EE 789
Algorithmic Design of Digital Systems

Assignment 2

Matrix Multiplication

Name:	**Satvikkumar Patel**
Roll number:	23M1117
Program:	M.tech (EE5 - Electronic Systems)
Department:	Electrical Engineering
Date of Sub:	November 17, 2023

EE 789
Assignment 2: Matrix Multiplication

October 9, 2023

You are given two 16x16 matrices A and B and are expected to design a circuit which multiplies these matrices.

If $A = [a_{ij}]$ and $B = [b_{ij}]$ $(0 \leq i, j \leq 15)$ then the product matrix $C = A \times B$ has entries c_{ij} with

$$c_{ij} = \sum_{k=0}^{1} 5 a_{ik} b_{kj}$$

A 16x16 matrix can be viewed in row-form or column form. In row-form, we can write

$$A = \begin{pmatrix} rA_0^T \\ rA_1^T \\ \ldots \\ rA_15^T \end{pmatrix}$$

where rA_0^T is the first row and so on. In column form, we can write A as

$$A = (cA_0 \; cA_1 \; \ldots \; cA_15)$$

where cA_0^T is the first column and so on. Similarly for B.

The product matrix C can be built in different ways.

$$c_{ij} = rA_i^T . cB_j^T$$

or (as a sum of rank-1 matrices):

$$C = \sum_{i=0}^{15} cA_i . rB_j^T$$

1 Sample code

I will share sample code for the multiplication of two 16x16 matrices. This includes a dot-product based implementation and a test-bench.

2 Assignment

You will attempt to implement the mmul routine in various ways. The test-bench is unchanged.

- Implement the matrix multiplication by speeding up the dot product implementation used in the sample code (10).
- Parallelize the matrix multiplication by dividing A and B into four 8x8 blocks, doing work on the 8x8 blocks and then combining this work to produce the final result (10).
- Implement the matrix product using the sum of rank-1 matrices approach (10).

In each case, you will have to measure the time required by the mmul routine in your implementation.

Sample code:

```
$parameter ORDER 16

//
// observation signals to keep track of progress.
//
$pipe mmul_I : $uint<8> $signal
$pipe mmul_J : $uint<8> $signal
$pipe mmul_START: $uint<8> $signal
$pipe mmul_END: $uint<8> $signal

$storage A B C: $array[ORDER][ORDER] $of $uint<32>

$module [storeA] $in (I J: $uint<8> wval: $uint<32>) $out () $is
{
  A[I][J] := wval
}
$module [storeB] $in (I J: $uint<8> wval: $uint<32>) $out () $is
{
  B[I][J] := wval
}

$module [loadC] $in (I J : $uint<8>) $out (Y : $uint<32>) $is
{
  Y := C[I][J]
}


$module [dot_product] $in (I J: $uint<8>) $out (result: $uint<32>) $is
{
  $branchblock[loop] {

    $dopipeline $depth 31 $fullrate
      $merge $entry $loopback
        $phi K := $zero<8> $on $entry nK $on $loopback
        $phi SUM0 := ($bitcast ($uint<32>) 0)  $on $entry nSUM0 $on $loopback
        $phi SUM1 := ($bitcast ($uint<32>) 0)  $on $entry nSUM1 $on $loopback
      $endmerge
      $volatile nK := (K + 2)
      $volatile continue_flag := (K < (ORDER-2))
      $volatile K1 := (K+1)

      nSUM0 := (SUM0 + (A[I][K] * B[K][J]))
      nSUM1 := (SUM0 + (A[I][K1] * B[K1][J]))
    $while continue_flag
  } (nSUM0 => R0 nSUM1 => R1)
  $volatile result := (R0 + R1)
}

$module [mmul] $in () $out () $is
{
  mmul_START := 1
  $branchblock[loop] {
      u := mmul_START v := mmul_END // for logging
      $merge $entry I_loopback
      $phi I := $zero<8> $on $entry nI $on I_loopback
    $endmerge
    $volatile nI := (I + 1)

    $dopipeline $depth 3
```

```
      $merge $entry $loopback
        $phi J := $zero<8> $on $entry nJ $on $loopback
      $endmerge
      $volatile nJ := (J + 1)
      $volatile continue_flag := (J < (ORDER - 1))

      // dot-product!
      C[I][J] := ($call dot_product (I J))

      mmul_I := I
      mmul_J := J

    $while continue_flag

      $if (I < (ORDER-1)) $then $place [I_loopback] $endif
  }
  mmul_END := 1
}
```

