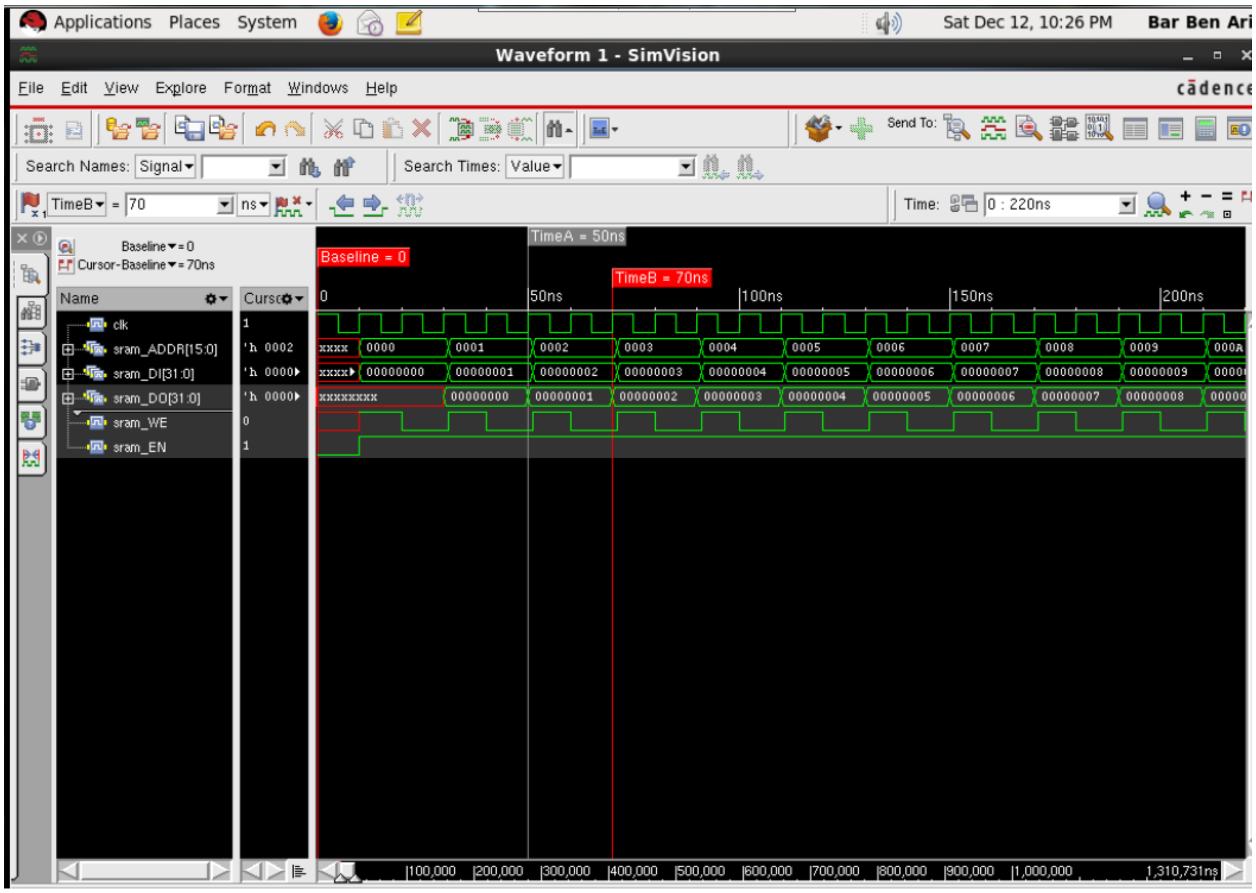
Onn Rengingad 304845951
Barak Levy 311431894
Bar Ben Ari 204034284

Lab 4 - Advanced Computer Architecture

Question 1: SRAM verification

In the following waveform screenshot it is possible to notice that a write transaction followed by a read transection of the same data has been successful:



TimeA shows the value i=1 was written successfully using sram_DI, and then at TimeB the same value was read successfully and was outputted on sram_DO, while both were accessed in sram_ADDR=i=1.
The requested sram.v & sram_tb.v files are attached in the .tar compressed file.

Question 2: processor implementation (no DMA)

The requested modified sram.v, alu.v, ctl.v, sp.v & top.v files are attached in the .tar compressed file.

Question 3: verification #1: example.bin

In this section we perform the first verification of the simple processor HDL implementation. The assembly code placed in example.bin was read by modified top.v, and the two output files

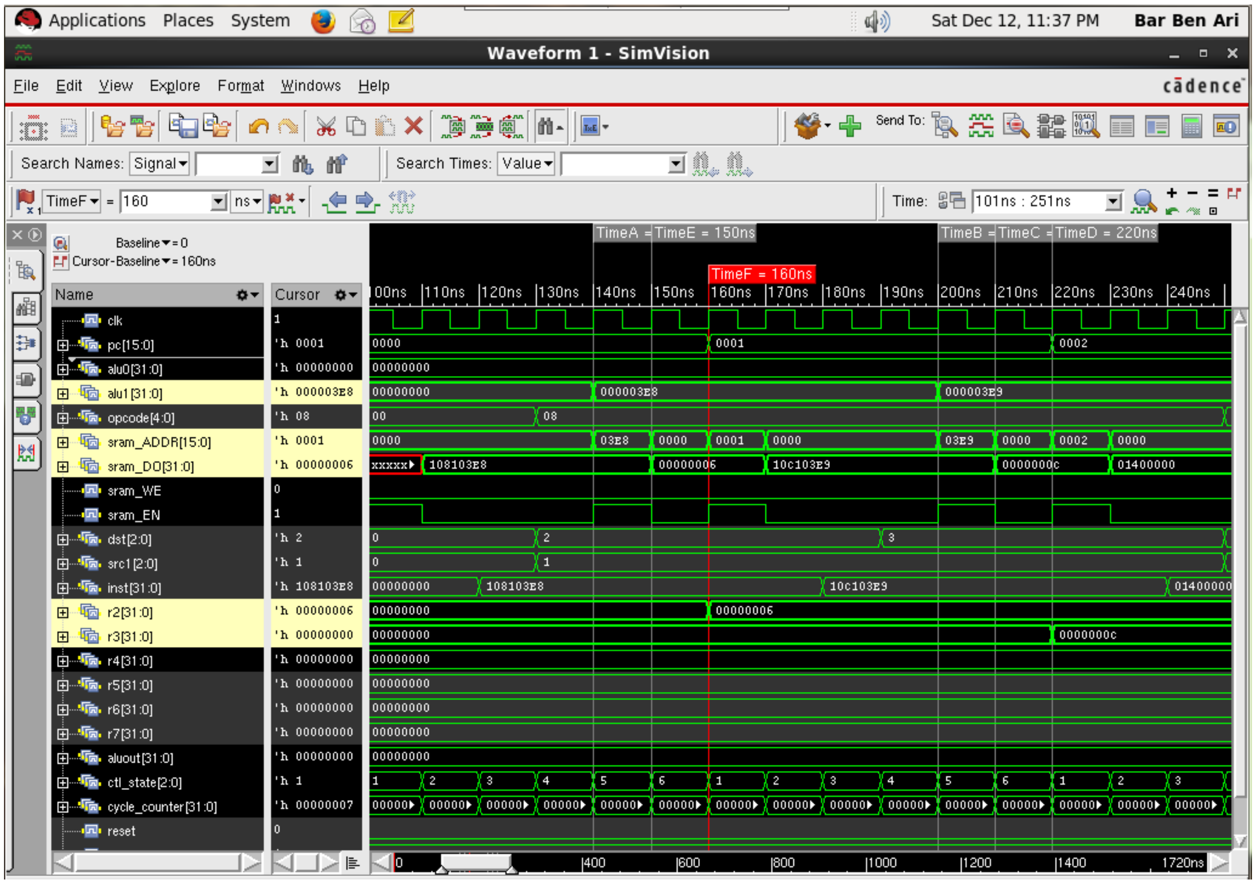
Onn Rengingad 304845951
Barak Levy 311431894
Bar Ben Ari 204034284

