Computer Architecture

Dr. Dheya Ghazi

Assignment2: Datapath implementation

Team #5

Name	ID Number	Seat Number
Moath Yousef AL-Tahrawi	2141099	4
Omar Tareq AL-Shamasneh	2139919	69
Mohammed AL-Tamimi	2042979	58
Abdullah Nasha’at	1937045	36
Ahmed Maen Barham	2330132	1

Abstract

In this assignment we are going to implement the Datapath of the basic computer architecture, using the past modules from assignment 1, The memory unit, Registers, and the ALU.

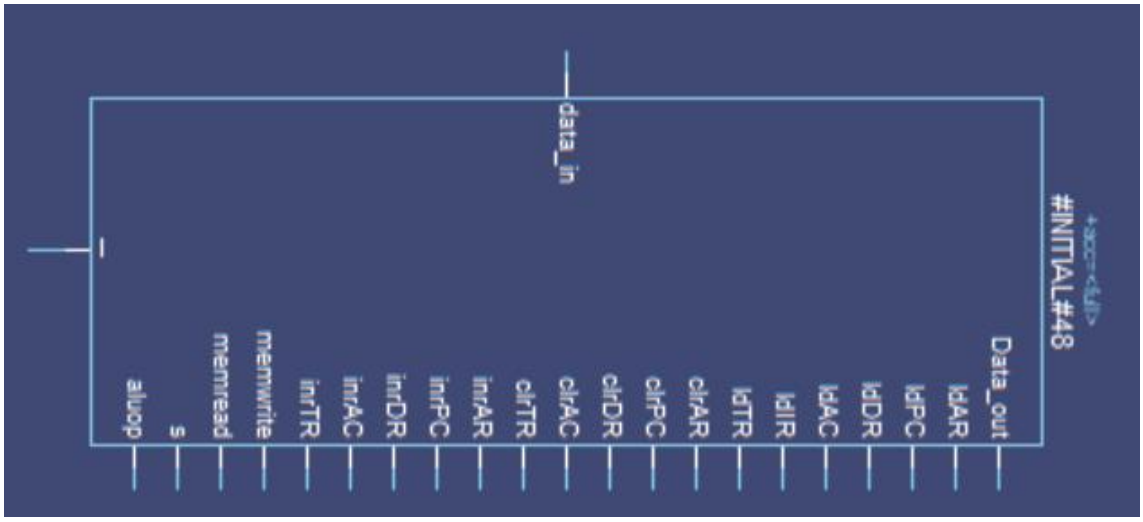
Our datapath is built on synchronous bus, which include a clock line between the modules and control lines. The bus is a shared transmission medium, which means only one device at a time can drive the bus with data.

We will show you how this datapath is going to implement assembly instructions in micro operations fashion, without using a control unit just by a testing plan, this testing plan will contain a control word which indicates the control signals that is used in this architecture, those control signals should come from the control unit to tell the system what to do, which we will write and enter to the datapath to create the cycles of each instruction (fetch, decode, execute).

This assignment will also consist of a memory initialization which is a text file that is integrated in the system using the system call ($readmemh("memory_init.hex",uut.memo1.mem);) which contains the data that should be saved in the memory of the datapath, such as the instructions machine codes and the values of the variables that is used in the ALU operations.

Datapath Implementation

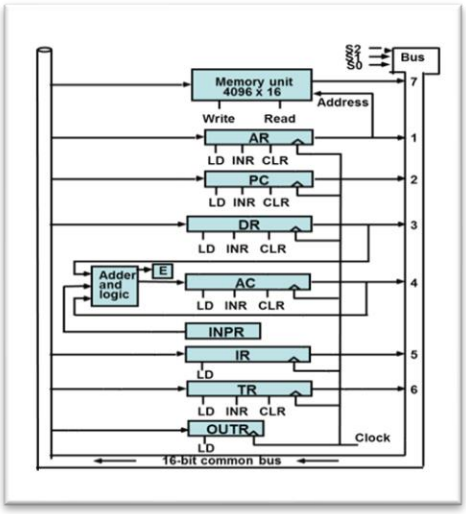
The Datapath consists of all the control signals that is used in the architecture, including load, clear, increment, bus selector lines, clk, Data_in, Data_out, memread, and memwrite, and the aluop.



Datapath BlackBox

- common bus :

In this Datapath, there is a common bus integrated from a separate module, this bus helps the Datapath modules to communicate with each other, maintaining the single bus driver priority, using the bus selector line which gives access to a single module to write on the bus at a time.



As you can see in the figure behind, the Datapath is connected to each module and teleports the data in between, for example: if the memory is the bus driver, the bus selector is set to 111 (7 in decimal) and the destination load signal is set to 1, in that way the memory writes the data on its destination.

The common bus selector values is given as follows:

Memory: 111, AR=001, PC=010, DR =011, AC= 100, IR= 101, TR= 110.

- **Note that the selector number 000 is free and can be used if you want to add another component.**
- **The data size driven in the bus = 16-bits**
- **The ALU is not connected directly to the bus (AC and DR is its inputs, and its Outputs goes to the AC)**

In that way all the system components are connected together in the Datapath module and are able to communicate as they should because of the connections they have in between.

Each module made in the previous phase has been added to the datapath module (instantiated) and connected properly to the ctrl signals and to the input and output wires, and to the common bus, each to there given selector line to ensure that the bus is only driven by one module at a time, and the bus value is not a garbage value.

- **The datapath inputs:**

Load signals (AR, PC,IR,TR,DR,AC), Clear signals(AR,PC,TR,DR,AC), increment signals AR,PC,TR,DR,AC), ALU operation signal (4 -bits), bus selector(3 – bits), clock, memread, memwrite, data_in(16-bit).

- **One output: data_out (16-bit).**
- **Data path wires:**

16-bit wires (AC,TR,IR,DR,PC, MEMORY, ALU).

12-bit wire for the AR.

1-bit wire for the extend bit.

- Modifications on modules from phase (1) :

- **The extend bit is moved into a separate module, with its own signals to clear and update its value (turned into a flip-flop instead of a variable), its instantiated into the ALU module.**

- **The extend bit module consists of all the important signals, such as the clock, the extend bit update, the New value of the Extend bit, and the E bit output.**

- **The E_bit module is instantiated in the ALU module, and in the datapath module**

- **Location 0 in the memory is reserved (blocked entrance if the AR tried to access it)**
- **we added a memread signal instead of making it with the same signal as memwrite (previously : memwrite =1 → write, memwrite =0 → memread) now each signal is independent.**

- **We added the INPR and OUTPR register modules to he registers code without integrating them to the datapath until we create the I/O operations.**

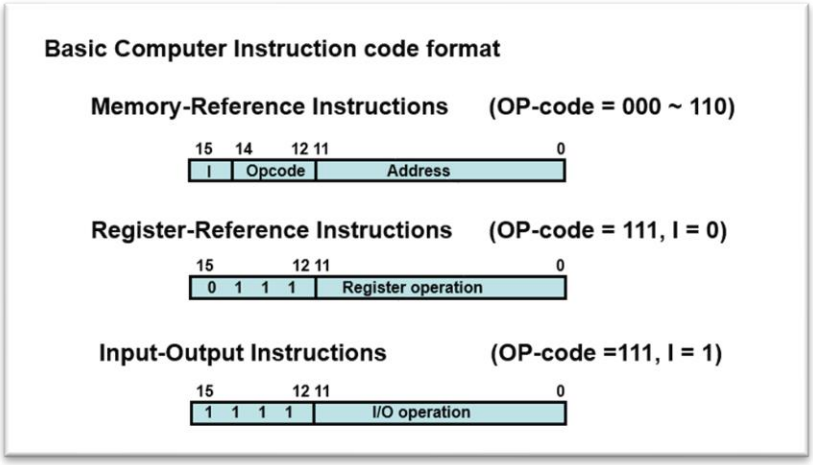
- Data path test plan:

- **We are going to test this datapath manually without adding a ctrl unit, or any of its components (decoders, sequence counter, Etc..), just by triggering the control signals manually at the time we want to trace the assembly instructions that we want to execute in the datapath.**
- **For our testbench, we manually organized a simple memory text file (as mentioned before) that contains all the data needed to run an assembly code**
- **The tests are based on micro operations that are integrated together to create the full operation of a FETCH, DECODE, EXECUTE instruction cycles.**

Datapath Test-Bench:

The Datapath Testbench consists of the datapath module instantiated in it given all the signals it needs from control unit (defined as independent signals given by the user to test the code) to create FETCH, DECODE , EXECUTE instruction cycles that runs assembly instructions for an AC based 1-address format assembly language.

The instruction format in this architecture differs depending on the type of instruction that is being executed, as shown in the figure below



- Bit no. 15 is responsible for giving the indirect bit a value to indicate the address (direct or indirect), or to indicate whether the instruction is driven by the register or an I/O in the register reference instructions.