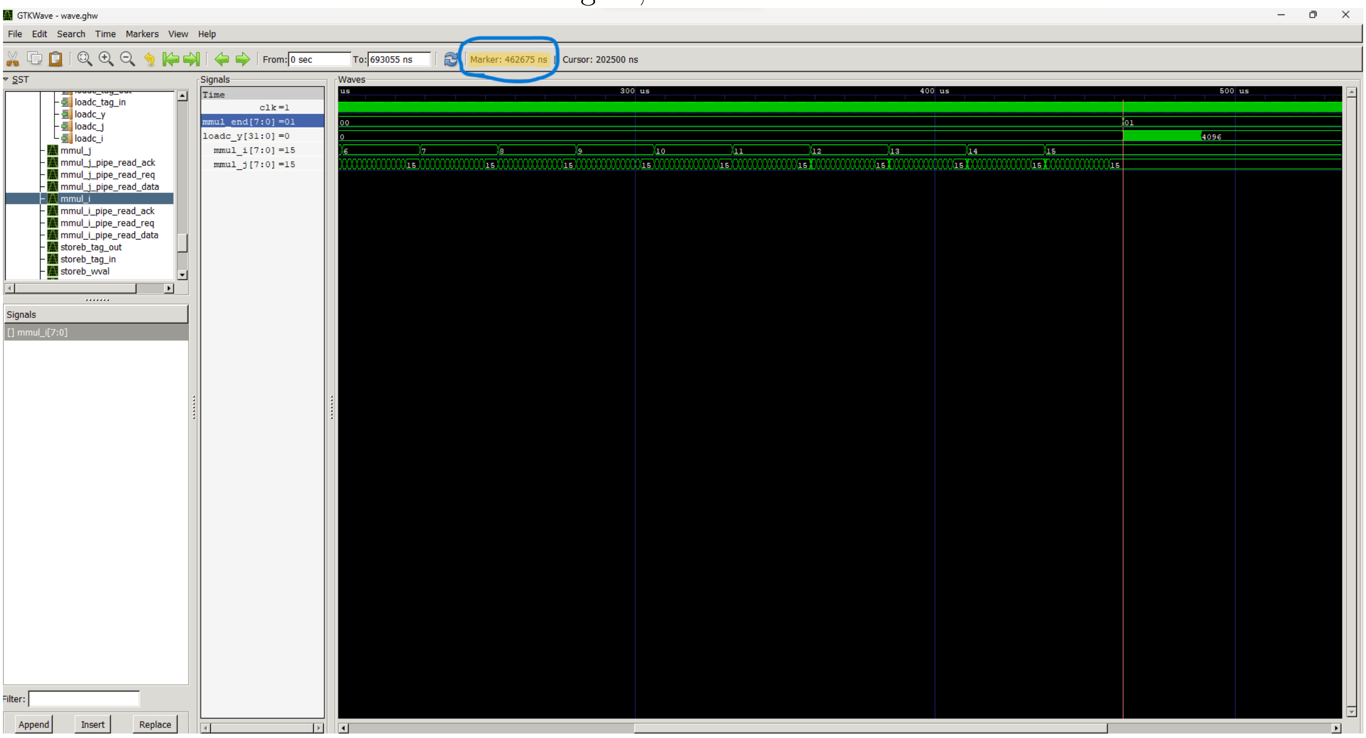
TestBench:

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <pthread.h>
#include <pthreadUtils.h>
#include <Pipes.h>
#include <pipeHandler.h>
#ifndef SW
#include "vhdlCStubs.h"
#endif

#define ORDER 16

// Give inputs as
// Matrix A:
// 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
// 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
// 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
// 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
// ...

// Matrix B:
// 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
// 2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2
// 3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3
// 4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4
// ...


int main(int argc, char* argv[])
{
  int I, J;
  for(I = 0; I < ORDER; I++)
  {
    for(J = 0; J < ORDER; J++)
    {
      storeA(I, J,(uint32_t)  I+1);
      storeB(I, J,(uint32_t)  J+1);
    }
  }
```

```
  fprintf(stderr,"Stored A, B\n");

  mmul();

  fprintf(stderr,"finished dot_product, results:\n");
  for(I = 0; I < ORDER; I++)
  {
    for(J = 0; J < ORDER; J++)
    {
      uint32_t result = loadC (I,J);
      fprintf(stderr,"C[%d][%d] = %d.\n",I, J, result);