verilog_trace.txt & verilog_sram_out.txt were compared with the text files given to us. The two pairs of files were identical (Success - positive verification of example.bin).

Question 4: verification #2: multiplying two signed numbers

In this section we perform a second verification of the simple processor HDL implementation on mult.asm assembly code, by running a simulation with the binary code translation, mult.bin. Following running the simulation, we will compare the (cycle) trace & sram_out text files to those created by the lower-level single processor simulator in C language.

Waveform screenshot 1: Numbers being read into registers from memory

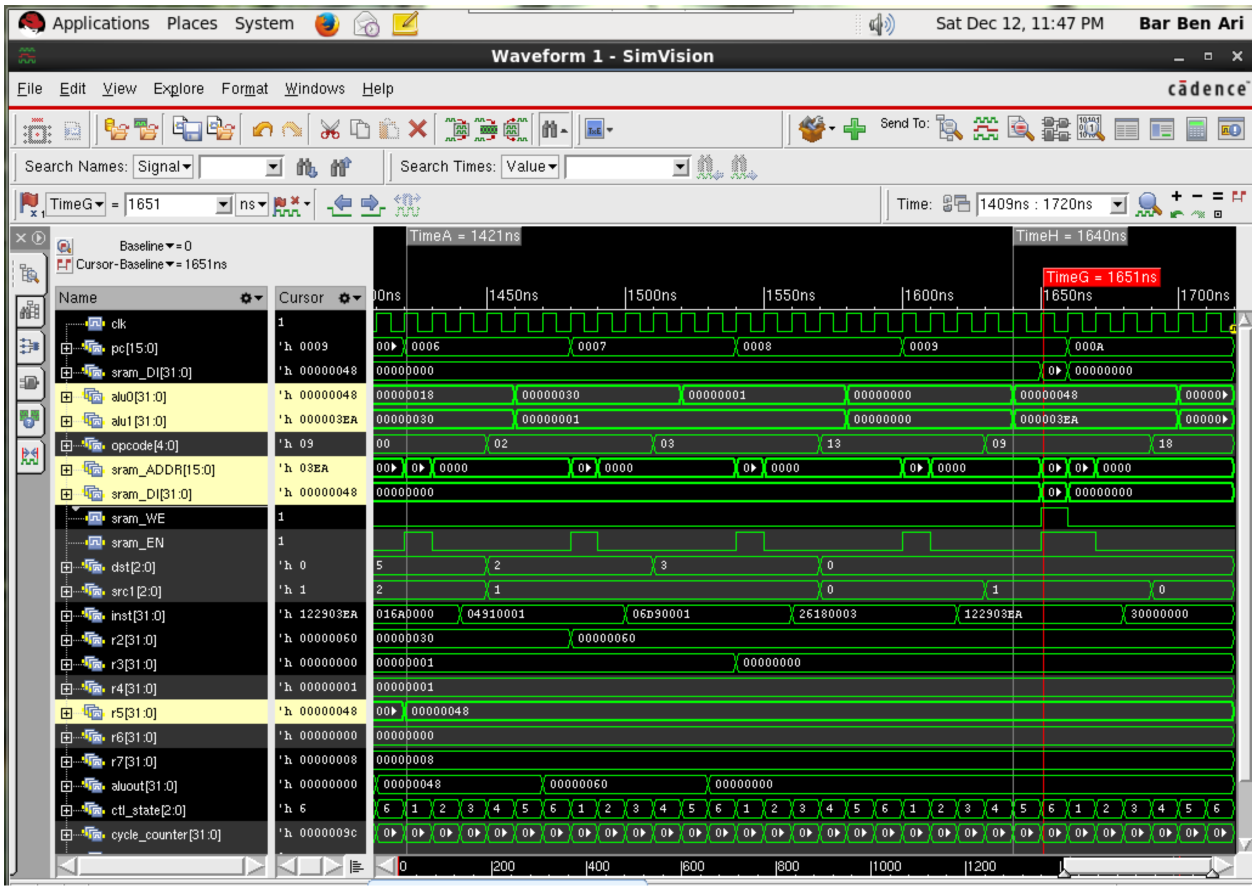


Cursors 'TimeA', 'TimeE' and 'TimeF' indicate different elements of the loading of the first number into register 'r2' (that equals 6 in this run), and 'TimeB', 'TimeC' and 'TimeD' indicate different element of the loading of the second number into register 'r3' (That equals 12 in this run). 'TimeA' shows when sram_ADDR was set to address 1000 (3E8), 'TimeE' shows when the Sram unit received the needed address (from sram_ADDR) and drew from the memory slot, the first number, into sram_DO (the output register of the Sram unit) and 'TimeF' shows when register 'r2' was updated with the value placed in sram_DO (6).

Onn Rengingad 304845951
Barak Levy 311431894
Bar Ben Ari 204034284

‘TimeB’ shows when sram_ADDR was set to address 1001 (3E9), ‘TimeC’ shows when the Sram unit received the needed address (from sram_ADDR) and drew from the memory slot, the second number, into sram_DO (the output register of the Sram unit) and ‘TimeD’ shows when register ‘r3’ was updated with the value placed in sram_DO (C).

