4DM4 Assignment #1
RISC Scheduling of the Chacha20 Stream Cipher

(version - Thurs, Oct. 20, 2022, 4:30 pm)
Due date: Sunday, Oct. 23, 11:59 pm)

Motivation:

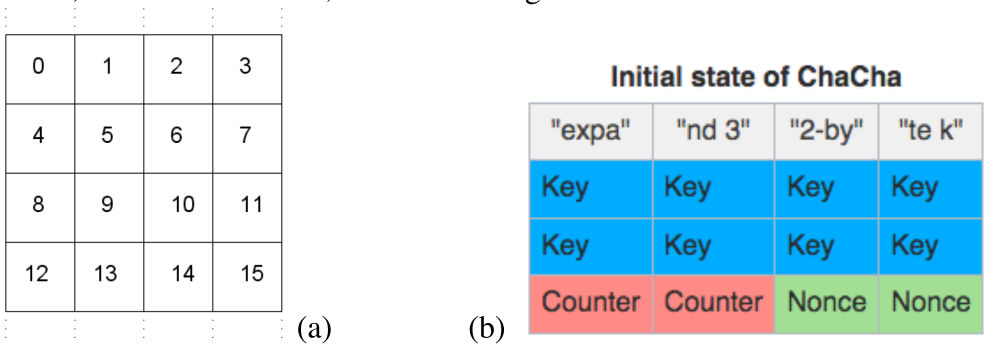
According to the US "National Academy of Engineering" (NAE), "cyber security" has emerged as an outstanding challenge in the 21st century. Cyber security is driving the tremendous investments into classical super-computing systems, and Quantum Computing systems. You will likely encounter cyber security issues at some point in your future careers. 4DM4 assignment #1 will introduce you to some basic and advanced concepts in cyber security. This assignment is new in 2022, and prof. is developing this assignment as the class progresses, so please be patient. (This assignment has undergone a few revisions, as I gathered feedback from the class.) Hopefully, it will be informative and fun.

A first goal of assignment #1 is for everyone to learn about RISC computer architecture, ie basic RISC pipelining and static scheduling of RISC instructions to improve performance. A second goal is for everyone to get an introduction to the Chacha20 stream cipher, which is used extensively to encrypt traffic over the "Internet of Things" (IoT).

1. The Chacha20 Stream Cipher

Chacha20 is a stream cipher, developed by Prof. Daniel Berstein of the University of Chicago, in 2008. The original version (2008) will encrypt a stream of data, with up to 512 Gigabytes of data. The plaintext is partitioned into multiple "blocks", with 64 bytes each, and each block is encrypted, one by one. It uses a 256-bit secret "symmetric" key, ie the sender and receiver each use the same secret key to encrypt and decrypt blocks of data. It uses a 64-bit block-counter, and it uses a 64-bit "nonce", ie a one-time random number. The cipher is designed to work well with 32-bit RISC computers.

Fig. 1 illustrates a block with 64 bytes, arranged as a 4x4 matrix of 32-bit words. Fig. 1b illustrates a "key-stream" block. Each key-stream block has a 128-bit constant in the top row of four words. The 128-bit constant is the phrase "expand 32-byte k", expressed with 16 ASCII characters (with 1 byte per character). Each "key-stream" block also contains the 256-bit key, a 64-bit block counter, and a 64-bit nonce, as shown in Fig. 1b.



0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

(a)

(b)

Initial state of ChaCha

"expa"	"nd 3"	"2-by"	"te k"
Key	Key	Key	Key
Key	Key	Key	Key
Counter	Counter	Nonce	Nonce

Fig. 1. (a) The Chacha block "state", arranged as a 4x4 matrix of 32-bit words.
(b) Chacha20 initial state, for each "key-stream" block.

Chacha20 uses 2 operations: (i) The "QUARTER-ROUND" operation which operates on one individual column or one diagonal of the 4x4 state matrix; (ii) A "DOUBLE-ROUND" operation, which consists of 8 "Quarter-Round" operations.

