 147
```

```
#define DATA_END 167
#define INIT_WRITE 102

// Example json response from network
// curl --user halffast.worker1:WyhZfpFS --data-binary '{ "id":"curltest",
"method":"getwork", "params":[] }' -H 'content-type: text/plain;'
http://localhost:8332/lp -i
//{"error": null,
//  "id": "curltest",
//  "result":
//  {"hash1":
"00000000000000000000000000000000000000000000000000000000000000000000000000000080000000
0000000000000000000000000000000000000000010000",
//     "data":
"00000029b39574cb8a5c25e94cdd844f7d3b942384af65fb264d5a0000000000000008adb353
c754fe07c40341728c5efe6e3782d994e0e3ebcb4879106de94dd1db0533b2c461900db990000000
0000008000000000000000000000000000000000000000000000000000000000000000000000000
00000000080020000",
//     "target":
"0000000000000000000000000000000000000000000000000000ffff00000000",
//     "midstate":
"49bc9ada38ba6b43814a3275edfa09b796be70080d5c641d32b80776aef13f78"}
//}

//defaults
const char *req = "{ \"id\":\"1\", \"method\":\"getwork\", \"params\":[] }";
const char *usrpwd = "halffast.worker1:WyhZfpFS";
const char *header = "content-type: text/plain;";
const char *pool_url = "http://207.10.143.139:8332/";
const char *lp_pool_url = "http://207.10.143:8332/lp";

struct timeval stop, start;

//initial data values
char current_data[DATA_LENGTH] =
"deadbee14cc2c57c7905fd399965282c87fe259e7da366e035dc087a0000141f000000006427b64
92f2b052578fb4bc23655ca4e8b9e2b9b69c88041b2ac8c771571d1be4de695931a2694217a00110
e000000800000000000000000000000000000000000000000000000000000000000000000000000
00000000080020000\0";

int vga_led_fd;
unsigned char nonce[4];

void die(const char *message)
{
        perror(message);
        exit(1);
}

void json_die(const int error_line, const char *error_text)
{
        fprintf(stderr, "error: on line %d: %s\n", error_line, error_text);
        exit(1);
}

void
json_data_error(json_int_t json_data_error_num){
        if(json_data_error_num != 0){
        switch(json_data_error_num){
                case 20 : printf("Other/Unknown\n");
```

```
                case 21 : printf("Job not found (=stale)\n");
                case 22 : printf("Duplicate share\n");
                case 23 : printf("Low difficulty share\n");
                case 24 : printf("Unauthorized worker\n");
                case 25 : printf("Not subscribed\n");
                default : printf("strange error google resuslts\n");
        }
        die("error in result from json response");
        }
}

/* Read and print the segment values */
void print_segment_info() {
  vga_led_arg_t vla;
  int i;

  for (i = VGA_LED_DIGITS+SIGNAL_OFFSET; i > -1; i--){
        vla.digit = i;
        if (ioctl(vga_led_fd, VGA_LED_READ_DIGIT, &vla)) {
        perror("ioctl(VGA_LED_READ_DIGIT) failed");
        return;
        }
        printf("%02x ", vla.segments);
  }
  printf("\n");
}

//Returns 1 if the if there is a solved nonce value
int checkTicket() {
  int i;
  vga_led_arg_t vla;
  for(i=NONCE_CHECK_START; i < NONCE_CHECK_FINISH; i+=5) {
        vla.digit = i;
        if (ioctl(vga_led_fd, VGA_LED_READ_DIGIT, &vla)) {
        perror("ioctl(VGA_LED_READ_DIGIT) failed");
        return 0;
        }
        if(vla.segments == 1) {
        int j, idx;
        idx = 3;
        for(j=i-1; j > i-5; j--) {
        vla.digit = j;
        if (ioctl(vga_led_fd, VGA_LED_READ_DIGIT, &vla)) {
                perror("ioctl(VGA_LED_READ_DIGIT) failed");
                return 0;
        }
        *(nonce+idx) = (unsigned char) vla.segments;
        idx--;
        }
        return 1;
        }