    }
  }

  return(0);
}
```

Results:

The time taken for the mmul_end variable to go 1, is 462675ns



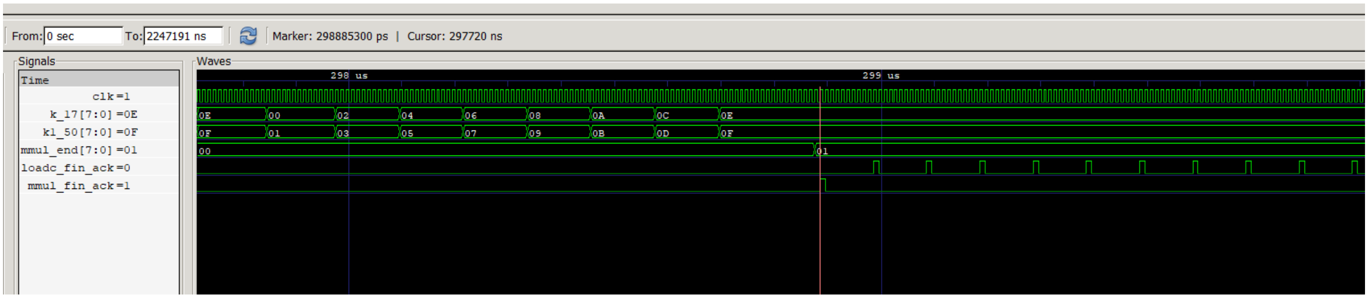
Faster Dot product:

The method used here is unrolling the pipe by 2. The matrix multiplication part is left as it is.

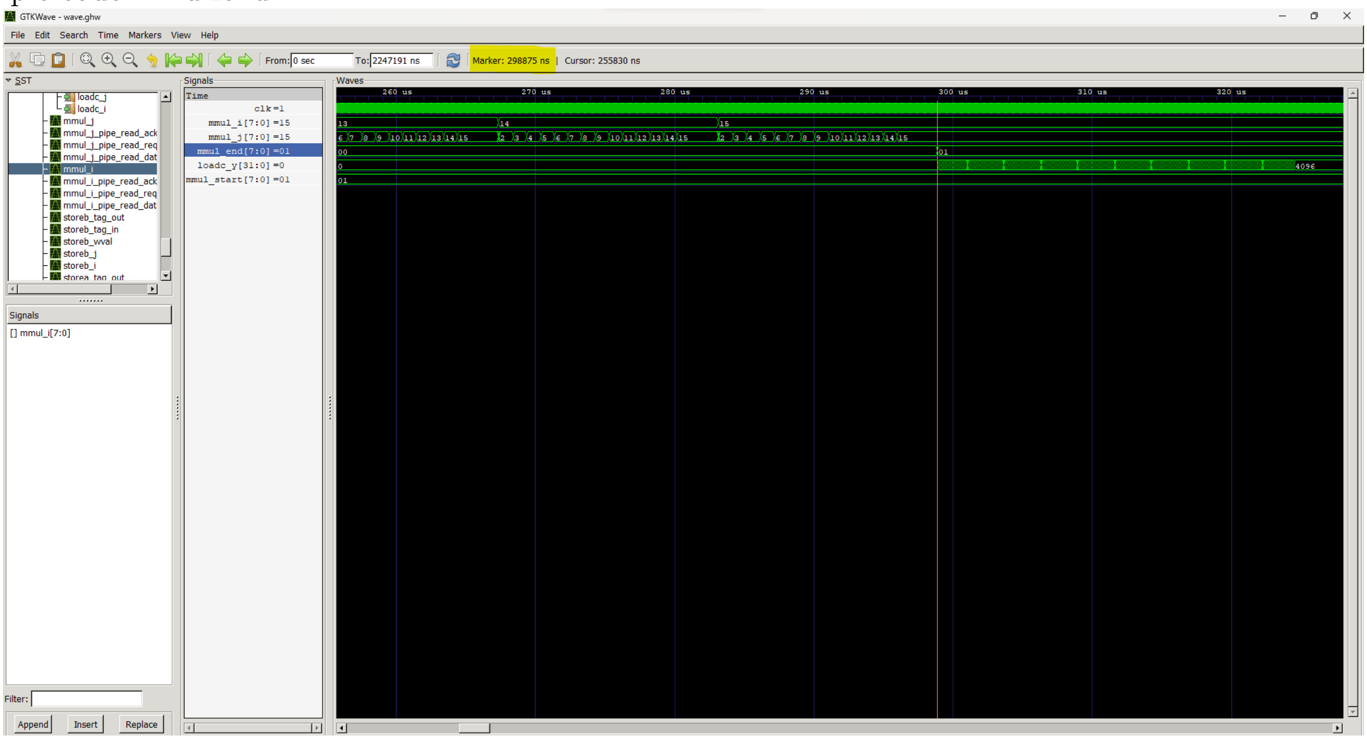
```
$parameter ORDER 16

//
// observation signals to keep track of progress.
//
$pipe mmul_I : $uint<8> $signal
$pipe mmul_J : $uint<8> $signal
$pipe mmul_START: $uint<8> $signal
$pipe mmul_END: $uint<8> $signal

$storage A B C: $array[ORDER][ORDER] $of $uint<32>

$module [storeA] $in (I J: $uint<8> wval: $uint<32>) $out () $is
{
  A[I][J] := wval
}
$module [storeB] $in (I J: $uint<8> wval: $uint<32>) $out () $is
{
  B[I][J] := wval
}

$module [loadC] $in (I J : $uint<8>) $out (Y : $uint<32>) $is
{
  Y := C[I][J]
}


$module [dot_product] $in (I J: $uint<8>) $out (result: $uint<32>) $is
{
  $branchblock[loop] {

    $dopipeline $depth 31 $fullrate
      $merge $entry $loopback
        $phi K := $zero<8> $on $entry nK $on $loopback
        $phi SUM0 := ($bitcast ($uint<32>) 0)  $on $entry nSUM0 $on $loopback
        $phi SUM1 := ($bitcast ($uint<32>) 0)  $on $entry nSUM1 $on $loopback
      $endmerge
      $volatile nK := (K + 2)
      $volatile continue_flag := (K < (ORDER-2))
      $volatile K1 := (K+1)

      nSUM0 := (SUM0 + (A[I][K] * B[K][J]))
      nSUM1 := (SUM0 + (A[I][K1] * B[K1][J]))
    $while continue_flag
  } (nSUM0 => R0 nSUM1 => R1)
  $volatile result := (R0 + R1)
}

$module [mmul] $in () $out () $is
{
  mmul_START := 1
  $branchblock[loop] {
      u := mmul_START v := mmul_END // for logging
      $merge $entry I_loopback
      $phi I := $zero<8> $on $entry nI $on I_loopback
    $endmerge
    $volatile nI := (I + 1)

```

```
    $dopipeline $depth 3
      $merge $entry $loopback
        $phi J := $zero<8> $on $entry nJ $on $loopback
      $endmerge
      $volatile nJ := (J + 1)
      $volatile continue_flag := (J < (ORDER - 1))

      // dot-product!
      C[I][J] := ($call dot_product (I J))