To encrypt a stream of data with B blocks of data, Chacha20 will first create a "key-stream", which consists of B "key-stream" blocks, each with the initial state shown in Fig. 1. The block-counter labels these key-stream blocks from 1 to B. The Chacha20 cipher then 'processes' each block of key-stream, by running 10 iterations of the DOUBLE-ROUND operation. Chacha20 will then ADD each word of the key-stream blocks (from 1 to B) with the corresponding word from the plaintext blocks (from 1 to B), to create the ciphertext blocks (from 1 to B). (The ADD ignores the carry out when adding 32-bit words.)

```
Lines 1, 2, 3:     a  += b;   d  ^=  a;   d  <<<= 16;
Lines 4, 5, 6:     c  += d;   b  ^=  c;   b  <<<= 12;
Lines 7, 8, 9:     a  += b;   d  ^=  a;   d  <<<= 8;
Lines 10-12:       c  += d;   b  ^=  c;   b  <<<= 7;
```

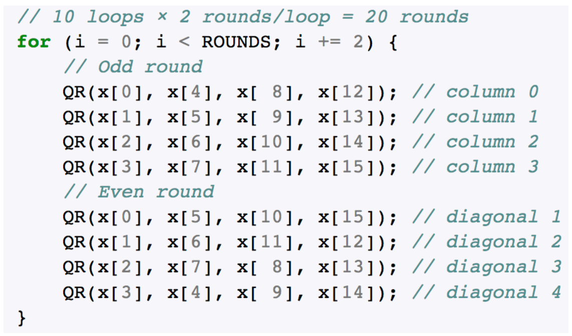
Fig. 2. The Chacha20 "QUARTER-ROUND" operation, in C-code.

Fig. 2 illustrates 12 lines of C-code, in the QUARTER-ROUND operation. In the first 3 lines, there are three operands (each a 32-bit word), "a", "b", and "d". The first 3 lines of C-code can be expanded, as follows:

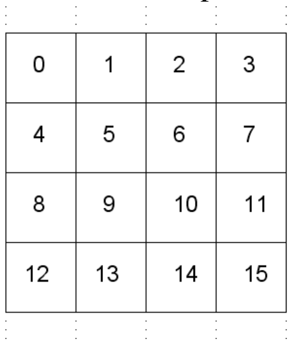
```
a = a + b                    (addition modulo 2^32, ie no carry bit generated):
d = XOR(d, a)                (32-bit logical XOR)
d = ROTATE_LEFT(d, 16)
```

<u>The Chacha20 DOUBLE-ROUND function:</u>

The Chacha20 DOUBLE-ROUND operation consists of 8 calls to the QUARTER-ROUND operation. The Chacha20 DOUBLE-ROUND operation will complete two "rounds" of computation, on each key-stream block, as shown below. The DOUBLE-ROUND operation is repeated in a loop for 10 times, thereby implementing 20 rounds of computation per block.



```
// 10 loops × 2 rounds/loop = 20 rounds
for (i = 0; i < ROUNDS; i += 2) {
    // Odd round
    QR(x[0], x[4], x[ 8], x[12]); // column 0
    QR(x[1], x[5], x[ 9], x[13]); // column 1
    QR(x[2], x[6], x[10], x[14]); // column 2
    QR(x[3], x[7], x[11], x[15]); // column 3
    // Even round
    QR(x[0], x[5], x[10], x[15]); // diagonal 1
    QR(x[1], x[6], x[11], x[12]); // diagonal 2
    QR(x[2], x[7], x[ 8], x[13]); // diagonal 3
    QR(x[3], x[4], x[ 9], x[14]); // diagonal 4
}
```



0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Fig. 3. Each "DOUBLE-ROUND" operation requires 8 "QUARTER-ROUND" operations.
The first 4 "QR" operations perform one "ROUND" on the 4 columns.
The last 4 "QR" operations perform one "ROUND" on the 4 diagonals

In this assignment, we will focus on implementing Chacha20, using basic RISC instructions. We will omit the initialization loop to create the initial-states for the B key-stream blocks for now - this is a simple loop with little computational effort. In this initialization loop, the 256-bit key, the 64-bit block counter, and the 64-bit nonce (a one-time random number), must be written into the B key-stream blocks. This initialization is straight-forward, so we will not consider it in this first assignment. We will skip the last step, which ADDs the key-stream block with the plaintext block, as this is also a simple loop with little computational effort.