<u>Waveform screenshot 2: Correct result being written back to memory</u>



Name	Cursor
clk	1
pc[15:0]	'h 0009
sram_DI[31:0]	'h 00000048
alu0[31:0]	'h 00000048
alu1[31:0]	'h 000003EA
opcode[4:0]	'h 09
sram_ADDR[15:0]	'h 03EA
sram_DI[31:0]	'h 00000048
sram_WE	1
sram_EN	1
dst[2:0]	'h 0
src1[2:0]	'h 1
inst[31:0]	'h 122903EA
r2[31:0]	'h 00000060
r3[31:0]	'h 00000000
r4[31:0]	'h 00000001
r5[31:0]	'h 00000048
r6[31:0]	'h 00000000
r7[31:0]	'h 00000008
aluout[31:0]	'h 00000000
ctl_state[2:0]	'h 6
cycle_counter[31:0]	'h 0000009c

Cursors ‘TimeA’, ‘TimeH’ and ‘TimeG’ indicate different elements of the storing of the outcome of the multiplication from register ‘r5’ into the memory address 3EA.
‘TimeA’ shows when register ‘r5’ was loaded with the multiplication data - outcome 72 (48 in Hex), ‘TimeH’ shows when the ALU unit’s registers were updated with the outcome memory address value (alu1) & with the outcome value itself (alu0) and ‘TimeG’ shows when the sram_ADDR (mem address) was updated by the ALU register alu1, and sram_DI (data value) was updated by the ALU register alu0, and at the same processor cycle the data was written to the memory as needed.

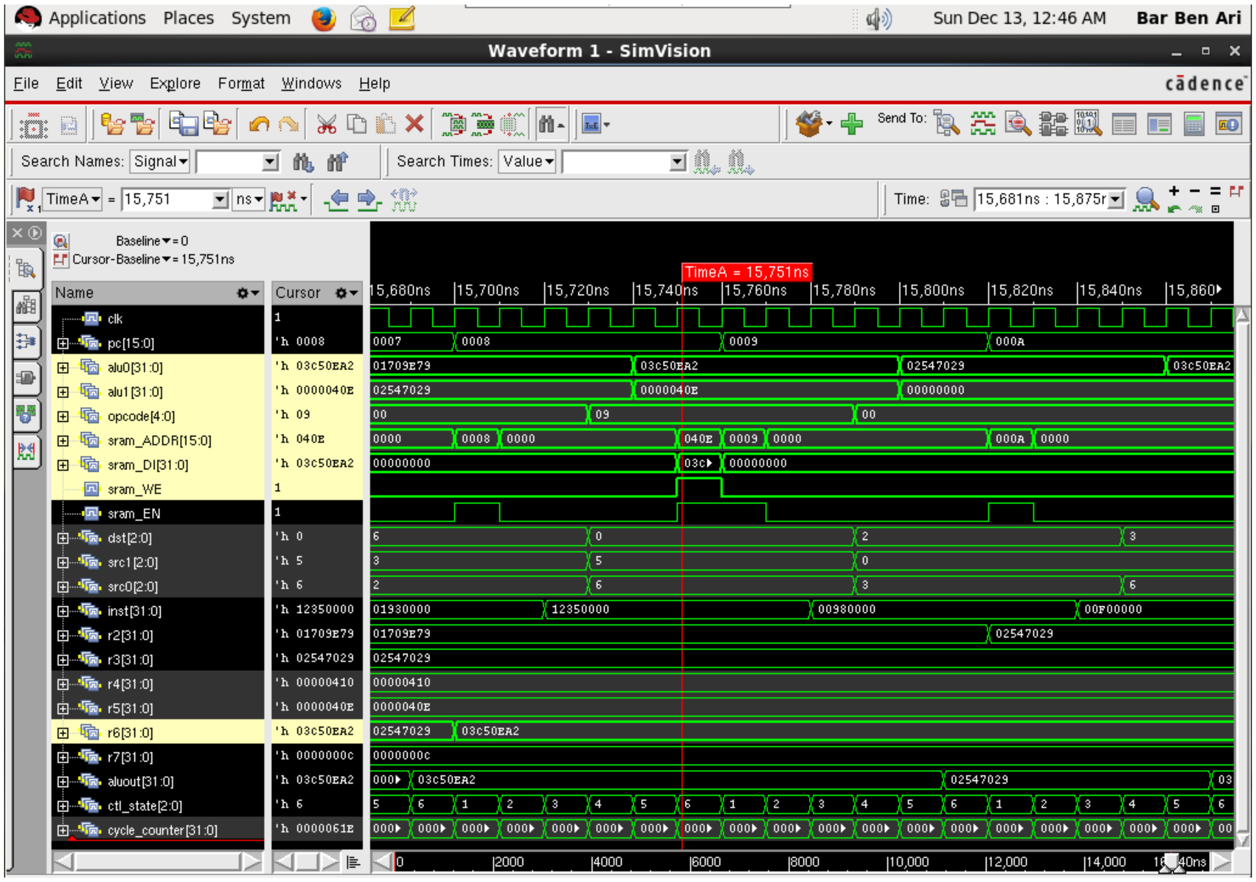
The outputted mult_verilog_trace.txt was compared with experiment 2’s mult.bin’s cycle trace text file, and the both were validated to be identical successfully, as needed.

Onn Rengingad 304845951
Barak Levy 311431894
Bar Ben Ari 204034284

Question 5: verification #3: Fibonacci

In this section we perform a third verification of the simple processor HDL implementation on fib.asm assembly code, by running a simulation with the binary code translation, fib.bin. Following running the simulation, we will compare the (cycle) trace & sram_out text files of this verilog implementation, to those created by the lower-level single processor simulator in C language.

<u>Waveform screenshot 1: The one-before-last fibonacci sequence value writing to memory</u>