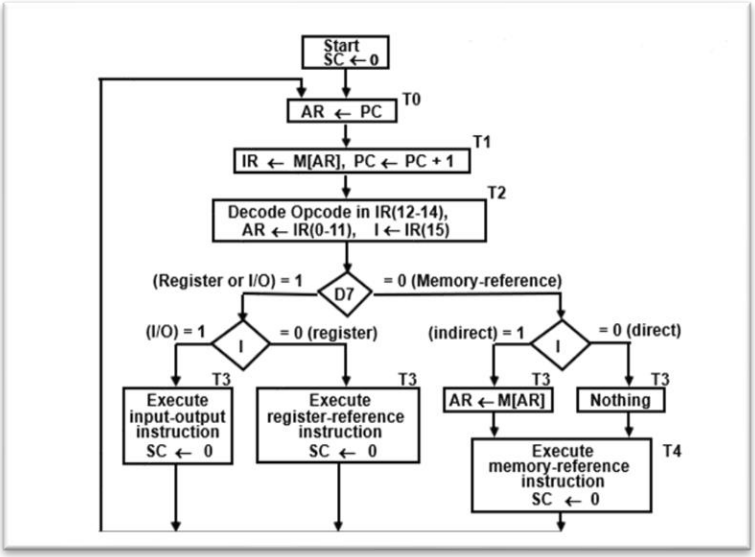
Symbol	Hex Code I = 0	Hex Code I = 1	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load AC from memory
STA	3xxx	Bxxx	Store content of AC into memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instr. if AC is positive
SNA	7008		Skip next instr. if AC is negative
SZA	7004		Skip next instr. if AC is zero
SZE	7002		Skip next instr. if E is zero
HLT	7001		Halt computer

- The Opcode indicates the instruction type as follows:

These are the assembly instructions that will be implemented in the test bench, with each instruction comes a machine code that will be saved manually in the memory unit.

- The (xxx) in the first 7 instructions refers to the location of the value that will be executed in the instructions (such as a value to load, or a location to store, or a number address, etc..)

- The cycles are indicated as shown in the following diagram:



Memory reference:

T0 and T1 : instruction Fetch
T2 and T3 : Decode
T4 and follows : Execute

The register and I/O reference:
T0 and T1 : instruction Fetch
T2: Decode
T3: Execute

- MICRO OPERATIONS TABLE:

Fetch and Decode Cycles Microoperations (for AND instruction)

Time Step	Microoperation	Signals
T0	AR ← PC	LD AR:1 S2S1S0 :010
T1	IR ← M[AR], PC ← PC + 1	S2S1S0:111, LD IR:1, INC PC:1
T2	AR ← IR[11-0], I ← IR[15]	S2S1S0:101, LD AR:1
For memory reference only: T3 Direct : Indirect :	 Nothing AR← M[AR]	 None S2S1S0:111 , ldAR=1

Direct Memory Reference Instructions - AND

Time Step	Microoperation	Signals
T4	DR ← M[AR]	S2S1S0:111, MemRead:1, LD DR:1
T5	AC ← AC ∧ DR, SC ← 0	S2S1S0:000, LD AC:1, ALU OP :0000

Direct Memory Reference Instructions - ADD

Time Step	Microoperation	Signals

T4	DR ← M[AR]	S2S1S0:111, MemRead:1, LD DR:1
T5	AC ← AC + DR, SC ← 0	S2S1S0:011, LD AC:1, ALU OP :0001

Direct Memory Reference Instructions - LDA

Time Step	Microoperation	Signals
T4	DR ← M[AR]	S2S1S0:111, MemRead:1, LD DR:1
T5	AC ← DR, SC ← 0	S2S1S0:011, LD AC:1, CLR SC:1 ALU OP :1101

Direct Memory Reference Instructions - STA

Time Step	Microoperation	Signals
T4	M[AR] ← AC	S2S1S0:100, MemWrite:1

Direct Memory Reference Instructions - BUN

Time Step	Microoperation	Signals
T4	PC ← AR, SC ← 0	S2S1S0:001, LD PC:1

Direct Memory Reference Instructions - BSA

Time Step	Microoperation	Signals
T4	M[AR] ← PC	S2S1S0:010, MemWrite:1
T5	PC ← AR + 1, SC ← 0	S2S1S0:001, LD PC:1

Direct Memory Reference Instructions - ISZ

Time Step	Microoperation	Signals
T4	DR ← M[AR]	S2S1S0: 111, LD DR:1
T5	DR ← DR + 1	S2S1S0: 011, INC DR:1
T6	M[AR] ← DR; if (DR == 0) PC ← PC + 1, SC ← 0	S2S1S0: 011, MemWrite:1, INCPC:1

Register Reference Instructions - CLA

Time Step	Microoperation	Signals
T3	AC ← 0	CLRAC =1

Register Reference Instructions - CLE

Time Step	Microoperation	Signals
T3	E ← 0	ALUOP =0111

Register Reference Instructions - CMA

Time Step	Microoperation	Signals
T3	AC ← ~AC	ALUOP =0011,LDAC=1

Register Reference Instructions - CME

Time Step	Microoperation	Signals
T3	E ← ~E	ALUOP =1000,

Register Reference Instructions - CIR

Time Step	Microoperation	Signals
T3	AC ← SHRAC, AC(15) ← E, E ← AC(	ALUOP= 0100, LDAC=1

Register Reference Instructions - CIL

Time Step	Microoperation	Signals
T3	AC ← shl AC, AC(0) ← E, E ← AC(15)	ALUOP= 0101, LDAC=1

Register Reference Instructions - INC

Time Step	Microoperation	Signals
T3	AC ← AC+1	INRAC =1

Register Reference Instructions - SPA

Time Step	Microoperation	Signals
T3	If(AC(15) =0) then (PC ← PC + 1)	INRPC=1;

Register Reference Instructions - SNA

Time Step	Microoperation	Signals
T3	If(AC(15) =1) then (PC ← PC + 1)	INRPC =1

Register Reference Instructions - SZA

Time Step	Microoperation	Signals
T3	If(AC = 0) then (PC ← PC + 1)	INRPC =1

Register Reference Instructions - SZE

Time Step	Microoperation	Signals
T3	If(E=0) then (PC ← PC + 1)	INRPC =1

Register Reference Instructions - HLT

Time Step	Microoperation	Signals
T3	If(E=0) then (PC ← PC + 1)	ALL SIGNALS CLEAR =1;

- Memory Organization:

- As mentioned before, our memory text file is organized and splitted into virtual segments to make the memory space easy to understand
- locations 300 → 325 contains the instructions machine codes that we want to trace in the test bench in order .
- locations 500 → 514 contains the values of the numbers that we want to operate on them
- locations 600 → 605 are the indirect addresses that contain the values of the numbers to operate
- locations 330 and 332 contain values to check the branch functionality only
- else any other location in the memory contain 0000 values, you may find some locations with garbage values but it won't effect the testing operation
- this memory is called with a system call(as mentioned in the intro) in the datapath module, the values will be saved in the memory array.
- you may find this memory text file with the code file to check.

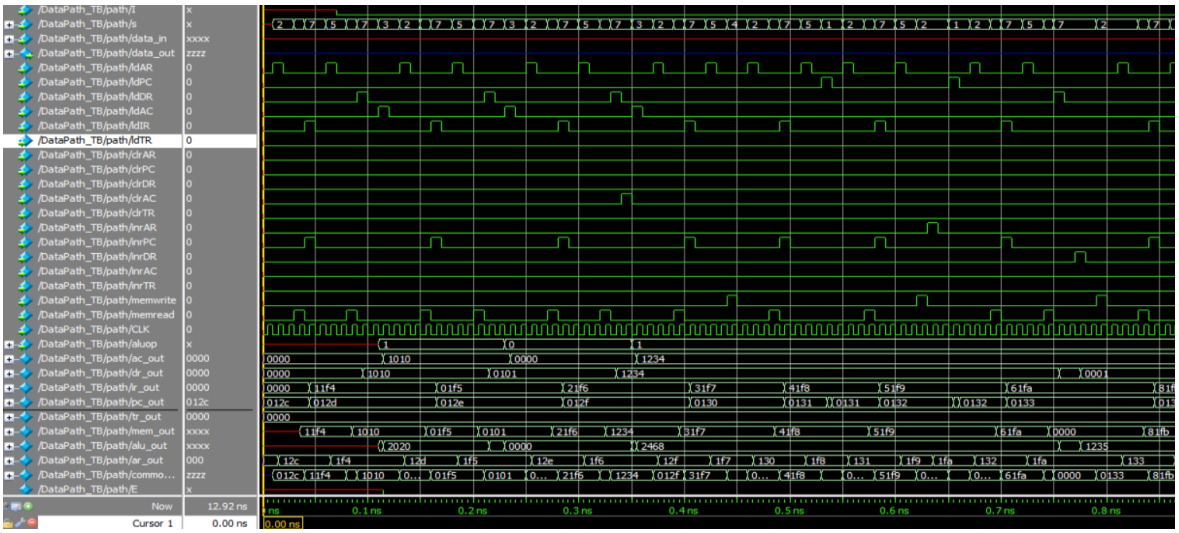
- Testing results :

- The code segment consists of 26 machine codes, each machine code indicates an instruction, and an address location to get the data into the data register, the testbench takes this machine code to the IR register where it is split into parts to understand the characteristics of the assembly instruction (memory direct, indirect, register reference, or I/O reference).
- Using the display system call, we displayed all the values needed to trace the outputs of the datapath to check if the system is working properly as it should
- The result will be consisting of the output display screen and the signals wave form for each instruction executed

First group test:

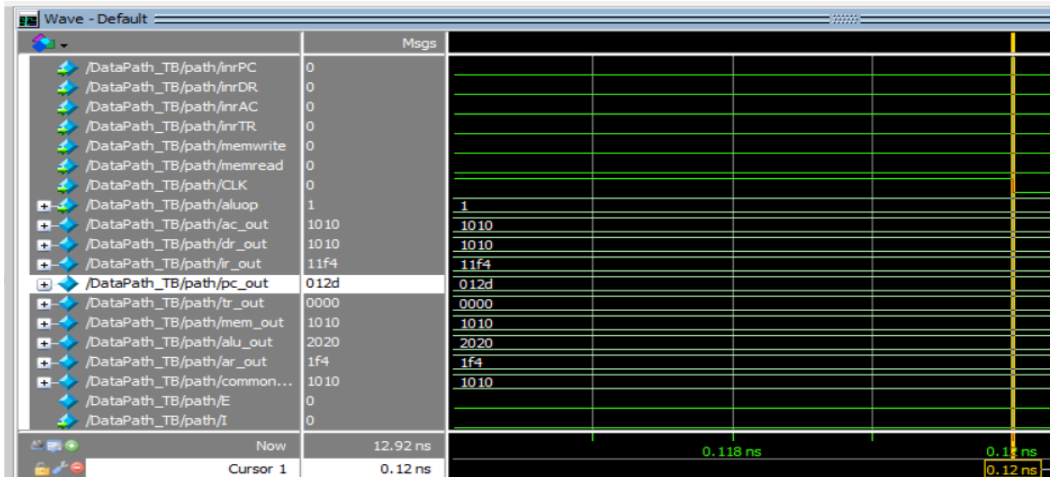
Direct Memory Reference intstructions:

This wave form shows all the signals sequence from the first to last memory reference inst.



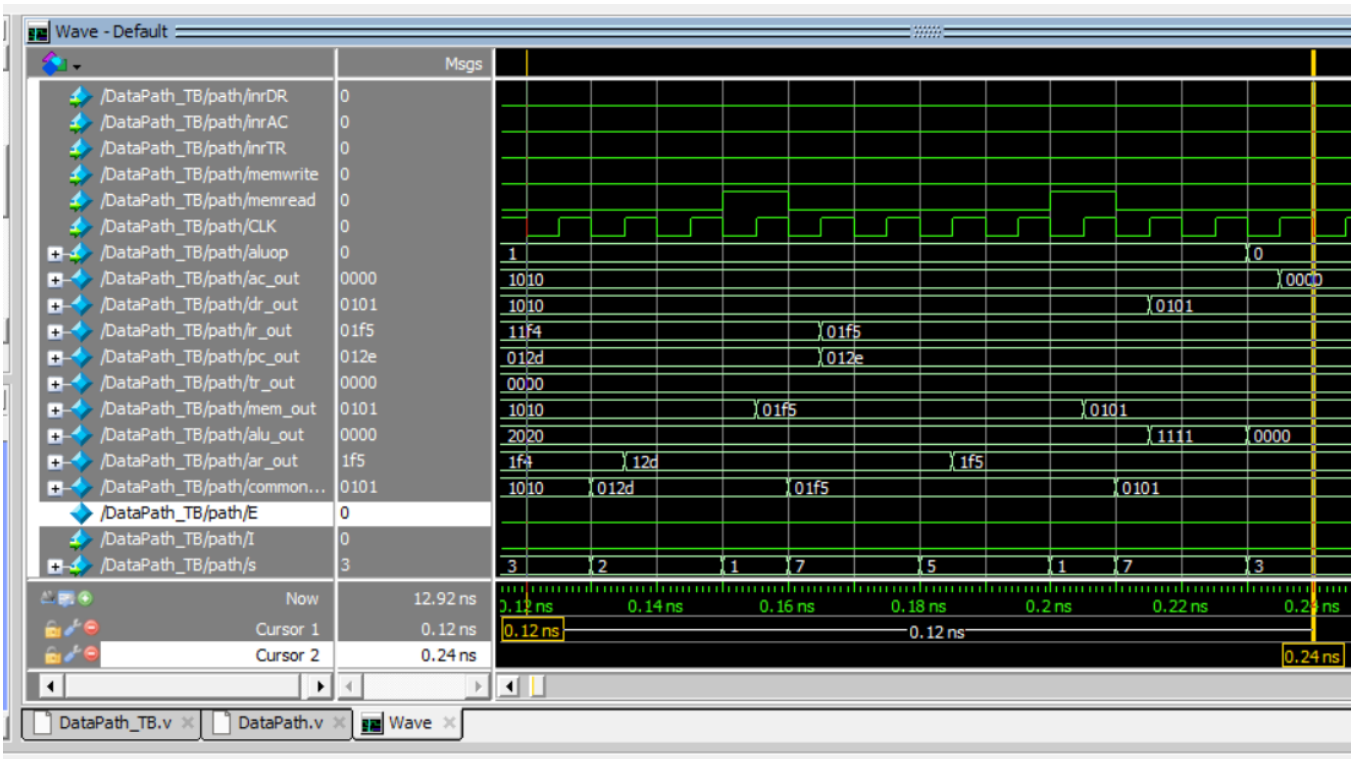
ADD instruction

```
VSIM 10> run -all
# Time: 0 | AR: 000 | PC: 012c | DR: 0000 | AC: 0000 | IR: 0000 | I: x
#
# ----------------------------Direct Memory Operations-----------------------------
#
# ADD INSTRUCTION
#
# Memory initialized from memory_init.txt.
# Fetching instruction...
#  T0: (AR <- PC) AR = 12c, PC = 012c
#  T1: (IR <- M[AR], PC <- PC + 1) IR = 11f4, PC = 012d
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 1f4
#  T3: (Direct : Nothing) I = 0
# Executing ADD instruction...
#  T4D1: (DR <- M[AR]) DR = 1010
#  T5D1: (AC <- AC + DR, E <- Cout) AC = 1010, E = 0
# Time: 120 | AR: 1f4 | PC: 012d | DR: 1010 | AC: 1010 | IR: 11f4 | I: 0
```



AND Instruction

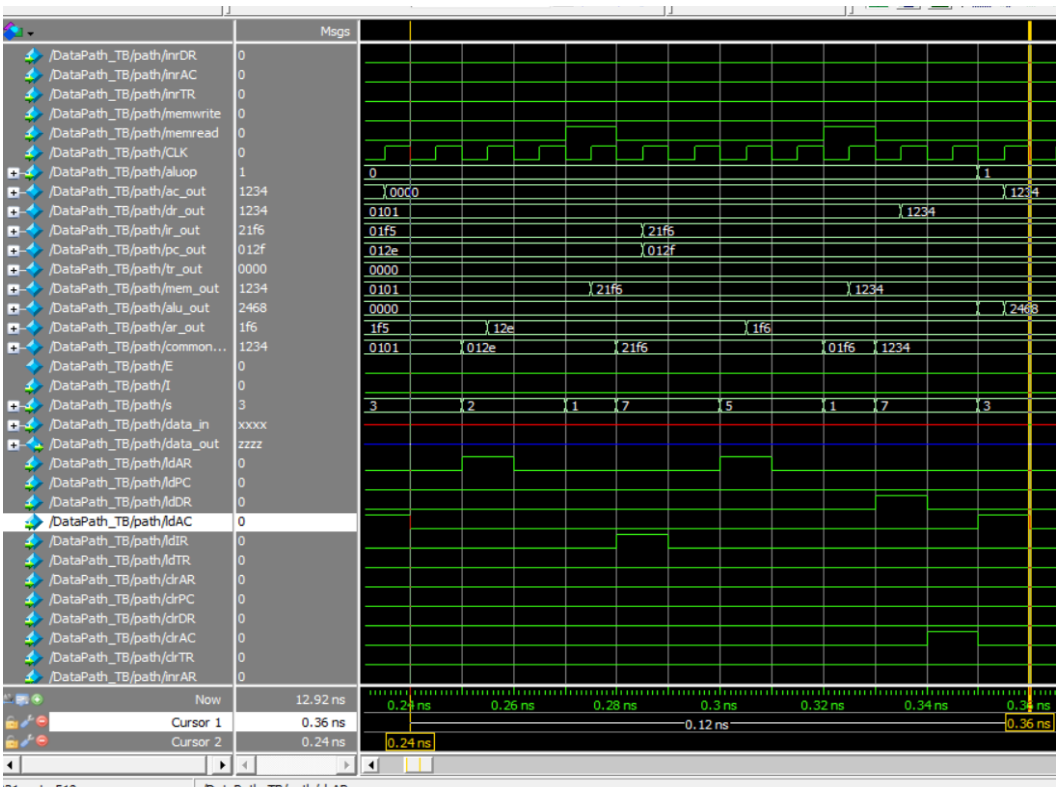
```
#
# AND INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 12d, PC = 012d
#  T1: (IR <- M[AR], PC <- PC + 1) IR = 01f5, PC = 012e
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 1f5
#  T3: (Direct : Nothing) I = 0
# Executing AND instruction...
#  T4D0: (DR <- M[AR]) DR = 0101
#  T5D0: (AC <- AC & DR) AC = 0000
# Time: 240 | AR: 1f5 | PC: 012e | DR: 0101 | AC: 0000 | IR: 01f5 | I: 0
```