The IETF has made a few small revisions to Chacha20, in 2015. According to the IETF, each key-stream block has a 256-bit key, a 32-bit block counter (rather than 64 bits), and a 96-bit nonce (rather than 64 bits). The IETF changes limit the size of the plaintext file to 256 Gigabytes.

To encode a message with 1,024 blocks, Chacha20 must create 1,024 blocks of key-stream, with the initial state shown in Fig. 1. (The block counter is incremented for each block.) The initial state is then encoded, by running the loop in Fig. 3 of the DOUBLE-ROUND operation, for 10 iterations. The net result is that 20 ROUNDS of computation are performed on each block of key-stream.

Assume the plaintext consists of 1,024 blocks stored in memory. Each block consists of 16 words, stored in linear order, ie 0, 1, 2, …, 15. Word 0 of block 0 of plaintext is stored in memory address 1,000,000.

Assume the initial key-stream consists of 1,024 blocks stored in memory. Each block consists of 16 words, stored in linear order, ie 0, 1, 2, …, 15. Word 0 of block 0 of key-stream is stored in memory address 2,000,000.

Assume your RISC code will have an outer loop to process all 1,024 blocks of key-stream data. The outer loop will contain an inner-loop, which implements 10 iterations of the DOUBLE-ROUND operation, per block.

Exercise Part (A) - The 7-stage RISC Pipeline.

Assume a 7-stage RISC pipeline, as shown in FIG. 4. This pipeline is similar to the 5-stage pipeline discussed in class. However, the Instruction-Fetch stage (IF) is split into 2 stages (IF1 and IF2 stages), since memory accesses tend to be slow. It thus takes 2 cc to fetch an instruction from memory, allowing for a faster clock rate. The memory stage is also split into 2 stages (MEM1 and MEM2), since memory accesses to be slow. It thus takes 2 cc to fetch data from memory.

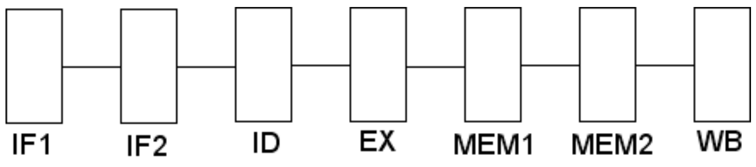


Fig. 4. A 7-stage RISC pipeline.

(A1) How many stalls (clock-cycles) occur, on LOAD instructions (when the instruction following a LOAD instruction uses the value being loaded) ? Use an excell timing diagram to illustrate your answer, and use an arrow (or asterisks *) to illustrate when data is FORWARDED. (Please see the class notes - the timing diagram has instructions along the y-axis, and clock-cycles along the x-axis. A sample excel file is posted on Avenue-to-Learn.)
• Add a comment for every forwarding of data or signals that occurs. A comment might be *'* forward R0 (EX* to *D) in cc 7'.*

(A2) How many branch-delay-slots (clock-cycles) occur, following a BRANCH instruction?

Make the same assumptions as in the class notes: Consider an instruction BNEZ R0, loop. Assume that branches are resolved as soon as possible, in the ID stage, without "stretching the clock". This leads to 2 cases to consider, as described in the class notes:

(1) If the value R0 being tested is available in a register at the start of the ID stage, then the "branch is resolved" in the ID stage - the ID stage will compare R0 with zero, and generate a "Take-Branch" signal which is sent back to the IF1 stage from the ID stage in clock-cycle 3. (See class notes in 4DM4 Lecture 10.)

(2) If the value R0 is being computed in the EX stage, by the previous instruction, and R0 needs to be forwarded to the ID stage, then we do not "stretch the clock" to allow R0 to be tested in the ID stage in the same clock cycle. The system will "stall" the BNEZ for 1 extra clock cycle, waiting for R0 to be computed and forwarded to the ID stage. (See class notes in 4DM4 Lecture 13a, Tutorial #4.)