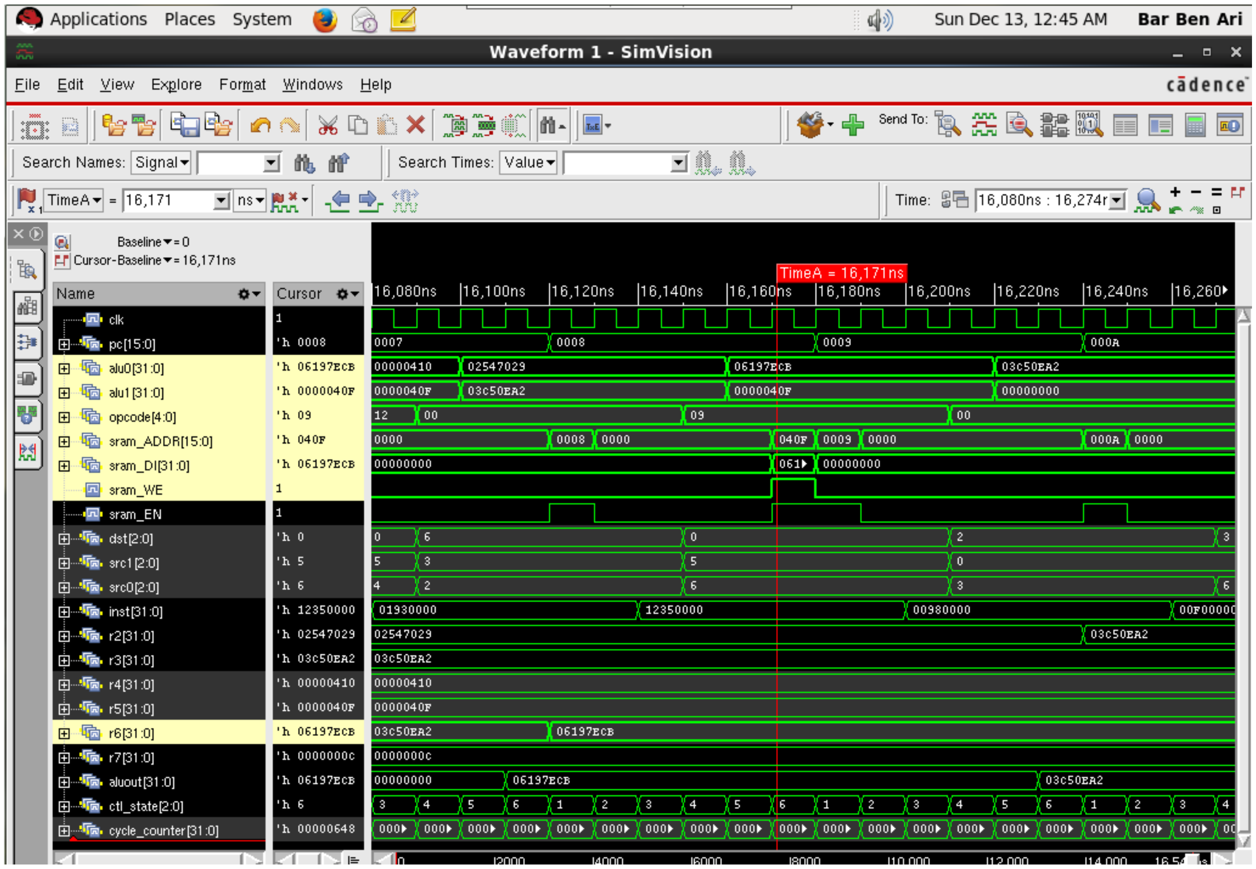


Name	Cursor
clk	1
pc[15:0]	'h 0008
alu0[31:0]	'h 03c50EA2
alu1[31:0]	'h 0000040E
opcode[4:0]	'h 09
sram_ADDR[15:0]	'h 040E
sram_DI[31:0]	'h 03c50EA2
sram_WE	1
sram_EN	1
dst[2:0]	'h 0
src1[2:0]	'h 5
src0[2:0]	'h 6
inst[31:0]	'h 12350000
r2[31:0]	'h 01709E79
r3[31:0]	'h 02547029
r4[31:0]	'h 00000410
r5[31:0]	'h 0000040E
r6[31:0]	'h 03c50EA2
r7[31:0]	'h 0000000c
aluout[31:0]	'h 03c50EA2
ctl_state[2:0]	'h 6
cycle_counter[31:0]	'h 0000061E

From time 15,700 nSec (when ctl_state appears to be 1 and beginning to execute another instruction), until 15,760 nSec a memory store operation (of the one-before-last fib sequence element) was performed by the single processor.
Cursor 'TimeA' shows when sram_ADDR was updated with the memory address value from the ALU unit's register alu1, and sram_DI was updated with the one-before-last fibonacci sequence element's value from the ALU unit's register alu0 (The was updated with the data inside register 'r6'.

Onn Rengingad 304845951
Barak Levy 311431894
Bar Ben Ari 204034284

Waveform screenshot 2: The last fibonacci sequence value writing to memory



Waveform 1 - SimVision

TimeA = 16,171 ns

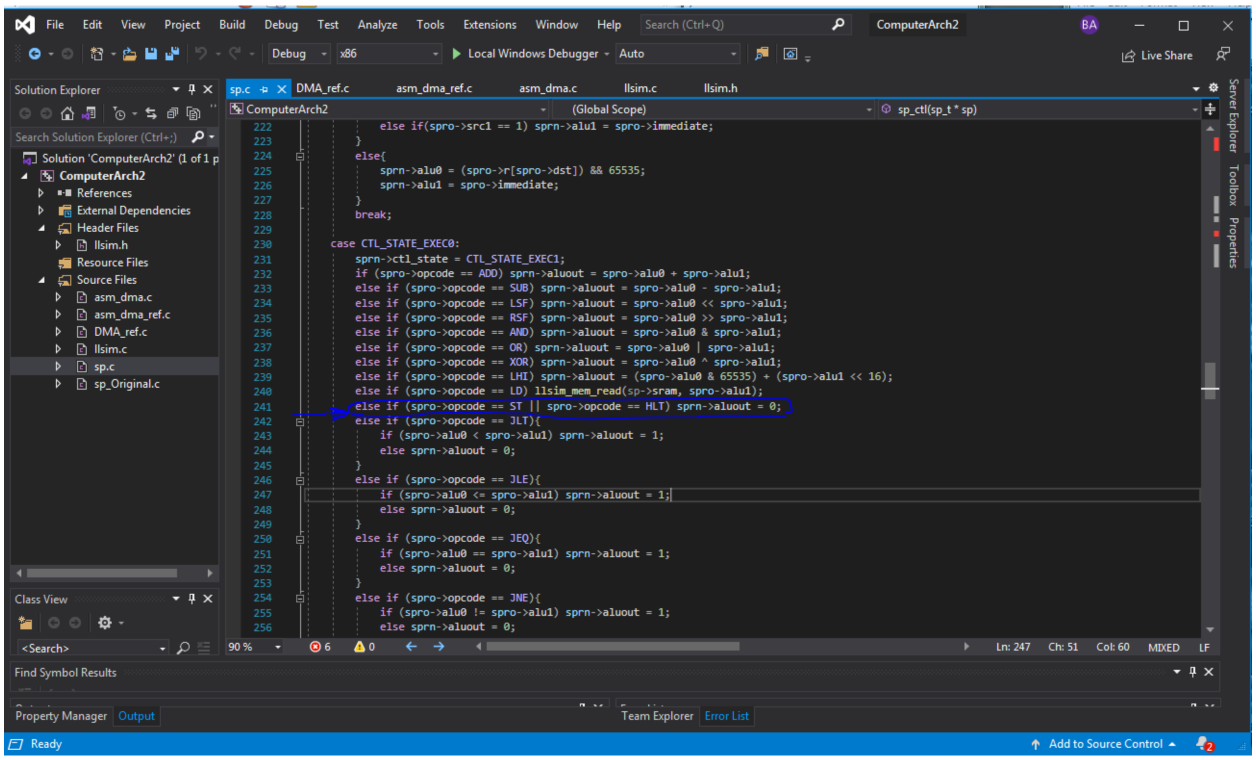
Baseline = 0
Cursor-Baseline = 16,171ns

Name	Cursor
clk	1
pc[15:0]	'h 0008
alu0[31:0]	'h 06197ECB
alu1[31:0]	'h 0000040F
opcode[4:0]	'h 09
sram_ADDR[15:0]	'h 040F
sram_DI[31:0]	'h 06197ECB
sram_WE	1
sram_EN	1
dst[2:0]	'h 0
src1[2:0]	'h 5
src0[2:0]	'h 6
inst[31:0]	'h 12350000
r2[31:0]	'h 02547029
r3[31:0]	'h 03C50EA2
r4[31:0]	'h 00000410
r5[31:0]	'h 0000040F
r6[31:0]	'h 06197ECB
r7[31:0]	'h 0000000C
aluout[31:0]	'h 06197ECB
ctl_state[2:0]	'h 6
cycle_counter[31:0]	'h 00000648

From time 16,120 nSec (when ctl_state appears to be 1 and beginning to execute another instruction), until 16,180 nSec a memory store operation (of the last examined fib sequence element) was performed by the single processor.
Cursor 'TimeA' shows when sram_ADDR was updated with the memory address value from the ALU unit's register alu1, and sram_DI was updated with the one-before-last fibonacci sequence element's value from the ALU unit's register alu0 (The was updated with the data inside register 'r6'.