  }
  return 0;
}

//print what was written to the kernel module
void print_result() {
  vga_led_arg_t vla;
  int i;
```

```
  int j=0;
  for (i = DATA_END; i > DATA_START; i--){
        vla.digit = i;
        if (ioctl(vga_led_fd, VGA_LED_READ_DIGIT, &vla)) {
        perror("ioctl(VGA_LED_READ_DIGIT) failed");
        return;
        }
        printf("%02x ", vla.segments);
        printf("%d ", i);
        if( (j++) % 5 == 0)
        printf("\n");
  }
  printf("\n");
}

void write_segments(unsigned char *segs)
{
  vga_led_arg_t vla;
  int i;
  for (i = 0 ; i < VGA_LED_DIGITS; i++){
        vla.digit = i;
        vla.segments = segs[i];
        if (ioctl(vga_led_fd, VGA_LED_WRITE_DIGIT, &vla)) {
        perror("ioctl(VGA_LED_WRITE_DIGIT) failed");
        return;
        }
  }
}

void init_write(){
        vga_led_arg_t vla;
        vla.digit = INIT_WRITE;
        vla.segments = 1;
        if (ioctl(vga_led_fd, VGA_LED_WRITE_DIGIT, &vla)) {
        perror("ioctl(VGA_LED_WRITE_DIGIT) failed");
        return;
        }
}

// write_result, write_response and request came from the jansson tutorial
// http://jansson.readthedocs.org/en/2.3/tutorial.html
struct write_result
{
        char *data;
        int pos;
};

size_t write_response(void *ptr, size_t size, size_t nmemb, void *stream)
{
        struct write_result *result = (struct write_result *)stream;

        if(result->pos + size * nmemb >= BUFFER_SIZE - 1)
        {
        fprintf(stderr, "error: too small buffer\n");
        return 0;
        }

        memcpy(result->data + result->pos, ptr, size * nmemb);
        result->pos += size * nmemb;
```

```
        return size * nmemb;
}

//send http request with easy_curl
char *request(const char *url, const char *bin_data)
{
        CURL *curl = NULL;
        CURLcode status;
        struct curl_slist *headers = NULL;
        char *data = NULL;
        long code;

        curl_global_init(CURL_GLOBAL_ALL);
        curl = curl_easy_init();
        if(!curl)
        goto error;

        data = malloc(BUFFER_SIZE);
        if(!data)
        goto error;

        struct write_result write_result = {
        .data = data,
        .pos = 0
        };

        //set up url
        curl_easy_setopt(curl, CURLOPT_URL, url);

        //set up bin_data the crux of the getwork request
        curl_easy_setopt(curl, CURLOPT_POSTFIELDS, ((void *) bin_data));
        curl_easy_setopt(curl, CURLOPT_POSTFIELDSIZE, ((long)-1) );

        //set up usrpwd
        curl_easy_setopt(curl, CURLOPT_USERPWD, usrpwd);

        //set up headers from init response
        headers = curl_slist_append(headers, header);
        curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);

        //set up writing repsonse
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_response);
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, &write_result);

        //preform the curl
        status = curl_easy_perform(curl);

        //error handle
        if(status != 0)
        {
        fprintf(stderr, "error: unable to request data from %s:\n", url);
        fprintf(stderr, "%s\n", curl_easy_strerror(status));
        goto error;
        }

        curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &code);
        if(code != 200)
        {
        fprintf(stderr, "error: server responded with code %ld\n", code);
        goto error;
```

```
        }

        curl_easy_cleanup(curl);
        curl_slist_free_all(headers);
        curl_global_cleanup();