LDA Instruction

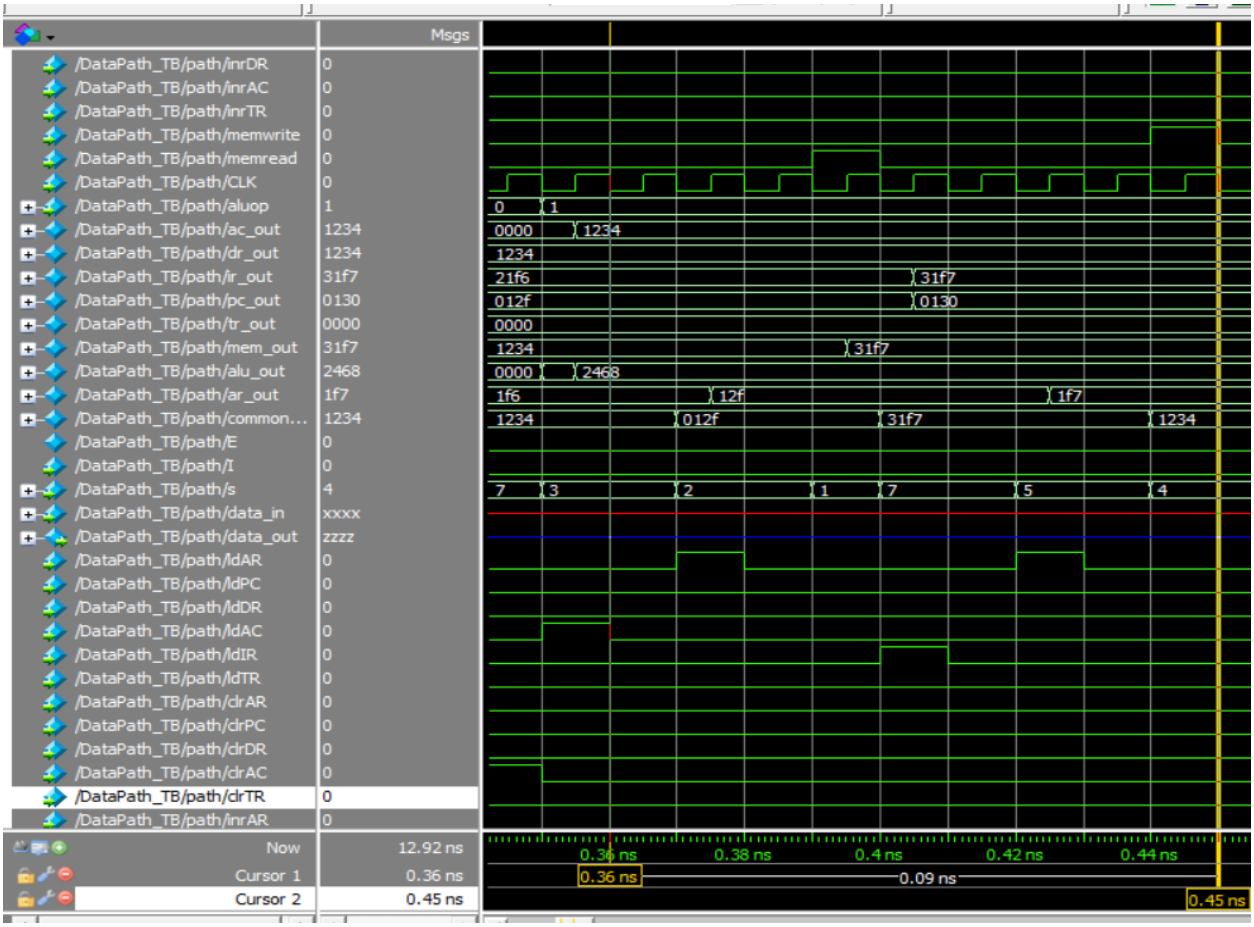
Transcript

```
# LDA INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 12e, PC = 012e
#  T1: (IR <- M[AR], PC <- PC + 1) IR = 21f6, PC = 012f
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 1f6
#  T3: (Direct : Nothing) I = 0
# Executing LDA instruction...
#  T4D2: (DR <- M[AR]) DR = 1234
#  T5D2: (AC <- DR) AC = 1234
# Time: 360 | AR: 1f6 | PC: 012f | DR: 1234 | AC: 1234 | IR: 21f6 | I: 0
```



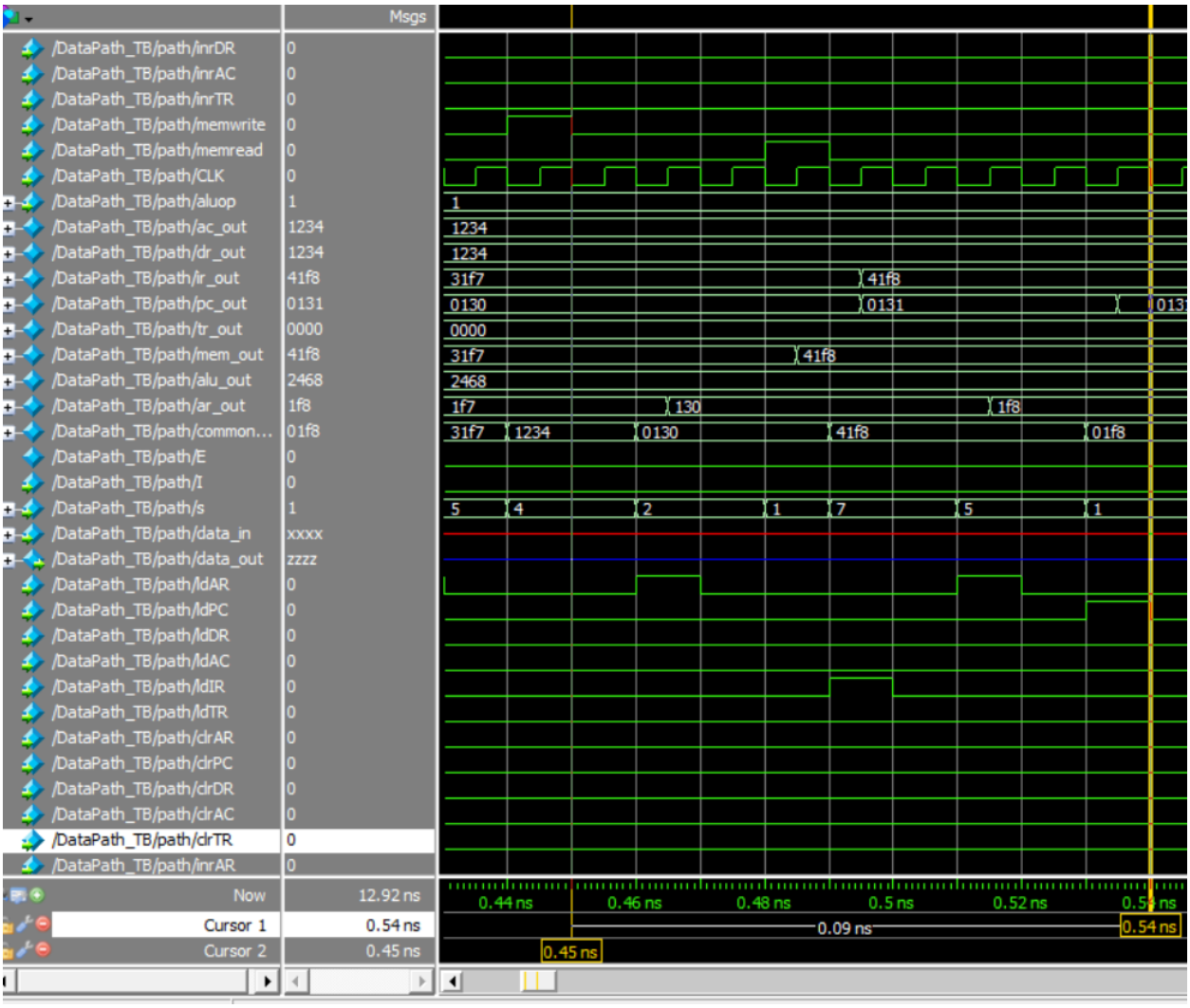
STA Instruction

```
#
# STA INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 12f, PC = 012f
#  T1: (IR <- M[AR], PC <- PC + 1) IR = 31f7, PC = 0130
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 1f7
#  T3: (Direct : Nothing) I = 0
# Executing STA instruction...
#  T4D3: (M[AR] <- AC) M[AR] = 1234
# Time: 450 | AR: 1f7 | PC: 0130 | DR: 1234 | AC: 1234 | IR: 31f7 | I: 0
#
```