As opposed to mult.bin, here, we had a mismatch between fib_verilog_trace.txt and experiment 2's fib.bin's cycle trace text file due to different handling of the 'aluout' register of the ALU unit ,when performing the operations 'ST' & 'HLT', in the lower-level simulator of experiment 2. There, we set 'aluout' register to 0 if one of this operations were performed:

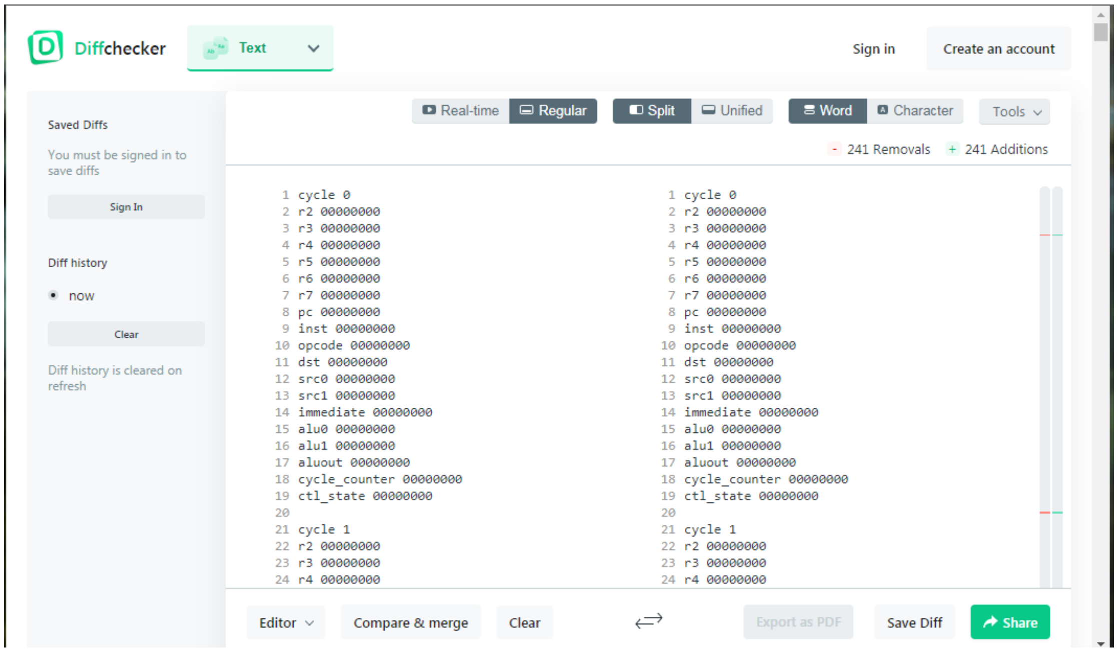
Onn Rengingad 304845951
Barak Levy 311431894
Bar Ben Ari 204034284



```c
            else if(spro->src1 == 1) sprn->alu1 = spro->immediate;
        }
        else{
            sprn->alu0 = (spro->r[spro->dst]) && 65535;
            sprn->alu1 = spro->immediate;
        }
        break;

    case CTL_STATE_EXEC0:
        sprn->ctl_state = CTL_STATE_EXEC1;
        if (spro->opcode == ADD) sprn->aluout = spro->alu0 + spro->alu1;
        else if (spro->opcode == SUB) sprn->aluout = spro->alu0 - spro->alu1;
        else if (spro->opcode == LSF) sprn->aluout = spro->alu0 << spro->alu1;
        else if (spro->opcode == RSF) sprn->aluout = spro->alu0 >> spro->alu1;
        else if (spro->opcode == AND) sprn->aluout = spro->alu0 & spro->alu1;
        else if (spro->opcode == OR) sprn->aluout = spro->alu0 | spro->alu1;
        else if (spro->opcode == XOR) sprn->aluout = spro->alu0 ^ spro->alu1;
        else if (spro->opcode == LHI) sprn->aluout = (spro->alu0 & 65535) + (spro->alu1 << 16);
        else if (spro->opcode == LD) llsim_mem_read(sp->sram, spro->alu1);
        else if (spro->opcode == ST || spro->opcode == HLT) sprn->aluout = 0;
        else if (spro->opcode == JLT){
            if (spro->alu0 < spro->alu1) sprn->aluout = 1;
            else sprn->aluout = 0;
        }
        else if (spro->opcode == JLE){
            if (spro->alu0 <= spro->alu1) sprn->aluout = 1;
            else sprn->aluout = 0;
        }
        else if (spro->opcode == JEQ){
            if (spro->alu0 == spro->alu1) sprn->aluout = 1;
            else sprn->aluout = 0;
        }
        else if (spro->opcode == JNE){
            if (spro->alu0 != spro->alu1) sprn->aluout = 1;
            else sprn->aluout = 0;
```

Also, the number of cycles that had this kind of mismatch is 241 which makes sense because there were 40 ST operations for 40 fibonacci sequence elements (6 cycles for every instruction) and another 1 HLT operation in the end (1 cycle):



- 241 Removals + 241 Additions

```
cycle 0                         cycle 0
r2 00000000                     r2 00000000
r3 00000000                     r3 00000000
r4 00000000                     r4 00000000
r5 00000000                     r5 00000000
r6 00000000                     r6 00000000
r7 00000000                     r7 00000000
pc 00000000                     pc 00000000
inst 00000000                   inst 00000000
opcode 00000000                 opcode 00000000
dst 00000000                    dst 00000000
src0 00000000                   src0 00000000
src1 00000000                   src1 00000000
immediate 00000000              immediate 00000000
alu0 00000000                   alu0 00000000
alu1 00000000                   alu1 00000000
aluout 00000000                 aluout 00000000
cycle_counter 00000000          cycle_counter 00000000
ctl_state 00000000              ctl_state 00000000

cycle 1                         cycle 1
r2 00000000                     r2 00000000
r3 00000000                     r3 00000000
r4 00000000                     r4 00000000
```