        /* zero-terminate the result */
        data[write_result.pos] = '\0';

        return data;

error:
        if(data)
        free(data);
        if(curl)
        curl_easy_cleanup(curl);
        if(headers)
        curl_slist_free_all(headers);
        curl_global_cleanup();
        return NULL;
}

unsigned char nibbleFromChar(char c)
{
        if(c >= '0' && c <= '9') return c - '0';
        if(c >= 'a' && c <= 'f') return c - 'a' + 10;
        if(c >= 'A' && c <= 'F') return c - 'A' + 10;
        return 255;
}

/* Convert a string of characters representing a hex buffer into a series of
bytes of that real value */
unsigned char *hexStringToBytes(unsigned char *inhex)
{
        unsigned char *retval;
        unsigned char *p;
        int len, i;

        len = strlen(inhex) / 2;
        retval = malloc(len+1);
        for(i=0, p = (unsigned char *) inhex; i<len; i++) {
        retval[i] = (nibbleFromChar(*p) << 4) | nibbleFromChar(*(p+1));
        p += 2;
        }
        retval[len] = 0;
        return retval;
}


void *proof_of_work(void *arg){
        char *proof_resp;
        char nonce_hex[9];
        json_t *json_setup, *json_resp;
        json_error_t json_error;
        json_t *result = NULL;
        while(1){
        //if(1){ testing
        if(checkTicket() == 1){
        gettimeofday(&stop, NULL);
        printf("%lu\n", (stop.tv_usec - start.tv_usec));
```

```
        printf("Miner solved a block\n");
        printf("Sending proof of work to Mining pool\n");

                //updating the data with solved nonce
                sprintf(nonce_hex, "%02x%02x%02x%02x",
nonce[0],nonce[1],nonce[2],nonce[3]);
                printf("%s\n",nonce_hex);
                strncpy((char *)arg+152, nonce_hex, 8);

                printf("current data in proof %s", (char *)arg );

        json_setup = json_pack("{ss,s[s]si}","method", "getwork", "params", (char
*)arg, "id",1);

                if(!json_setup){
                json_die(json_error.line, json_error.text);
                }

                proof_resp = request(pool_url, json_string_value(json_setup));
                if(!proof_resp){
                die("proof_of_work repsonse failed check args");
                }

                printf("resp %s\n", proof_resp);
                json_resp = json_loads(proof_resp, 0, &json_error);

                if(!json_resp){
                json_die(json_error.line, json_error.text);
                }
                result = json_object_get(json_resp, "result");

                json_decref(json_resp);
                json_decref(json_setup);
                printf("result %s\n", json_string_value(result));
                // break; testing
    }
        }
}

void
request_and_write_work(char * url){
        char *init_resp;
        json_t *json_resp;
        json_error_t json_error;
        json_t *data_error = NULL, *result = NULL, *data = NULL, *target = NULL,
*midstate = NULL;

        init_resp = request(url, req);
        if(!init_resp){
        die("initial repsonse failed check args");
        }

        json_resp = json_loads(init_resp, 0, &json_error);

        if(!json_resp){
        json_die(json_error.line, json_error.text);
        }

        data_error = json_object_get(json_resp, "error");
```

```
        json_data_error(json_integer_value(data_error));

        result = json_object_get(json_resp, "result");
        data = json_object_get(result, "data");
        target = json_object_get(result, "target");
        midstate = json_object_get(result, "midstate");

        if(!result || !data || !target || !midstate){
        fprintf(stderr, "%s\n", init_resp);
        die("initial repsonse format error");
        }

        memcpy(current_data, init_resp+CHARS_TO_DATA, DATA_LENGTH);

        unsigned char *data_bytes = hexStringToBytes(json_string_value(data));
        unsigned char *midstate_bytes =
hexStringToBytes(json_string_value(midstate));
        unsigned char *data_bytes = hexStringToBytes(data_test);
        unsigned char *midstate_bytes = hexStringToBytes(midstate_test);

        unsigned char header_buffer[HEADER_BUFFER_SIZE];
        memcpy(header_buffer, midstate_bytes, MIDSTATE_SIZE);
        memcpy(header_buffer+MIDSTATE_SIZE, data_bytes+DATA_SIZE, DATA_SIZE);