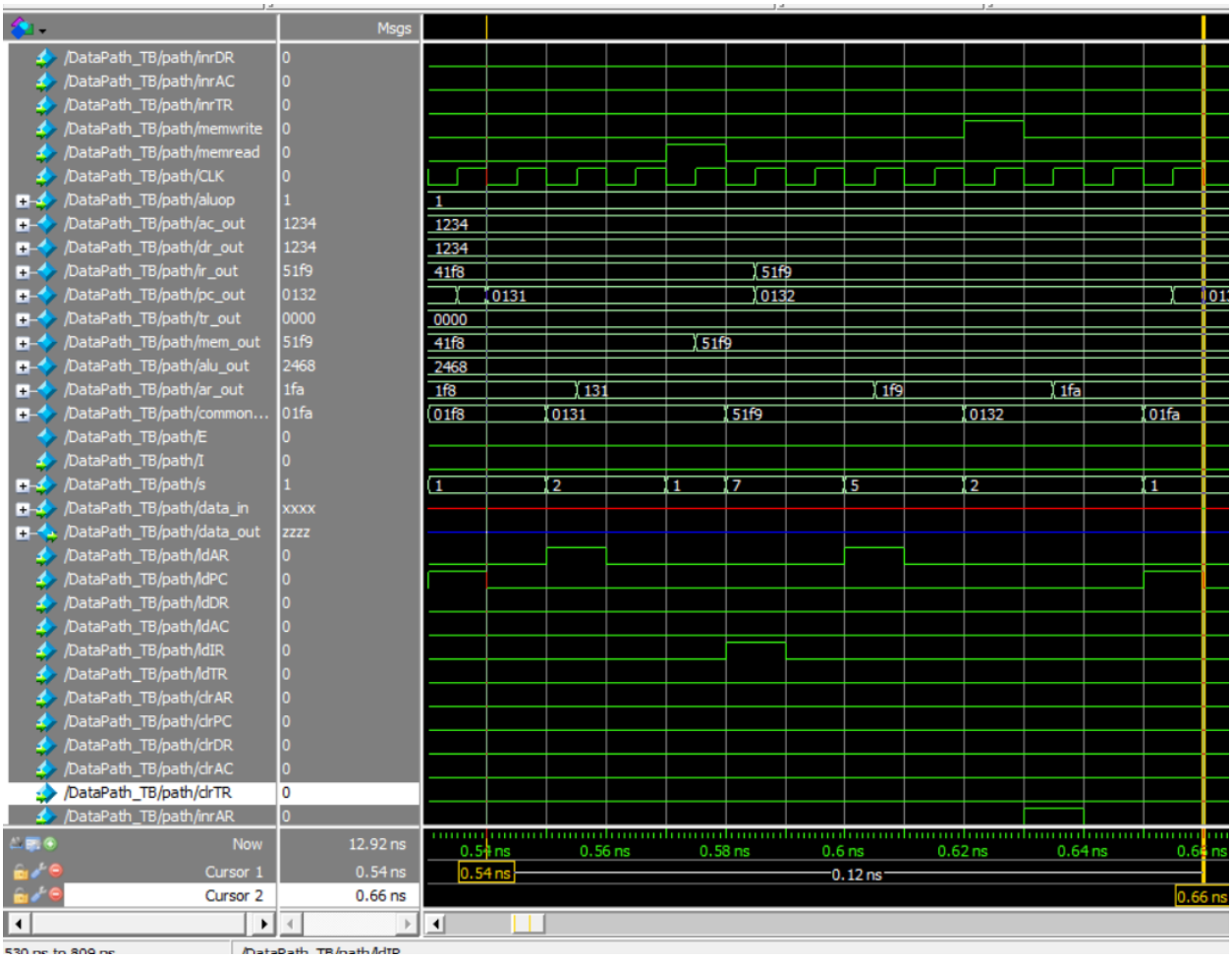
BUN instruction

```
#
# BUN INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 130, PC = 0130
#  T1: (IR <- M[AR], PC <- PC + 1) IR = 41f8, PC = 0131
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 1f8
#  T3: (Direct : Nothing) I = 0
# Executing BUN instruction...
#  T4D4: (PC <- AR) PC = 01f8
# Time: 540 | AR: 1f8 | PC: 01f8 | DR: 1234 | AC: 1234 | IR: 41f8 | I: 0
```



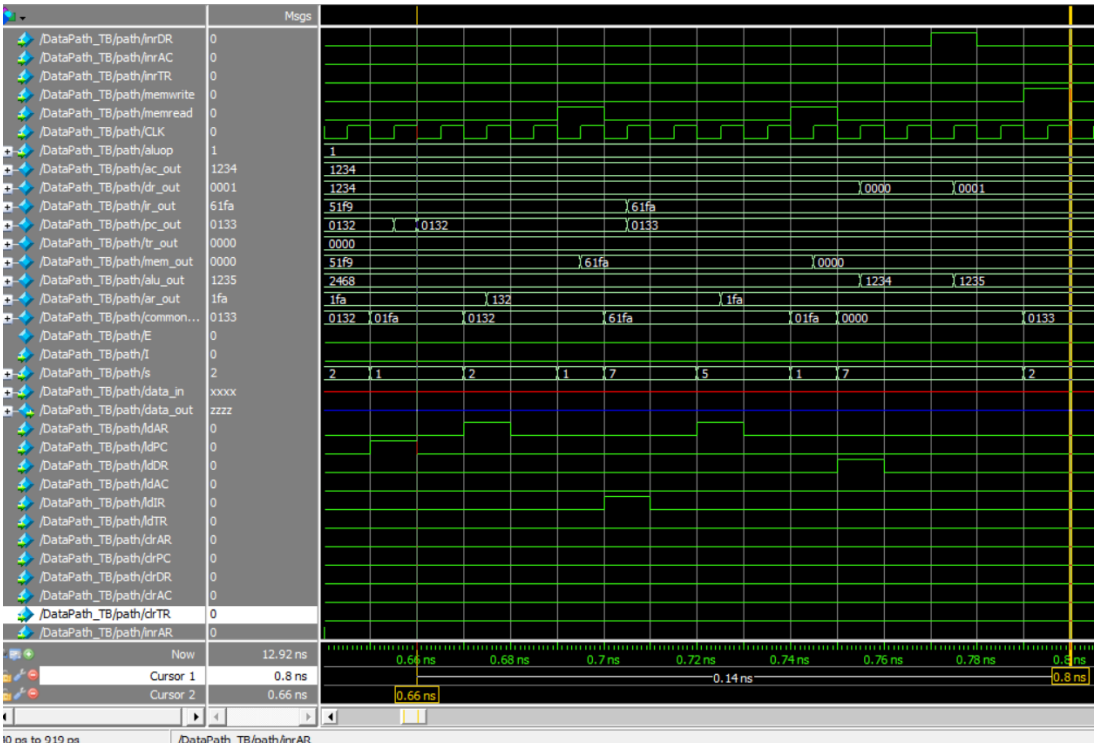
BSA Instruction

```
#
# BSA INSTRUCTION
# Fetching instruction...
#   T0: (AR <- PC) AR = 131, PC = 0131
#   T1: (IR <- M[AR], PC <- PC + 1) IR = 51f9, PC = 0132
# Decoding instruction...
#   T2: (AR <- IR[0-11], I <- IR[15]) AR = 1f9
#   T3: (Direct : Nothing) I = 0
# Executing BSA instruction...
#   T4D5: (M[AR] <- PC, AR <- AR + 1) M[444] = 0132, AR = 1fa
#   T5D5: (PC <- AR) PC = 01fa
# Time: 660 | AR: 1fa | PC: 01fa | DR: 1234 | AC: 1234 | IR: 51f9 | I: 0
#
```



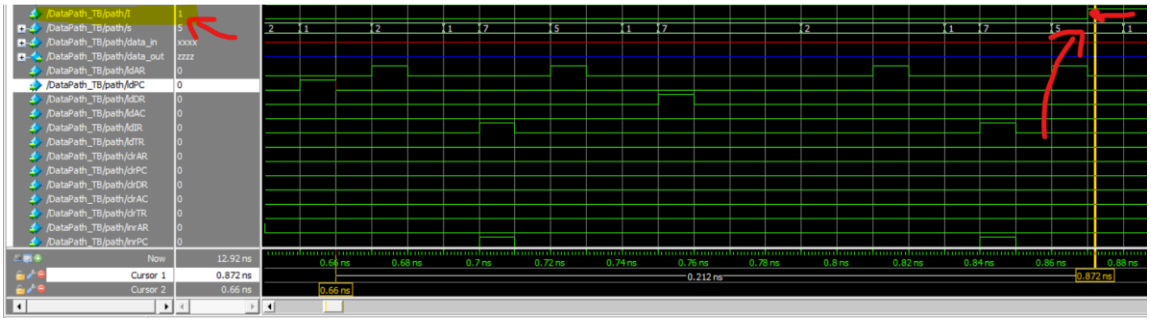
ISZ Instruction

```
#
# ISZ INSTRUCTION
# Fetching instruction...
#   T0: (AR <- PC) AR = 132, PC = 0132
#   T1: (IR <- M[AR], PC <- PC + 1) IR = 61fa, PC = 0133
# Decoding instruction...
#   T2: (AR <- IR[0-11], I <- IR[15]) AR = 1fa
#   T3: (Direct : Nothing) I = 0
# Executing ISZ instruction...
#   T4D6: (DR <- M[AR]) DR = 0000
#   T5D6: (DR <- DR + 1) DR = 0001
#   T6D6: (M[AR] <- DR, [If (DR == 0) : PC <- PC + 1]) M[AR] = 0133, PC = 0133
# Time: 800 | AR: 1fa | PC: 0133 | DR: 0001 | AC: 1234 | IR: 61fa | I: 0
#
```