      mmul_I := I
      mmul_J := J

    $while continue_flag

      $if (I < (ORDER-1)) $then $place [I_loopback] $endif
  }
  mmul_END := 1
}
```

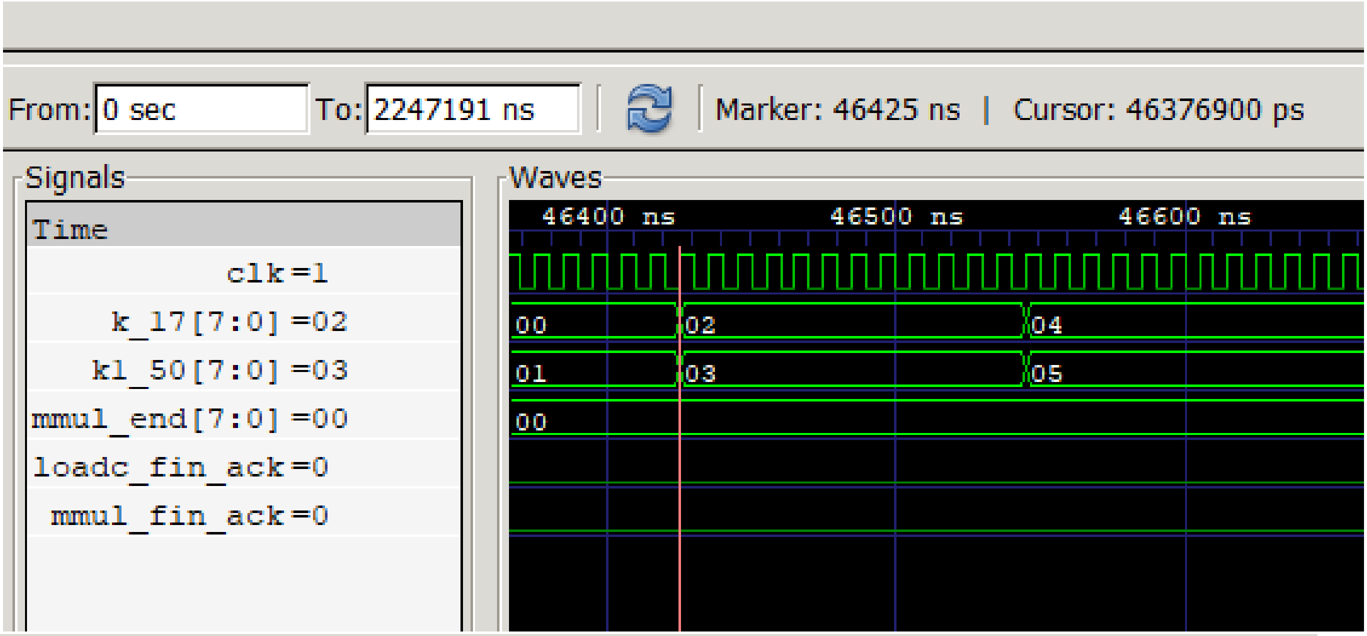
Results:

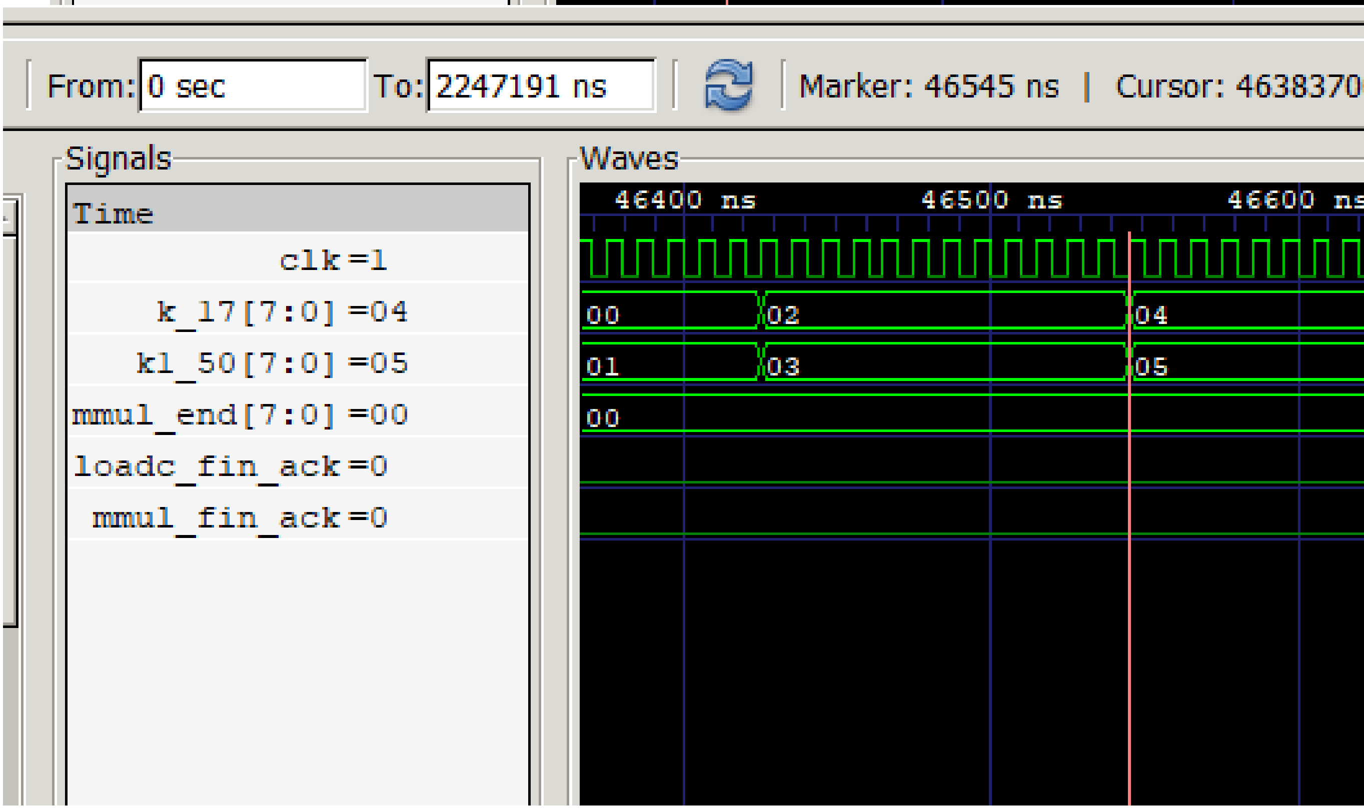


The time taken for the mmul_end variable to go 1, is $298875ns$, which is lesser compared to sample code mmul_end.



The time taken for the loop variable to count up by 1 is, $46545 - 46425 = 120ns$. The clock time period being 10ns, it takes 12 cycles for the loop interval to count up by 1.





Matrix Multiplication by dividing the matrix

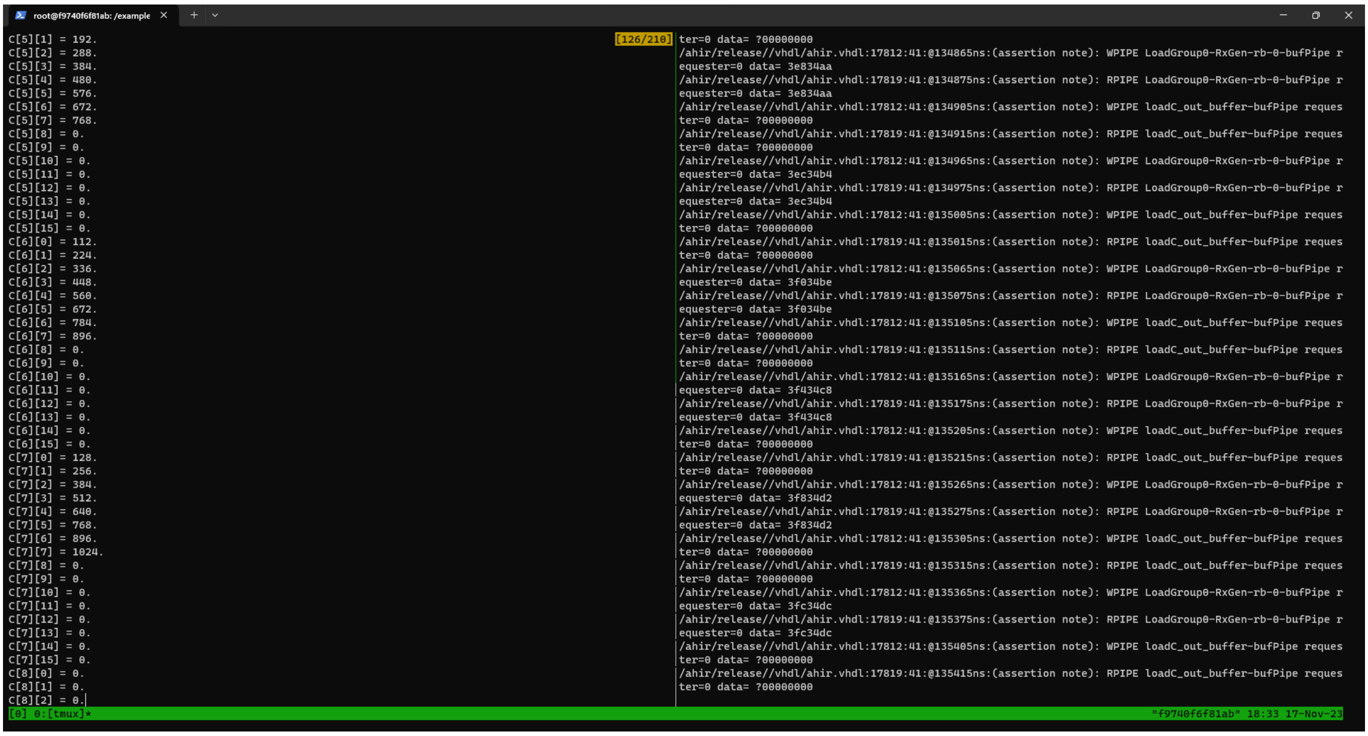
```
$parameter ORDER 16
$parameter ORDER_by_2 8