Onn Rengingad 304845951
Barak Levy 311431894
Bar Ben Ari 204034284

Question 6: DMA implementation

We created a DMA state machine capable of copying a block of memory in the background while the main operation regime continues execution. The DMA was implemented using a hardware combinational and synchronous machine defined with a set of states, to be able to process both, the main operation and memory copy operation, in parallel. The DMA state machine is composed of a different set of states compared to those defined in Experiment 2. The main thing that should be understood is that the main operation & the DMA uses a shared resource which is the SRAM. When transferring from C lower-level SP simulator implementation to an actual Hardware implementation, the parallelism between any LOAD(LD)\STORE(ST) operations executed by the Single Processor main execution, and the COPY operations executed by the DMA must be completely **compartmentalized**.
If the the SP's states & the DMA's states won't perfectly align with one another, then the SP might access the SRAM (shared resource) at the same time the DMA attempts accessing it (and vice versa), meaning, only when both, the SP & DMA, have **exactly the same number of states** AND every entity **accesses the resource at the different states**, then there would be a complete **compartmentalized** between them, and all LD, ST, COPY operations would succeed.

Thus, because experiment 2 used only 3 states (DMA_IDLE, DMA_WAIT, DMA_START), we can't use the same DMA machine, and must choose a set of 7 states (including IDLE state), which is the same number as the SP's no. of states, when some states are only used as transition between states and to "fill" the needed amount of states. (Overall, only 3 states are needed - READ, WRITE & UPDATE REGISTERS states). The **states** are as follows:

1) **DMA_STATE_IDLE** - this state informs that the DMA is silent, before being ordered to perform any memory block copy (In particular, only one COPY is performed at once), or after completing a memory block copy. When a COPY instruction is received the DMA transfers to DMA_STATE_START state.

2) **DMA__STATE_START** - this state informs that the DMA has updated all of its registers (Source address, destination address, block size etc.). This state is one of the few states we had to add in order to reach 7 states, and it transitions to DMA_STATE_READ (Take note, that during this state the SP is in DMA_STATE_FETCH0 which accesses the SRAM for a new instruction & if the DMA were to read\write to the memory, then SP & DMA would conflict).

3) **DMA_STATE_READ** - this state sets the SRAM registers (address, Enable=1, WriteEnable=0) so the SRAM will draw the data from the desired address and place it in SRAM's DO (DataOut) register. This data will later be written to a destination address predefined by the COPY operation. Also, this state sets the DMA Write bit to '1', so it is clear for the DMA to set the SRAM registers for writing in DMA_STATE_WRITE. This state transitions to DMA_STATE_PREPARE.

Onn Rengingad 304845951
Barak Levy 311431894
Bar Ben Ari 204034284

4) **DMA_STATE_PREPARE** - this state “prepares'' the DMA data register by transferring the data placed at the SRAM’s dataout register into the DMA data register. Then, this state transitions to DMA_STATE_WRITE.

5) **DMA_STATE_WRITE** - this state uses the now-prepared DMA data register & the other DMA registers (Source add., Dest. add. etc) to set the SRAM unit for writing data in the destination address by setting its registers as needed (address, DI=DMA data, Enable=1, WriteEnable=1). During this state, the DMA Write bit is set to ‘0’ to signal the DMA that a writing operation was executed and the next operation is another read (unless the entire block has finished being copied). This state transitions to DMA_STATE_UPDATE.

6) **DMA_STATE_UPDATE** - this state is the final mandatory state. In this state the DMA’s registers are advanced e.g. the source\destination addresses registers increment by ‘1’ to proceed to the following address, the block size address is decremented by ‘1’. Also, it is examined whether the entire block has finished being copied - if it had, then the DMA operation seizes & the DMA’s state becomes IDLE until a further COPY operation. If it hadn’t, then the DMA’s state becomes DMA_STATE_WAIT in order to complete the 7-states machine that promises compartmentalization between the SP & DMA when accessing the SRAM (LD\ST\COPY).

7) **DMA_STATE_WAIT** - another state that is added to secure compartmentalization between the SP & DMA when accessing the SRAM. this state transitions to DMA_STATE_START.

(Take note that the SP’s LD\ST operations access SRAM during EXEC0 & EXEC1 which occur during DMA_STATE_UPDATE & DMA_STATE_WAIT respectively, and by so the states assignment in this order assures compartmentalization between the SP & DMA)

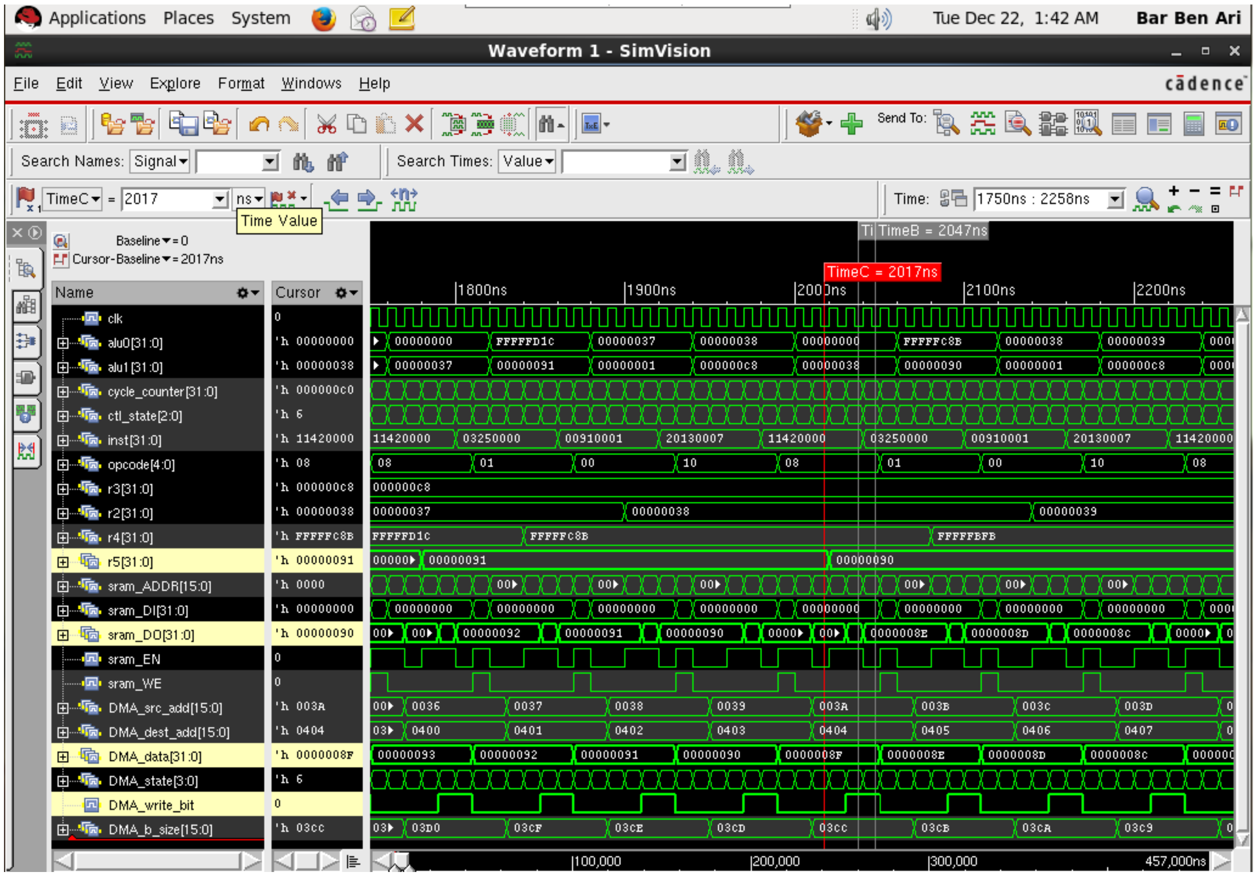
Lastly, the instructions we programmed to utilize the DMA operation are:
1) **COPY** - this instruction receives a source address (src0 reg), a destination address(dest reg) and the number of memory cells in the block we’d like to copy(src1 reg), and it initiates the memory block copy operation.
2) **POLL** - this instruction receives a destination address (dst reg) to which the DMA inserts the number of cells remaining to be copied.
Sp.c contains the DMA functionality (Generally, POLL doesn’t access the memory(SRAM), because it only samples one of the DMA’s registers(block size), so it wasn’t mentioned earlier).