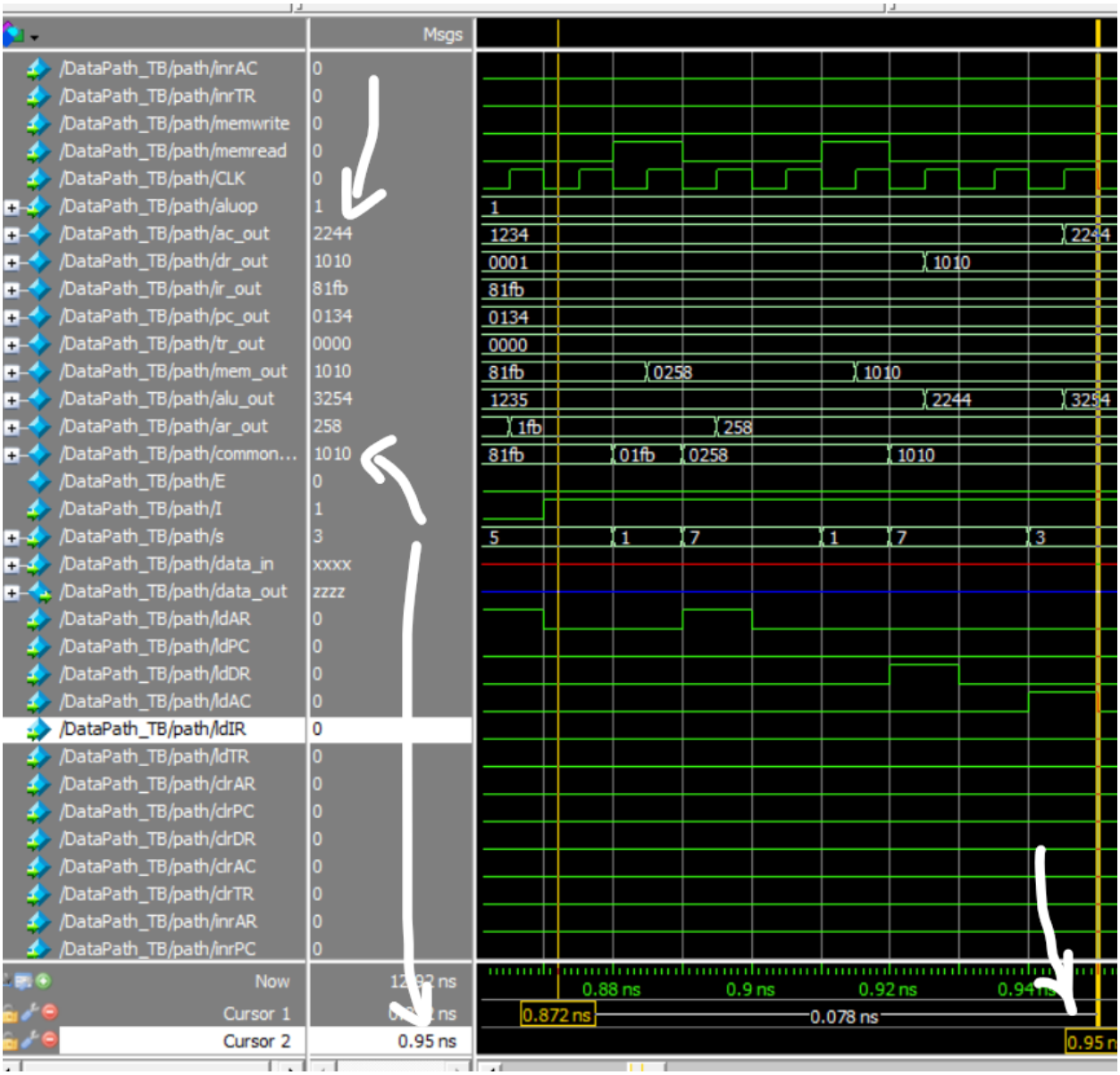
Second group test: Indirect Memory reference

The picture below shows the I bit being set for the next group of instructions



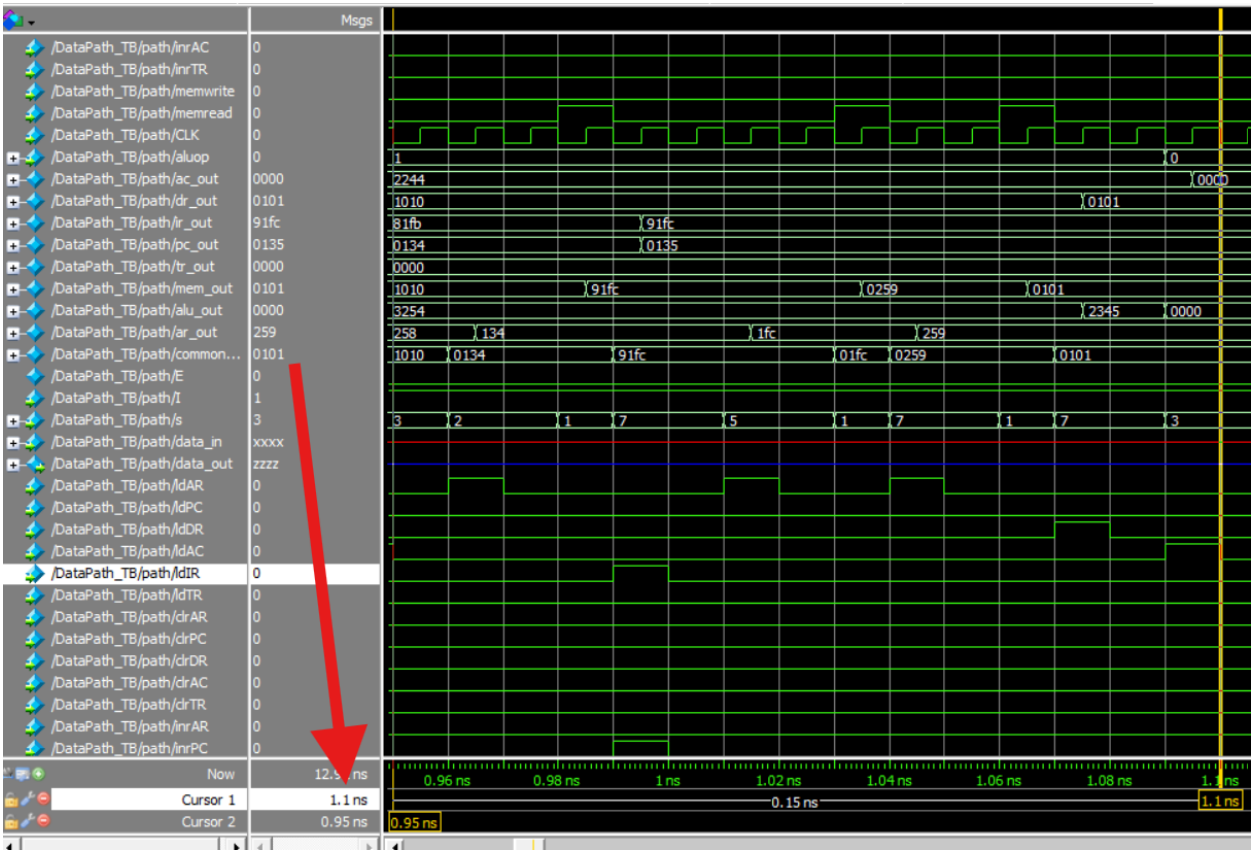
ADD Instruction

```
#
# ----------------------------Indirect Memory Operations----------------------------
# ADD INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 133, PC = 0133
#  T1: (IR <- M[AR], PC <- PC + 1) IR = 81fb, PC = 0134
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 1fb
#  T3 (Indirect): AR <- M[AR], AR = 258
# Executing ADD instruction...
#  T4D1: (DR <- M[AR]) DR = 1010
#  T5D1: (AC <- AC + DR, E <- Cout) AC = 2244, E = 0
# Time: 950 | AR: 258 | PC: 0134 | DR: 1010 | AC: 2244 | IR: 81fb | I: 1
```