//
// observation signals to keep track of progress.
//
$pipe mmul_I : $uint<8> $signal
$pipe mmul_J : $uint<8> $signal
$pipe mmul_START: $uint<8> $signal
$pipe mmul_END: $uint<8> $signal

$storage A B C: $array[ORDER][ORDER] $of $uint<32>

$module [storeA] $in (I J: $uint<8> wval: $uint<32>) $out () $is
{
  A[I][J] := wval
}
$module [storeB] $in (I J: $uint<8> wval: $uint<32>) $out () $is
{
  B[I][J] := wval
}

$module [loadC] $in (I J : $uint<8>) $out (Y : $uint<32>) $is
{
  Y := C[I][J]
}


$module [dot_product] $in (I J: $uint<8>) $out (result: $uint<32>) $is
{
  $branchblock[loop] {

    $dopipeline $depth 31 $fullrate
      $merge $entry $loopback
        $phi K := $zero<8> $on $entry nK $on $loopback
        $phi SUM0 := ($bitcast ($uint<32>) 0)  $on $entry nSUM0 $on $loopback
        $phi SUM1 := ($bitcast ($uint<32>) 0)  $on $entry nSUM1 $on $loopback
      $endmerge
      $volatile nK := (K + 2)
      $volatile continue_flag := (K < (ORDER-2))
      $volatile K1 := (K+1)

      nSUM0 := (SUM0 + (A[I][K] * B[K][J]))
      nSUM1 := (SUM0 + (A[I][K1] * B[K1][J]))
    $while continue_flag
  } (nSUM0 => R0 nSUM1 => R1)
  $volatile result := (R0 + R1)
}