(Use an excell timing diagram to illustrate your two timing diagrams for the 2 cases, and use an arrow (or asterisks *) to illustrate when any register values, and the "Take-Branch" signal, are FORWARDED. Please see the sample timing diagram on ATL - the excel timing diagram has instructions along the y-axis, and clock-cycles along the x-axis.)
• Add a comment for every forwarding of data or signals, ie *'* forward R0 (EX* to *D) in cc 7'.*

Exercise Part (B) - Generate RISC Code for The Chacha20 Stream Cipher

(B1) Create an excel timing-diagram for the un-optimized RISC code, of the first 3 lines of the QUARTER-ROUND operation. Use the 7-stage pipeline in part B. You may assume that a register (say R0) contains the address of the first word of the block. (Your RISC code in parts B1 to B4 should show how all memory addresses are calculated, so use explicit memory addresses in your RISC instructions, ie LOAD R1,0(R0), where the memory address is 0+R0).
• Highlight any assumptions that you make. Use a label, ie "ASSUMPTION #1" and explain it.

Assume the compiler will translate each line of C code into several RISC instructions, one at a time. Each line of C-code will generate several RISC instructions, including LOADs and STOREs. These RISC instructions will fetch 2 operands from memory, perform some work, then write one result back to memory. The compiler will then move onto the next line of C-code. This un-optimized code will have many redundant LOADs and STOREs, since no optimization has been performed. (Typical RISC instructions were shown in 4DM4 Lecture 7, and are summarized on the last page.)

SHOW ALL FORWARDING: In all the timing diagrams, show the forwarding of all data and signals. There are 2 ways to show the forwarding: (a) draw forwarding arrows onto the PDF file, (b) use a asterisk '*' at the source and destination of the forwarding, and add a comment to your excel file explaining each '*'.

• Add a comment for every forwarding, ie *'* forward R0 (EX* to *D) in cc 7'.*

(B2) Write the un-optimized RISC code, for one QUARTER-ROUND operation (with 12 lines of C-code as shown in Fig. 2). (Use your insights gathered from B1.)

Your answer can be a table with 3 columns: Column 1 is the RISC instruction. Column 2 is the number of stall cycles associated with that instruction. Column 3 can be a comment. You do not need to create a detailed timing-table.

• Add a comment for every line of code. (Clear and concise comments take thought, and are worth marks.)

(B3) Calculate the number of clock-cycles for the un-optimized QUARTER-ROUND in B2 to execute. (You can ignore the time to flush the pipeline.) Explain your answer clearly.

(B4) Optimize the QUARTER-ROUND code in B2, to minimize stalls. (The compiler will re-arrange instructions, to minimize stalls. The compiler will minimize the number of LOADs and STORES, to minimize stalls.) Create a compressed timing-diagram, to show how the QUARTER-ROUND will execute once it has been optimized. The timing-diagram will illustrate how many stalls occur. Explain how many clock cycles it takes, to execute the optimized QUARTER-ROUND operation.

In the compressed timing diagram, use comments to indicate all forwarding of data and signals. A comment might be *'* forward R0 (EX to D) in cc 7'.* (If you have already made an uncompressed timing diagram, you can use that instead.)

(B5) Generate the optimized RISC code for one ROUND which operates on the 4 COLUMNs of a block. One simple solution here is to repeat your optimized code from B4 for each column of the block, (ie repeat 4 times for 4 columns). (However, this code might not be optimized. You should optimize it, if possible.) Create a table with 3 columns, to show how one ROUND will execute once it has been optimized. Column 1 identifies a block of RISC code (it might be a QR from B4, or a subset of a QR from B4.) Column 2 illustrates the number of clock cycles, and stall cycles, associated with that block of RISC code. Column 3 can be a comment, as shown below. (Show the memory addresses of the variables A,B,C,D that you use, for each column.)

If you move any code around to optimize it, explain which code from which QR is moved, and explain where it is moved to.

Code	Clock cycles	Comment
Optimized Quarter-Round code from B4(A,B,C,D)	xx clock cycles to execute yy clock cycles for stalls	A,B,C,D represent addresses of words in 1st column

Explain how many clock cycles it takes, to execute the optimized ROUND operation. (You can ignore the time to flush the pipeline.) Explain your answer clearly; a numerical answer alone may not be sufficient.

(B6) Generate the optimized RISC code for the outer loop and inner loop of Chacha20. The outer loop will process B blocks of key-stream. The inner-loop will have 10 iterations, of the DOUBLE-ROUND operation in Fig 3. (You do not need to write out all the RISC code for the double-round, as you can verbally explain that you use the code from B5 for each round in the inner loop.)