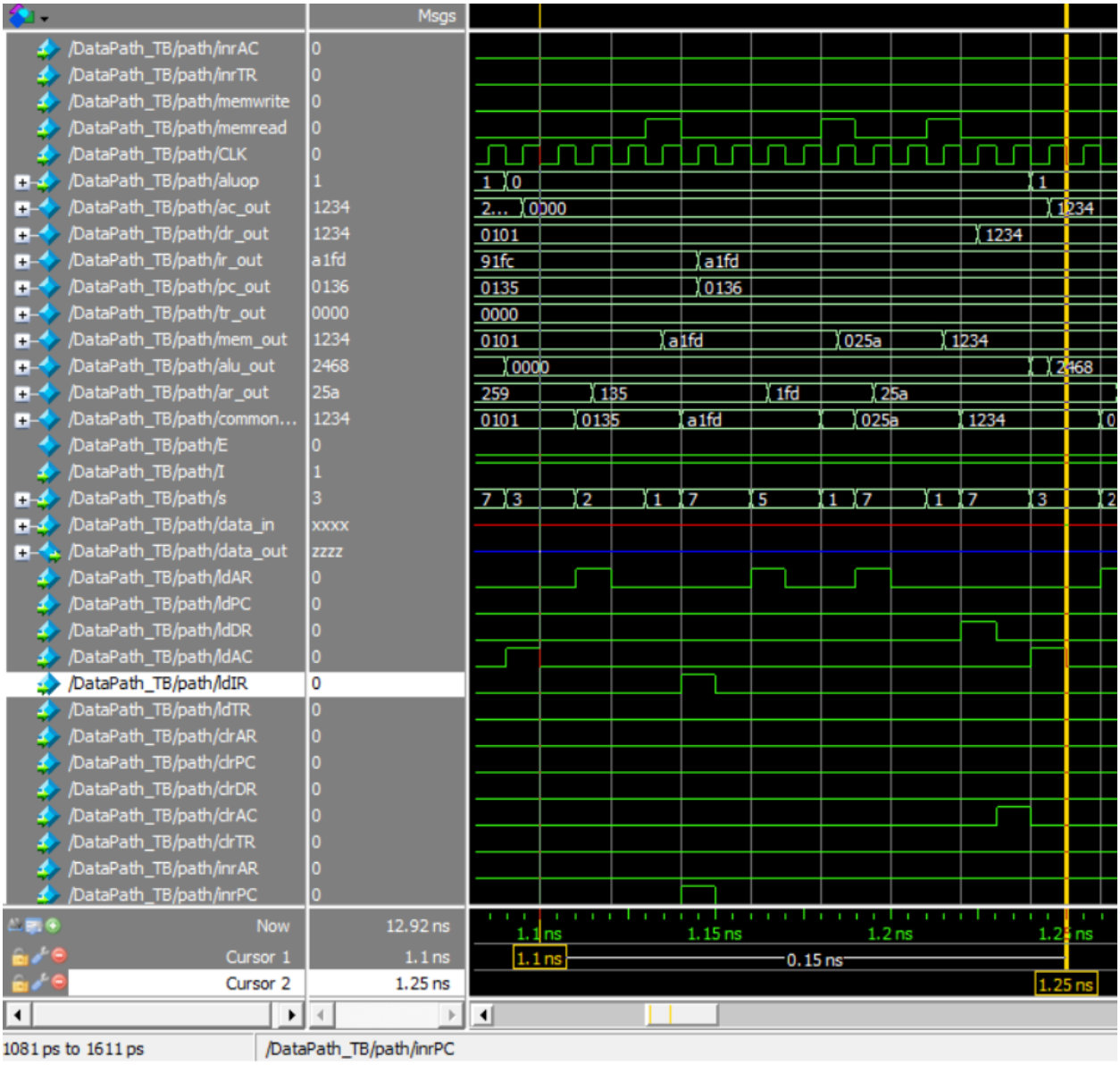
AND Instruction

```
# AND INSTRUCTION
# Fetching instruction...
#   T0: (AR <- PC) AR = 134, PC = 0134
#   T1: (IR <- M[AR], PC <- PC + 1) IR = 91fc, PC = 0135
# Decoding instruction...
#   T2: (AR <- IR[0-11], I <- IR[15]) AR = 1fc
#   T3 (Indirect): AR <- M[AR], AR = 259
# Executing AND instruction...
#   T4D0: (DR <- M[AR]) DR = 0101
#   T5D0: (AC <- AC & DR) AC = 0000
# Time: 1100 | AR: 259 | PC: 0135 | DR: 0101 | AC: 0000 | IR: 91fc | I: 1
#
```



LDA instruction

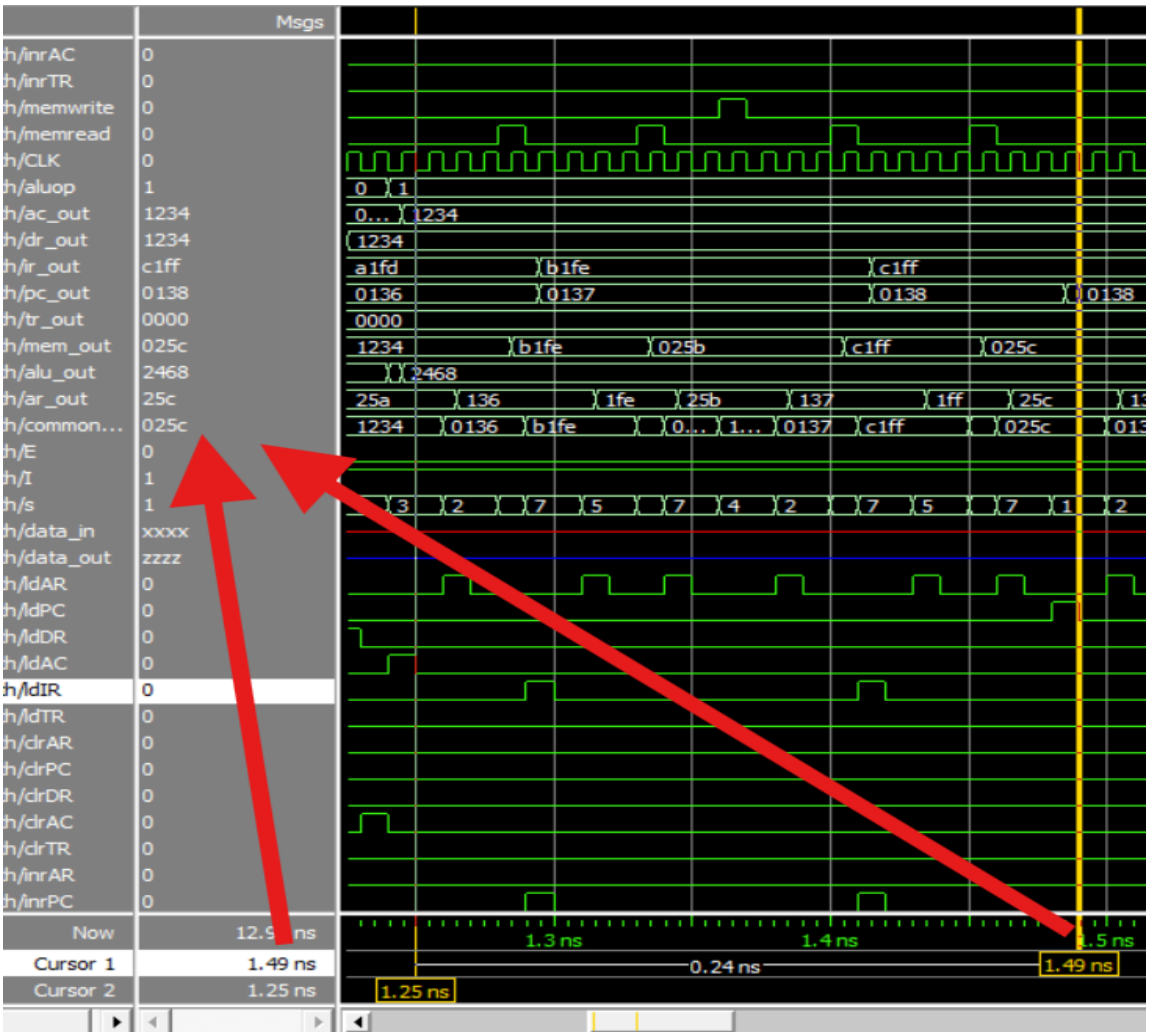
```
#
# LDA INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 135, PC = 0135
#  T1: (IR <- M[AR], PC <- PC + 1) IR = a1fd, PC = 0136
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 1fd
#  T3 (Indirect): AR <- M[AR], AR = 25a
# Executing LDA instruction...
#  T4D2: (DR <- M[AR]) DR = 1234
#  T5D2: (AC <- DR) AC = 1234
# Time: 1250 | AR: 25a | PC: 0136 | DR: 1234 | AC: 1234 | IR: a1fd | I: 1
#
```