$module [mmul] $in () $out () $is
{
  mmul_START := 1
  $branchblock[loop1] {
      u := mmul_START v := mmul_END // for logging
      $merge $entry I_loopback
      $phi I := $zero<8> $on $entry nI $on I_loopback
    $endmerge
    $volatile nI := (I + 1)

```

```
    $dopipeline $depth 3
      $merge $entry $loopback
        $phi J := $zero<8> $on $entry nJ $on $loopback
      $endmerge
      $volatile nJ := (J + 1)
      $volatile continue_flag := (J < (ORDER_by_2 - 1))

      // dot-product!
      C[I][J] := ($call dot_product (I J))

      mmul_I := I
      mmul_J := J

    $while continue_flag

      $if (I < (ORDER_by_2-1)) $then $place [I_loopback] $endif
  }


  $branchblock[loop2] {
      u := mmul_START v := mmul_END // for logging
      $merge $entry I_loopback
      $phi I := $zero<8> $on $entry nI $on I_loopback
    $endmerge
    $volatile nI := (I + 1)

    $dopipeline $depth 3
      $merge $entry $loopback
        $phi J := $zero<8> $on $entry nJ $on $loopback
      $endmerge
      $volatile nJ := (J + 1)
      $volatile continue_flag := (J < (ORDER_by_2 - 1))
      $volatile I_1 := (I + 8)
      $volatile J_1 := (J + 8)

      // dot-product!
      C[I_1][J_1] := ($call dot_product (I_1 J_1))

      mmul_I := I
      mmul_J := J

    $while continue_flag

      $if (I < (ORDER_by_2-1)) $then $place [I_loopback] $endif
  }

  $branchblock[loop3] {
      u := mmul_START v := mmul_END // for logging
      $merge $entry I_loopback
      $phi I := $zero<8> $on $entry nI $on I_loopback
    $endmerge
    $volatile nI := (I + 1)

    $dopipeline $depth 3
      $merge $entry $loopback
        $phi J := $zero<8> $on $entry nJ $on $loopback
      $endmerge
      $volatile nJ := (J + 1)
      $volatile continue_flag := (J < (ORDER_by_2 - 1))
      //$volatile I_1 := (I + 8)
      $volatile J_1 := (J + 8)

```

```
      // dot-product!
      C[I][J_1] := ($call dot_product (I J_1))

      mmul_I := I
      mmul_J := J

    $while continue_flag

      $if (I < (ORDER_by_2-1)) $then $place [I_loopback] $endif
  }

  $branchblock[loop4] {
      u := mmul_START v := mmul_END // for logging
      $merge $entry I_loopback
      $phi I := $zero<8> $on $entry nI $on I_loopback
    $endmerge
    $volatile nI := (I + 1)

    $dopipeline $depth 3
      $merge $entry $loopback
        $phi J := $zero<8> $on $entry nJ $on $loopback
      $endmerge
      $volatile nJ := (J + 1)
      $volatile continue_flag := (J < (ORDER_by_2 - 1))
      $volatile I_1 := (I + 8)
      //$volatile J_1 := (J + 8)

      // dot-product!
      C[I_1][J] := ($call dot_product (I_1 J))