Onn Rengingad 304845951
Barak Levy 311431894
Bar Ben Ari 204034284

Question 7: DMA verification

In this section, we will present a waveform diagram showing the memory access of both the assembly program (During accumulative-negative-sum operation in which register r5 is constantly loaded with data from addresses 50 - 200), and the DMA machine (Which executes the COPY operation from source address 30 to destination address 1000 over a block of size 1000 units), demonstrating structural hazard detection and resolution.

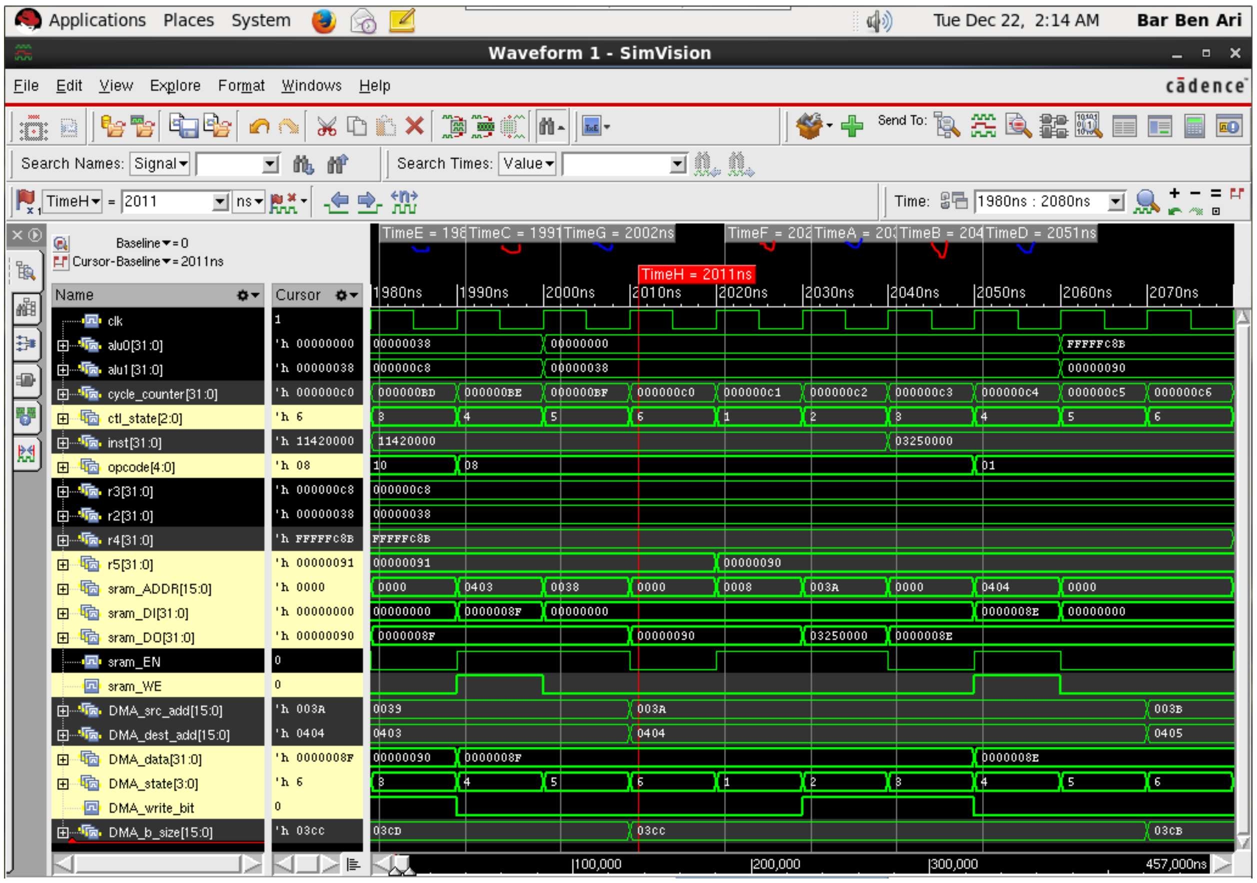
First image, is of a vast scale of events within dma.bin:



Name	Cursor
clk	0
alu0[31:0]	'h 00000000
alu1[31:0]	'h 00000038
cycle_counter[31:0]	'h 000000C0
ctl_state[2:0]	'h 6
inst[31:0]	'h 11420000
opcode[4:0]	'h 08
r3[31:0]	'h 000000C8
r2[31:0]	'h 00000038
r4[31:0]	'h FFFFFC8B
r5[31:0]	'h 00000091
sram_ADDR[15:0]	'h 0000
sram_DI[31:0]	'h 00000000
sram_DO[31:0]	'h 00000090
sram_EN	0
sram_WE	0
DMA_src_add[15:0]	'h 003A
DMA_dest_add[15:0]	'h 0404
DMA_data[31:0]	'h 0000008F
DMA_state[3:0]	'h 6
DMA_write_bit	0
DMA_b_size[15:0]	'h 03CC

And now, we will show a zoomed in version in which we will describe a DMA transfer of a single address followed by an LD operation of the SP followed by another DMA transfer of a single address:

Onn Rengingad 304845951
Barak Levy 311431894
Bar Ben Ari 204034284


Applications Places System
Tue Dec 22, 2:14 AM
Bar Ben Ari
Waveform 1 - SimVision
File Edit View Explore Format Windows Help
cadence
Search Names: Signal
Search Times: Value
TimeH = 2011 ns
Time: 1980ns : 2080ns
Baseline = 0
Cursor-Baseline = 2011ns
Name
Cursor
TimeE = 198 TimeC = 1991 TimeG = 2002ns
TimeF = 202 TimeA = 20 TimeB = 204 TimeD = 2051ns
TimeH = 2011ns
1980ns 1990ns 2000ns 2010ns 2020ns 2030ns 2040ns 2050ns 2060ns 2070ns
100,000 200,000 300,000 457,000ns

```
clk                    1
alu0[31:0]             'h 00000000
alu1[31:0]             'h 00000038
cycle_counter[31:0]    'h 000000c0
ctl_state[2:0]         'h 6
inst[31:0]             'h 11420000
opcode[4:0]            'h 08
r3[31:0]               'h 000000c8
r2[31:0]               'h 00000038
r4[31:0]               'h FFFFFC8B
r5[31:0]               'h 00000091
sram_ADDR[15:0]        'h 0000
sram_DI[31:0]          'h 00000000
sram_DO[31:0]          'h 00000090
sram_EN                0
sram_WE                0
DMA_src_add[15:0]      'h 003A
DMA_dest_add[15:0]     'h 0404
DMA_data[31:0]         'h 0000008F
DMA_state[3:0]         'h 6
DMA_write_bit          0
DMA_b_size[15:0]       'h 03cc
```

From left to right: 'Flag E' shows the DMA on state DMA_STATE_PREPARE which loads sram_DO to DMA_data (It is accomplished at the beginning of the following cycle), then 'Flag C' loads the data in DMA_data to sram_DI along with the address at DMA_dest_add into sram_ADDR(and sram_WE was set to '1'), which leads to writing '8F' from address '039' to '403'. At the following cycle ('Flag G') the instruction's opcode has already been '08' (LD), and the relevant address ('038') was loaded from alu1 to sram_ADDR, and sram_WE was set to '0' for reading. Then, at 'Flag H', sram_DO received the value '90' at SP's control state EXEC1 as expected. At the same time, the DMA addresses registers were incremented by '1'. Then, at 'Flag F' register r5 was loaded with sram_DO ('90'), SP's control state moved to FETCH0 to read from SRAM the next instruction '0325000' after completing the LD instruction successfully, and the DMA's state became START for another single-address copy. Then, at 'Flag A', sram_ADDR was loaded with the DMA's source address '03A' and the rest of the SRAM registers were set by the DMA for another read, and SP moved on to decode another instruction. Then. at 'Flag B', SRAM completed reading from the source address of the DMA the value '8E' (while sram_WE has already been '0') which is held at sram_DO. Then, at 'Flag D' sram_ADDR was loaded with the DMA's destination address, sram_WE was set to '1' and sram_DI was loaded with DMA_data to perform another read (DMA_state = WRITE)... This proceeds in a cyclic manner as desired.

Onn Rengingad 304845951
Barak Levy 311431894
Bar Ben Ari 204034284

Concerning the comparison between the Cycle trace file & SRAM image of the lower-level C simulator & the verilog implementation:
SRAM - The two memory images are identical (besides the memory addresses that remained un-initialized in the Verilog Single Processor implementation, which appears as a string of 'x's - 'xxxxxxxx', while in the lower-level C simulator, these addresses were initialized to '0' - '00000000').
Cycle trace - The trace files are identical as well besides the problem presented at section 5 (HLT differs in a cycle in the 'aluout' field).
Overall, the execution of the verilog implementation and lower-level C implementation yielded identical results.

DMA operation validation assembly code:

```
//Setting registers for COPY operation
asm_cmd(ADD, 4, 1, 0, 30);   // 0: R4 = 30 -COPY source address
asm_cmd(ADD, 5, 1, 0, 1000); // 1: R5 = 1000 -COPY destination address
asm_cmd(ADD, 6, 1, 0, 1000); // 2: R6 = 1000 -COPY operation number of cells taken

//Initiate copy regime in a different thread
asm_cmd(COPY, 5, 4, 6, 0);   // 3: COPY - DMA copy operation is initiated:
// It will copy 1000 cells from address


//Begin an Accumulative-negative-sum computation in the main thread
//Variables initiation
asm_cmd(ADD, 2, 1, 0, 50);   // 4: R2 = 50
asm_cmd(ADD, 3, 1, 0, 200);  // 5: R3 = 200
asm_cmd(ADD, 4, 0, 0, 10000);// 6: R4 = 0

//Load values and compute
asm_cmd(LD, 5, 0, 2, 0);     // 7: R5 = Mem[R2]
asm_cmd(SUB, 4, 4, 5, 0);    // 8: R4 -= R5
asm_cmd(ADD, 2, 2, 1, 1);    // 9: R2++
asm_cmd(JLT, 0, 2, 3, 7);    // 10: if R2<R3 jump to line 7


//Polling the DMA copy operation after the main Accumulative-negative-sum computation operation had finished
asm_cmd(POLL, 7, 0, 0, 0);   // 11: POLL - R[7] recieves the number of remaining cells to be copied by the DMA
asm_cmd(JNE, 7, 0, 0, 18);   // 12: if R[7]!=0 return to line 17 until COPY operation is done

//Following the termination of the DMA and Sum calculation operations, we compare the copied block of addresses
//Setting registers for comparison operation
asm_cmd(ADD, 4, 1, 0, 30);   // 13: R4 = 30 -COPY source address
asm_cmd(ADD, 5, 1, 0, 1000); // 14: R5 = 1000 -COPY destination address
asm_cmd(ADD, 6, 1, 0, 1000); // 15: R6 = 1000 -COPY operation total number of cells

asm_cmd(LD, 2, 0, 4, 0);     // 16: R2 = Mem[R4]
asm_cmd(LD, 3, 0, 5, 0);     // 17: R3 = Mem[R5]
asm_cmd(JNE, 0, 2, 3, 25);   // 18: if R2!=R3 (meanining Mem[R4]!=Mem[R5]) than DMA copy operation failed! jumping to pc=25
asm_cmd(ADD, 4, 4, 1, 1);    // 19: R4++ (Advancing source address)
asm_cmd(ADD, 5, 5, 1, 1);    // 20: R5++ (Advancing destination address)
asm_cmd(SUB, 6, 6, 1, 1);    // 21: R6-- (Decresing the number of cells remanining to validate)
asm_cmd(JNE, 0, 6, 0, 16);   // 22: if R6!=0 than there are still copied cell that need to be validated. jumping to pc=16 to proceed validating

asm_cmd(ADD, 2, 0, 1, 1);    // 23: R2=1 -> DMA Copy & Fib operation were successful
asm_cmd(JIN, 0, 1, 0, 26);   // 24: Jumping to HALT (Indirect jump to the end)
asm_cmd(ADD, 2, 0, 0, 0);    // 25: R2=0 -> DMA Copy Failed
asm_cmd(HLT, 0, 0, 0, 0);    // 26: HALT....
```