        init_write();

        sleep(1);
        gettimeofday(&start, NULL);
        write_segments(header_buffer);
        printf("Json response from bitcoin network:\n %s\n", init_resp); //for
testing
        printf("Writing Data to hardware:\n");
        print_segment_info();

        json_decref(json_resp);
        free(init_resp);
}

int
main(int argc, char**argv){
        static const char filename[] = "/dev/vga_led";
        if((vga_led_fd = open(filename, O_RDWR)) == -1){
        fprintf(stderr, "could not open %s\n", filename);
        return -1;
        }

        pthread_t ack_thread;
        pthread_create(&ack_thread, NULL, proof_of_work, current_data);

        init_write();

        sleep(1);
        print_segment_info();
        request_and_write_work(pool_url);

        //long polling hangs until there is new work to operate on
        while(1){
        request_and_write_work(lp_pool_url);
        }
```

```
        return 0;

}
```

vga_led.c

```
/*
 * Device driver for the VGA LED Emulator
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod vga_led.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree vga_led.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "vga_led.h"

#define DRIVER_NAME "vga_led"

/*
 * Information about our device
 */
struct vga_led_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory */
    u8 segments[VGA_LED_DIGITS];
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_digit(int digit, u8 segments)
{
```

```
    iowrite8(segments, dev.virtbase + digit);
    dev.segments[digit] = segments;
}

static unsigned int read_digit(int digit){
    return ioread8(dev.virtbase + digit);
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long vga_led_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    vga_led_arg_t vla;

    switch (cmd) {
    case VGA_LED_WRITE_DIGIT:
        if (copy_from_user(&vla, (vga_led_arg_t *) arg,
                           sizeof(vga_led_arg_t)))
                return -EACCES;
        if (vla.digit > 103)
                return -EINVAL;
        write_digit(vla.digit, vla.segments);
        break;

    case VGA_LED_READ_DIGIT:
        if (copy_from_user(&vla, (vga_led_arg_t *) arg,
                           sizeof(vga_led_arg_t)))
                return -EACCES;
        printk(KERN_INFO "%02x\n", vla.digit);
        if (vla.digit > 187)
                return -EINVAL;
               vla.segments = read_digit(vla.digit);
        if (copy_to_user((vga_led_arg_t *) arg, &vla,
                         sizeof(vga_led_arg_t)))
                return -EACCES;
        break;

    default:
        return -EINVAL;
    }

    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations vga_led_fops = {
    .owner      = THIS_MODULE,
    .unlocked_ioctl = vga_led_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice vga_led_misc_device = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = DRIVER_NAME,
    .fops       = &vga_led_fops,
};
```

```
/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init vga_led_probe(struct platform_device *pdev)
{
    int ret;

    /* Register ourselves as a misc device: creates /dev/vga_led */
    ret = misc_register(&vga_led_misc_device);

    /* Get the address of our registers from the device tree */
    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret) {
        ret = -ENOENT;
        goto out_deregister;
    }

    /* Make sure we can use these registers */
    if (request_mem_region(dev.res.start, resource_size(&dev.res),
                           DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }

    /* Arrange access to our registers */
    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_mem_region;
    }

    return 0;

out_release_mem_region:
    release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
    misc_deregister(&vga_led_misc_device);
    return ret;
}

/* Clean-up code: release resources */
static int vga_led_remove(struct platform_device *pdev)
{
    iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    misc_deregister(&vga_led_misc_device);
    return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id vga_led_of_match[] = {
    { .compatible = "altr,vga_led" },
    {},
};
MODULE_DEVICE_TABLE(of, vga_led_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
```

```
static struct platform_driver vga_led_driver = {
    .driver    = {
        .name    = DRIVER_NAME,
        .owner    = THIS_MODULE,
        .of_match_table = of_match_ptr(vga_led_of_match),
    },
    .remove    = __exit_p(vga_led_remove),
};