      mmul_I := I
      mmul_J := J

    $while continue_flag

      $if (I < (ORDER_by_2-1)) $then $place [I_loopback] $endif
  }

  mmul_END := 1
}
```

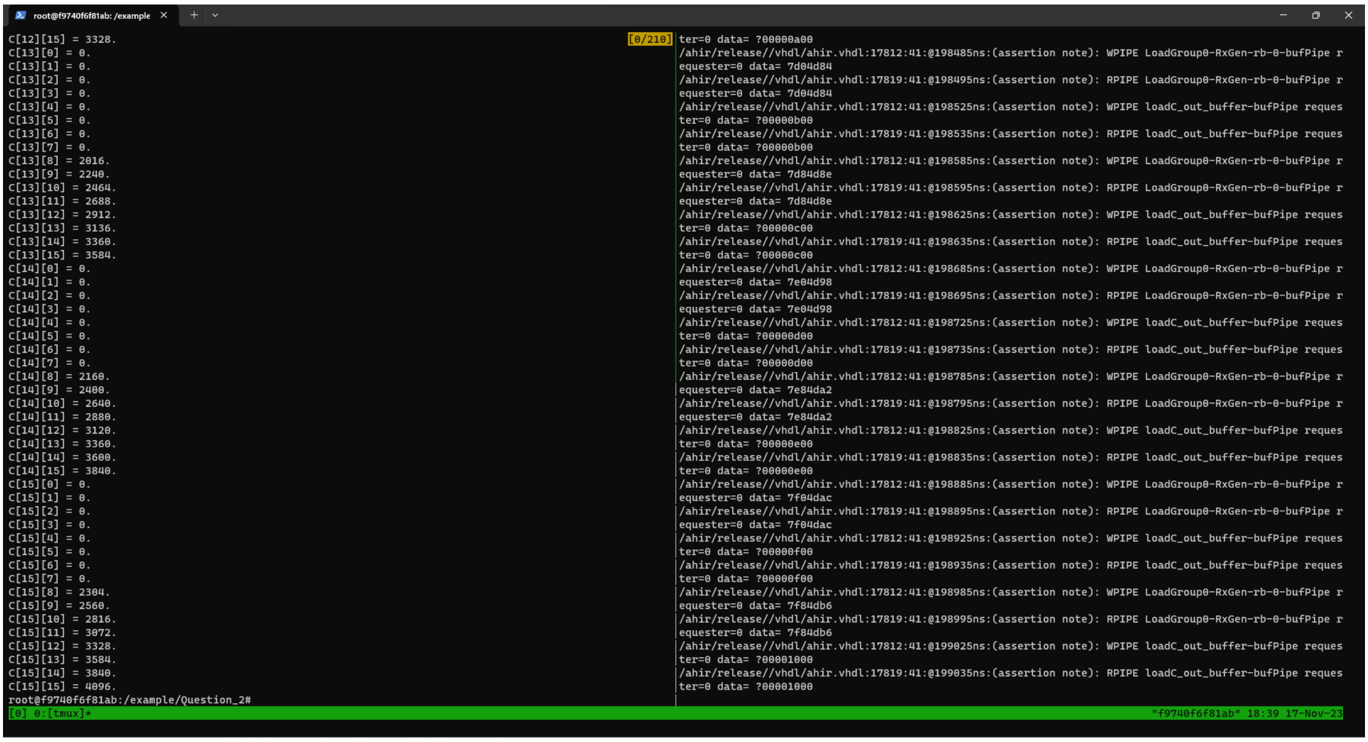
Four branch blocks are used, in order to find the resultant multiplication matrix. - The loop 1 is finding the top left matrix.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{18} \\ a_{21} & a_{22} & \cdots & a_{28} \\ \vdots & \vdots & \ddots & \vdots \\ a_{81} & a_{82} & \cdots & a_{88} \end{bmatrix}$$



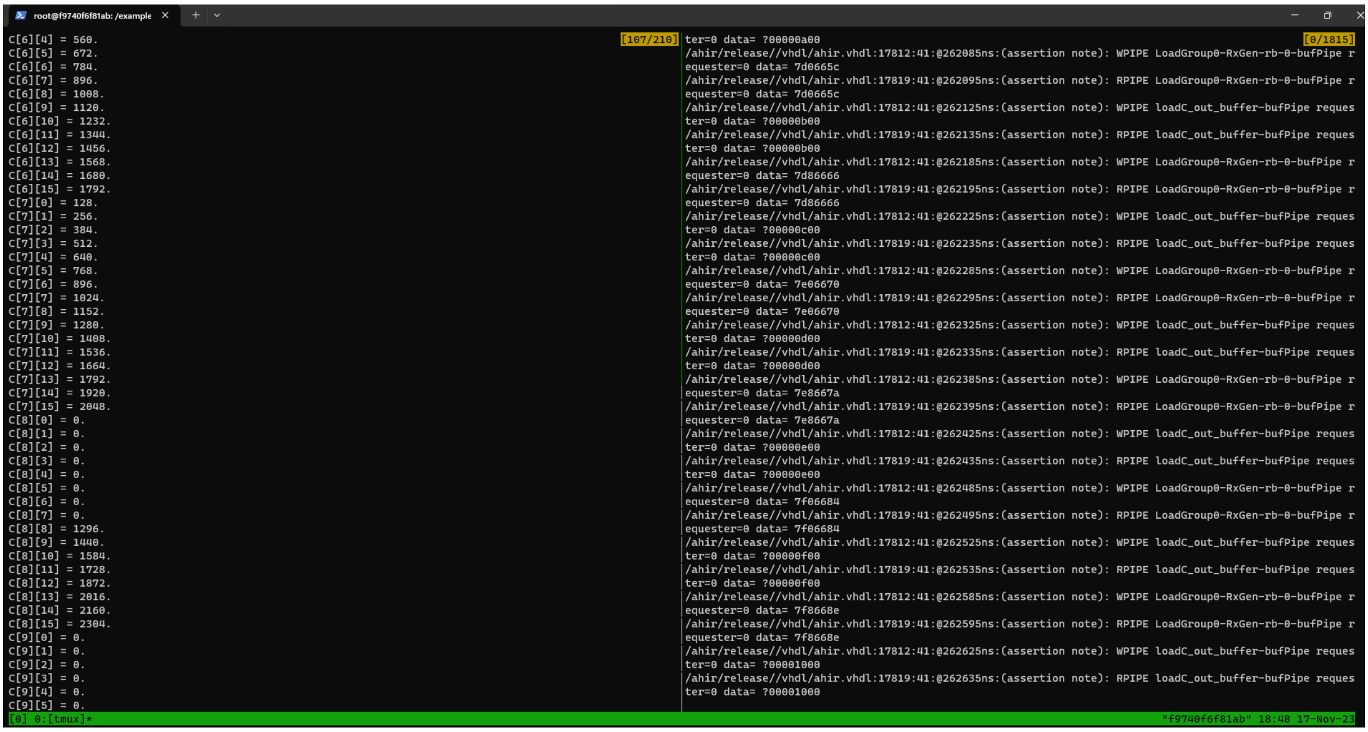
- The loop 2 is finding the bottom right matrix.

$$\begin{bmatrix} a_{919} & a_{920} & \cdots & a_{916} \\ a_{1019} & a_{1020} & \cdots & a_{1016} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1619} & a_{1620} & \cdots & a_{1616} \end{bmatrix}$$



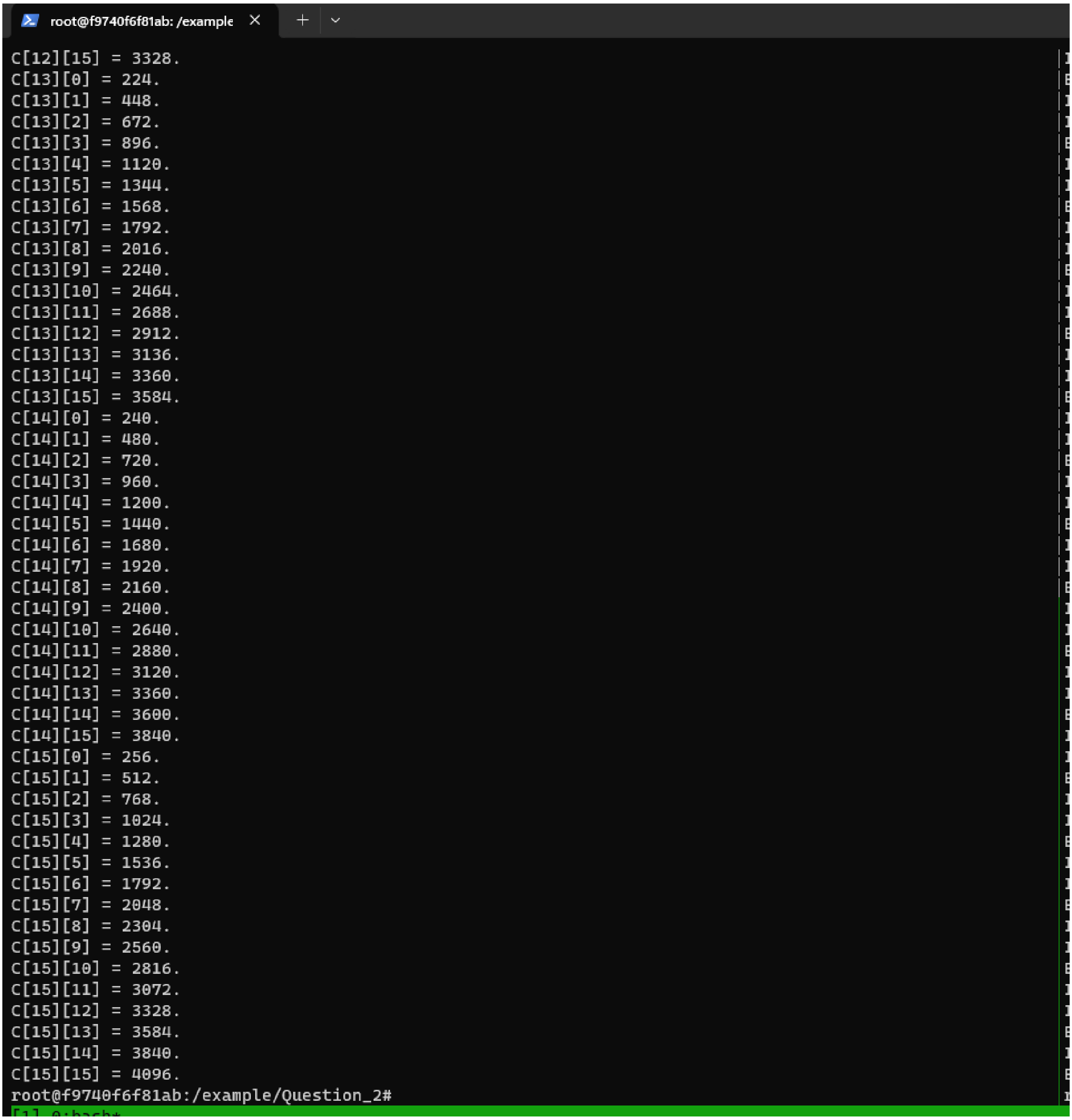
- The loop 3 is finding bottom left matrix.

$$\begin{bmatrix} a_{911} & a_{912} & \cdots & a_{98} \\ a_{1011} & a_{1012} & \cdots & a_{108} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1611} & a_{1612} & \cdots & a_{168} \end{bmatrix}$$



- The loop 4 is finding top right matrix.

$$\begin{bmatrix} a_{19} & a_{110} & \cdots & a_{116} \\ a_{29} & a_{210} & \cdots & a_{216} \\ \vdots & \vdots & \ddots & \vdots \\ a_{89} & a_{810} & \cdots & a_{816} \end{bmatrix}$$