Explain any of your optimizations: If you move any code around for optimization, please explain which lines of code were moved, explain where they were moved from, and where they were moved to.

Your answer can be a table with 3 columns: Column 1 can refer to a RISC instruction (or block of RISC code). Column 2 is the number of stall cycles associated with that instruction or block of RISC code. Column 3 can be a comment. You do not need to create a detailed timing-table. (In the 1st column, a RISC instruction could also say: "Insert RISC code from B5 for one round'. The number of stalls for that code can go into the 2nd column.)
• Add a comment for every line of code. (Clear and concise comments take thought, and are worth marks.)

Clearly explain how many clock cycles it takes, to process 1,024 blocks of data. Explain how many clock cycles it takes, to execute the optimized outer and inner loops for Chacha20. If any instructions cause stalls, identify these instructions, and explain how many stalls per instruction occur. (You can ignore the time to flush the pipeline.)
• If you make any assumptions, highlight those assumptions. Use a label ie "ASSUMPTION #1" and explain it.

(B7) Assume a 2.5 GHz clock rate. Assume a RISC machine that fetches 1 instruction at a time. How long does it take (in nanoseconds, microseconds, or milliseconds) to process 1,024 blocks, in part B5 ? Explain your answer clearly; a numerical answer alone may not be sufficient.

SUBMISSION:
Submit your report on Avenue-to-Learn, by the due date.

SUBMISSION FORMAT:
Create a report file called 4DM4-Assignment-1-report-XX-YY, where XX and YY are the initials of your team members.

Include your brief report in one PDF file, so the reader can see everything by opening one PDF file. Include your well-documented assembly-code, in the PDF file. If your space-time diagrams can fit into the PDF file without too much trouble, then include them. Otherwise, create a separate PDF file for each space-time diagram. They must be clearly labeled.

If you are taking 6DM4, please do the assignment alone.

ADDENDUM:

Here are some Typical RISC Instructions, shown as high-level assembly-code (without explicit memory addresses). See 4DM4 class notes for Lecture 7. (All of these instructions, except the XOR, where presented in L7.)

```
LW       Rb,b             % load word 'b' from memory into reguister Rb
LW       Rc, c            % load word
ADD      Ra, Rb, Rc       % default 32-bit add (with carry)
SW       a, Ra            % store word

SHIFT.L  Rd, Ra,  #3      % shift 32 bit word, 3 bits to the left, shift-in zeros

ADD.I    Rd, Ra, #7       % add immediate constant 7 to Ra, store in Rd

ROT.R    Rd, Ra, #9       % rotate 32 bit word in Ra, 9 bits to the right, store result in Rd

XOR      Rc, Ra, Rb       % XOR (Ra, Rb), and store result in Rc
```

•You can assume other reasonable RISC instructions. However, always highlight your assumptions, so that the TA can understand your work.

•You can use LD instead of LW, to load a 32 bit word, but this is an assumption that you make, so you must highlight it, so that the TAs can understand what you are doing. (LD often means "Load double-precision".)

•You can assume that all your ADD instructions do not have carry-out bits from the most significant bit. If you make this assumption, you must highlight it, so that the TAs understand what you are doing.

• Alternatively, you can assume an ADD.nc (add with no carry) instruction exists. This instruction omits the carry-out from the most significant bit, and you can use it.

ASSEMBLY-CODE MACROs.

• The 'high-level' assembly-code above cannot be executed by machine, since information is missing.

• Explicit memory addresses, for variables like "a" and "b", are missing.

• Explicit registers to use, for Ra and Rb, are not given.

• Lets assume that we have a "Macro-Pre-Processor", to exchange a text string for another text string.

• We can define macro like this (these go before the assembly-code):

• #define Ra => R1; Rb => R2; Rc => R3; …

• #define A => 0(R0); B => +8(R0); ….

• With these macros, the 'high-level' assembly-code shown above can be executed by a machine, since the missing information is provided in the macros.

• You do not need to use macos.

• However, if you are using high-level assembly code in your solutions, please add the macros, to show the missing information.