/* Called when the module is loaded: set things up */
static int __init vga_led_init(void)
{
    pr_info(DRIVER_NAME ": init\n");
    return platform_driver_probe(&vga_led_driver, vga_led_probe);
}

/* Called when the module is unloaded: release resources */
static void __exit vga_led_exit(void)
{
    platform_driver_unregister(&vga_led_driver);
    pr_info(DRIVER_NAME ": exit\n");
}

module_init(vga_led_init);
module_exit(vga_led_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("VGA 7-segment LED Emulator");
```

vga_led.h

```
#ifndef _VGA_LED_H
#define _VGA_LED_H

#include <linux/ioctl.h>

#define VGA_LED_DIGITS 96

typedef struct {
  unsigned char digit;       /* 0, 1, .. , VGA_LED_DIGITS - 1 */
  unsigned char segments; /* LSB is segment a, MSB is decimal point */
} vga_led_arg_t;

#define VGA_LED_MAGIC 'q'

/* ioctls and their arguments */
#define VGA_LED_WRITE_DIGIT _IOW(VGA_LED_MAGIC, 1, vga_led_arg_t *)
#define VGA_LED_READ_DIGIT  _IOWR(VGA_LED_MAGIC, 2, vga_led_arg_t *)

#endif
```

syscon-bitcoin-test2.tcl

```
# Base addresses of the peripherals: take from Qsys
set bitcoin_miner 0x0

puts "Started system-console-test-script"
```

```
# Using the JTAG chain, check the clock and reset"

set j [lindex [get_service_paths jtag_debug] 0]
open_service jtag_debug $j
puts "Opened jtag_debug"

#issues reset request to the system thru JTAG to Avalon Master Bridge
puts "Checking the JTAG chain loopback: [jtag_debug_loop $j {1 2 3 4 5 6}]"
jtag_debug_reset_system $j

puts -nonewline "Sampling the clock: "
foreach i {1 1 1 1 1 1 1 1 1 1 1} {
       puts -nonewline [jtag_debug_sample_clock $j]
}
puts ""

puts "Checking reset state: [jtag_debug_sample_reset $j]"

close_service jtag_debug $j
puts "Closed jtag_debug"

# Perform bus reads and writes
set m [lindex [get_service_paths master] 0]
open_service master $m
puts "Opened master"

set state [master_read_8 $m [expr $bitcoin_miner + 104] 1]
puts "state0: $state"

#send 'start' signal to miner
puts "Sending start signal to miner...\n"
master_write_8 $m [expr $bitcoin_miner + 102] 0x01
set start [master_read_8 $m [expr $bitcoin_miner + 102] 1]
puts "start: $start"

puts "Writing incorrect header data"
foreach {r v} {0 0x90 1 0xf7 2 0x41 3 0xaf 4 0xb3 5 0xab 6 0x06 7 0xf1 8 0xa5 9
0x82 10 0xc5 11 0xc8 12 0x5e 13 0xe7 14 0xa5 15 0x61 16 0x91 17 0x2b 18 0x25 19
0xa7 20 0xcd 21 0x09 22 0xc0 23 0x60 24 0xa8 25 0x9b 26 0x3c 27 0x2a 28 0x73 29
0xa4 30 0xff 31 0xff 32 0xff 33 0xff 34 0xff 35 0xbe 36 0x4d 37 0xe6 38 0x95 39
0x93 40 0x1a 41 0x26 42 0x94 43 0x21 44 0x7a 45 0x22 46 0x22 47 0x0e 48 0x00 49
0x00 50 0x00 51 0x80 52 0x00 53 0x00 54 0x00 55 0x00 56 0x00 57 0x00 58 0x00 59
0x00 60 0x00 61 0x00 62 0x00 63 0x00 64 0x00 65 0x00 66 0x00 67 0x00 68 0x00 69
0x00 70 0x00 71 0x00 72 0x00 73 0x00 74 0x00 75 0x00 76 0x00 77 0x00 78 0x00 79
0x00 80 0x00 81 0x00 82 0x00 83 0x00 84 0x00 85 0x00 86 0x00 87 0x00 88 0x00 89
0x00 90 0x00 91 0x00 92 0x80 93 0x02 94 0x00 95 0x00} {
       master_write_8 $m [expr $bitcoin_miner + $r] $v
}
```

syscon-bitcoin-test3.tcl