```
root@f9740f6f81ab: /example
C[12][15] = 3328.
C[13][0] = 224.
C[13][1] = 448.
C[13][2] = 672.
C[13][3] = 896.
C[13][4] = 1120.
C[13][5] = 1344.
C[13][6] = 1568.
C[13][7] = 1792.
C[13][8] = 2016.
C[13][9] = 2240.
C[13][10] = 2464.
C[13][11] = 2688.
C[13][12] = 2912.
C[13][13] = 3136.
C[13][14] = 3360.
C[13][15] = 3584.
C[14][0] = 240.
C[14][1] = 480.
C[14][2] = 720.
C[14][3] = 960.
C[14][4] = 1200.
C[14][5] = 1440.
C[14][6] = 1680.
C[14][7] = 1920.
C[14][8] = 2160.
C[14][9] = 2400.
C[14][10] = 2640.
C[14][11] = 2880.
C[14][12] = 3120.
C[14][13] = 3360.
C[14][14] = 3600.
C[14][15] = 3840.
C[15][0] = 256.
C[15][1] = 512.
C[15][2] = 768.
C[15][3] = 1024.
C[15][4] = 1280.
C[15][5] = 1536.
C[15][6] = 1792.
C[15][7] = 2048.
C[15][8] = 2304.
C[15][9] = 2560.
C[15][10] = 2816.
C[15][11] = 3072.
C[15][12] = 3328.
C[15][13] = 3584.
C[15][14] = 3840.
C[15][15] = 4096.
root@f9740f6f81ab:/example/Question_2#
```

Results:

The time difference in finishing the matrix multiplication is can be seen. The right side waveforms are of the sample code while the left side waveforms is the modified matrix multiplication algorithm. The sample code mmul_end is being reported at $462675ns$ and the modified code mmul_end is reported at $300625ns$.