```
# Base addresses of the peripherals: take from Qsys
set bitcoin_miner 0x0

puts "Started system-console-test-script"
```

```
# Using the JTAG chain, check the clock and reset"

set j [lindex [get_service_paths jtag_debug] 0]
open_service jtag_debug $j
puts "Opened jtag_debug"

#issues reset request to the system thru JTAG to Avalon Master Bridge
puts "Checking the JTAG chain loopback: [jtag_debug_loop $j {1 2 3 4 5 6}]"
jtag_debug_reset_system $j

puts -nonewline "Sampling the clock: "
foreach i {1 1 1 1 1 1 1 1 1 1 1} {
       puts -nonewline [jtag_debug_sample_clock $j]
}
puts ""

puts "Checking reset state: [jtag_debug_sample_reset $j]"

close_service jtag_debug $j
puts "Closed jtag_debug"

# Perform bus reads and writes
set m [lindex [get_service_paths master] 0]
open_service master $m
puts "Opened master"

set state [master_read_8 $m [expr $bitcoin_miner + 104] 1]
puts "state0: $state"

#send 'start' signal to miner
puts "Sending start signal to miner...\n"
master_write_8 $m [expr $bitcoin_miner + 102] 0x01
set start [master_read_8 $m [expr $bitcoin_miner + 102] 1]
puts "start: $start"

#nonce range is from r=44..47, reverse endian
puts "Writing correct header data"
foreach {r v} {0 0x90 1 0xf7 2 0x41 3 0xaf 4 0xb3 5 0xab 6 0x06 7 0xf1 8 0xa5 9
0x82 10 0xc5 11 0xc8 12 0x5e 13 0xe7 14 0xa5 15 0x61 16 0x91 17 0x2b 18 0x25 19
0xa7 20 0xcd 21 0x09 22 0xc0 23 0x60 24 0xa8 25 0x9b 26 0x3c 27 0x2a 28 0x73 29
0xa4 30 0x8e 31 0x22 32 0x15 33 0x71 34 0xd1 35 0xbe 36 0x4d 37 0xe6 38 0x95 39
0x93 40 0x1a 41 0x26 42 0x94 43 0x21 44 0x7a 45 0x00 46 0x11 47 0x0e 48 0x00 49
0x00 50 0x00 51 0x80 52 0x00 53 0x00 54 0x00 55 0x00 56 0x00 57 0x00 58 0x00 59
0x00 60 0x00 61 0x00 62 0x00 63 0x00 64 0x00 65 0x00 66 0x00 67 0x00 68 0x00 69
0x00 70 0x00 71 0x00 72 0x00 73 0x00 74 0x00 75 0x00 76 0x00 77 0x00 78 0x00 79
0x00 80 0x00 81 0x00 82 0x00 83 0x00 84 0x00 85 0x00 86 0x00 87 0x00 88 0x00 89
0x00 90 0x00 91 0x00 92 0x80 93 0x02 94 0x00 95 0x00} {
       master_write_8 $m [expr $bitcoin_miner + $r] $v
}
```

syscon-bitcoin-readpll.tcl

```
proc hex2bin {hex} {
    set h [string range $hex 2 end]
  binary scan [binary format H* $h] B* bin
  return $bin
}

set bitcoin_miner 0x0
```

```
set m [lindex [get_service_paths master] 0]
open_service master $m
puts "Opened master"

set hb ""
for {set i 95} {$i >= 0} {incr i -1} {
    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
    append hb " $tmp"
}

set nonce ""
for {set i 99} {$i > 95} {incr i -1} {
    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
    append nonce " $tmp"
}


#set n_out ""
#for {set i 107} {$i > 103} {incr i -1} {
#    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
#    append n_out " $tmp"
#}

set start [master_read_8 $m [expr $bitcoin_miner + 102] 1]
set loading [master_read_8 $m [expr $bitcoin_miner + 101] 1]
set ticket [master_read_8 $m [expr $bitcoin_miner + 100] 1]
set nstate [master_read_8 $m [expr $bitcoin_miner + 103] 1]

puts "Output:"
puts "header_buffer:\n$hb"
puts "<stop><start>: [hex2bin $start]"
puts "<loading><loaddone>: [hex2bin $loading]"
puts "nonce: $nonce"
puts "<ticket><gold_nonce32>: [hex2bin $ticket]"
puts "nstate <MINERS1nout32><MINERS0nout32><nonce_out32><read_gold_nonce>:
[hex2bin $nstate]"


puts "\n"

# Read nonces from the nonce_ram
set nonceram0 ""
for {set i 107} {$i > 103} {incr i -1} {
    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
    append nonceram0 " $tmp"
}

set nonceram1 ""
for {set i 111} {$i > 107} {incr i -1} {
    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
    append nonceram1 " $tmp"
}

set nonceram2 ""
for {set i 115} {$i > 111} {incr i -1} {
    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
    append nonceram2 " $tmp"
}
```

```
set nonceram3 ""
for {set i 119} {$i > 115} {incr i -1} {
    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
    append nonceram3 " $tmp"
}

set nonceram4 ""
for {set i 123} {$i > 119} {incr i -1} {
    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
    append nonceram4 " $tmp"
}

set nonceram5 ""
for {set i 127} {$i > 123} {incr i -1} {
    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
    append nonceram5 " $tmp"
}

set nonceram6 ""
for {set i 131} {$i > 127} {incr i -1} {
    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
    append nonceram6 " $tmp"
}

set nonceram7 ""
for {set i 135} {$i > 131} {incr i -1} {
    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
    append nonceram7 " $tmp"
}

# Read the results from result_ram
set resultram0 ""
for {set i 152} {$i > 147} {incr i -1} {
    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
    append resultram0 " $tmp"
}

set resultram1 ""
for {set i 157} {$i > 152} {incr i -1} {
    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
    append resultram1 " $tmp"
}

set resultram2 ""
for {set i 162} {$i > 157} {incr i -1} {
    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
    append resultram2 " $tmp"
}

set resultram3 ""
for {set i 167} {$i > 162} {incr i -1} {
    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
    append resultram3 " $tmp"
}

set resultram4 ""
for {set i 172} {$i > 167} {incr i -1} {
    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
    append resultram4 " $tmp"
}
```

```
set resultram5 ""
for {set i 177} {$i > 172} {incr i -1} {
    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
    append resultram5 " $tmp"
}

set resultram6 ""
for {set i 182} {$i > 177} {incr i -1} {
    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
    append resultram6 " $tmp"
}

set resultram7 ""
for {set i 187} {$i > 182} {incr i -1} {
    set tmp [master_read_8 $m [expr $bitcoin_miner + $i] 1]
    append resultram7 " $tmp"
}

puts "Nonce Ram"
puts "0: $nonceram0"
puts "1: $nonceram1"
puts "2: $nonceram2"
puts "3: $nonceram3"
puts "4: $nonceram4"
puts "5: $nonceram5"
puts "6: $nonceram6"
puts "7: $nonceram7"

puts "Results Ram: <ticket><golden nonce>"
puts "0: $resultram0"
puts "1: $resultram1"
puts "2: $resultram2"
puts "3: $resultram3"
puts "4: $resultram4"
puts "5: $resultram5"
puts "6: $resultram6"
puts "7: $resultram7"

close_service master $m
puts "Closed master"
```