STA AND BUN INSTRUCTIONS

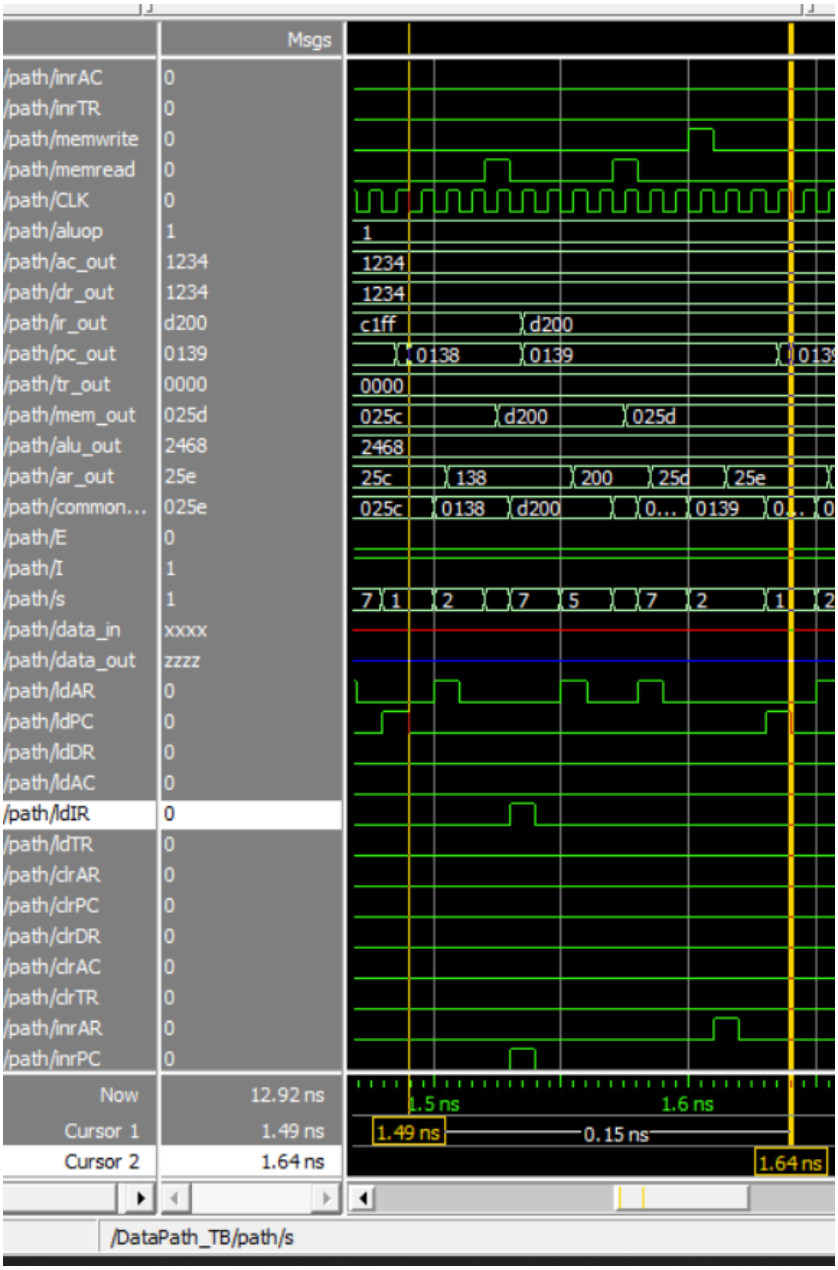
```
# STA INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 136, PC = 0136
#  T1: (IR <- M[AR], PC <- PC + 1) IR = b1fe, PC = 0137
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 1fe
#  T3 (Indirect): AR <- M[AR], AR = 25b
# Executing STA instruction...
#  T4D3: (M[AR] <- AC) M[AR] = 1234
# Time: 1370 | AR: 25b | PC: 0137 | DR: 1234 | AC: 1234 | IR: b1fe | I: 1
```

```
# BUN INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 137, PC = 0137
#  T1: (IR <- M[AR], PC <- PC + 1) IR = c1ff, PC = 0138
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 1ff
# T3 (Indirect): AR <- M[AR], AR = 25c
# Executing BUN instruction...
#  T4D4: (PC <- AR) PC = 025c
# Time: 1490 | AR: 25c | PC: 025c | DR: 1234 | AC: 1234 | IR: c1ff | I: 1
```



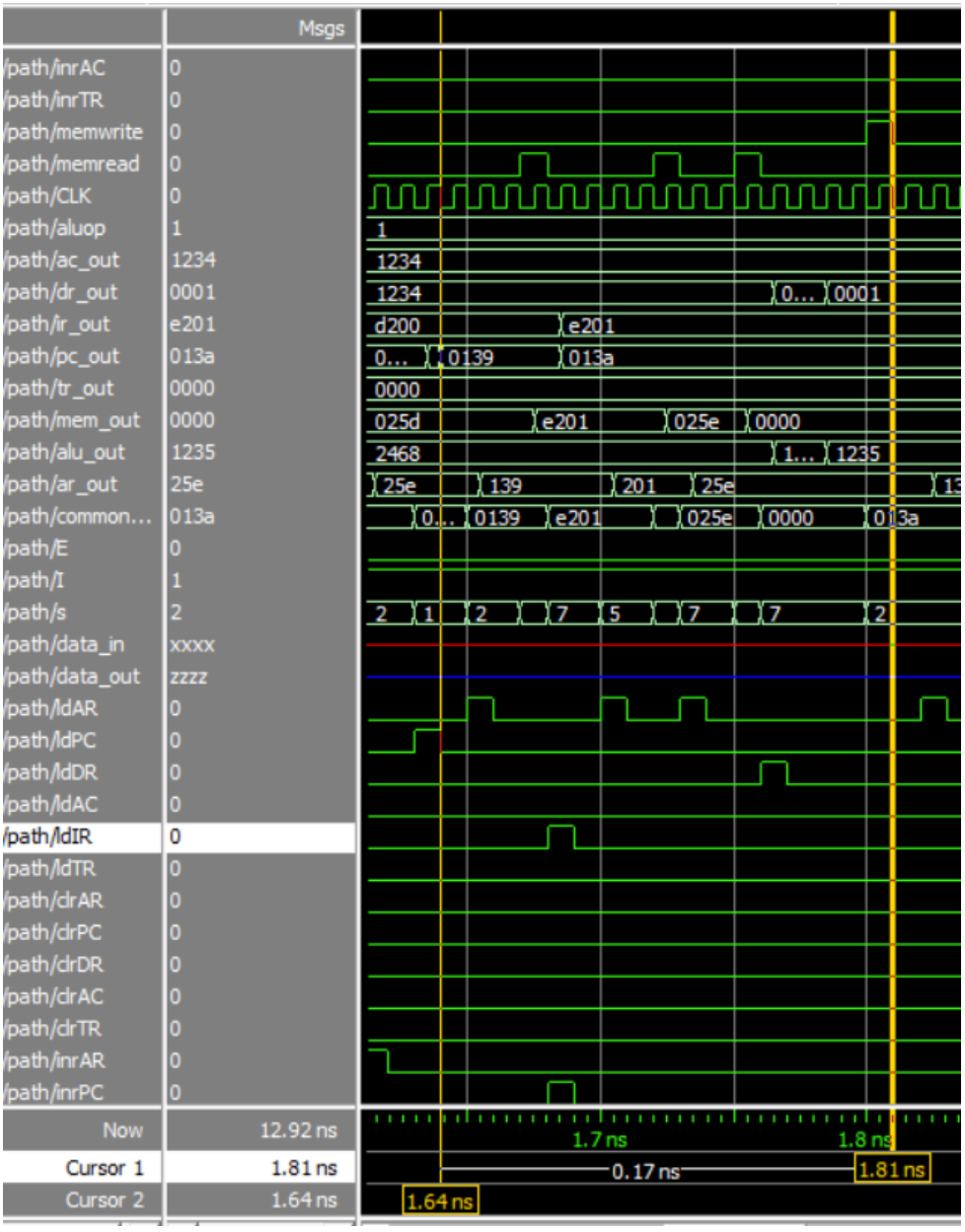
BSA Instruction

```
# BSA INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 138, PC = 0138
#  T1: (IR <- M[AR], PC <- PC + 1) IR = d200, PC = 0139
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 200
#  T3 (Indirect): AR <- M[AR], AR = 25d
# Executing BSA instruction...
#  T4D5: (M[AR] <- PC, AR <- AR + 1) M[444] = 0139, AR = 25e
#  T5D5: (PC <- AR) PC = 025e
# Time: 1640 | AR: 25e | PC: 025e | DR: 1234 | AC: 1234 | IR: d200 | I: 1
```



ISZ Instruction

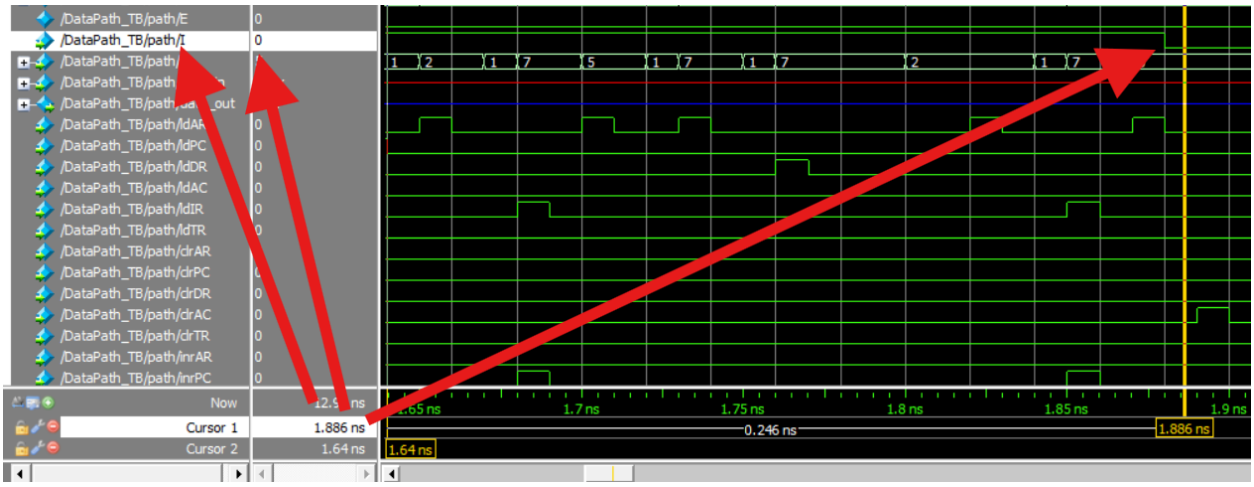
```
# ISZ INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 139, PC = 0139
#  T1: (IR <- M[AR], PC <- PC + 1) IR = e201, PC = 013a
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 201
#  T3 (Indirect): AR <- M[AR], AR = 25e
# Executing ISZ instruction...
#  T4D6: (DR <- M[AR]) DR = 0000
#  T5D6: (DR <- DR + 1) DR = 0001
#  T6D6: (M[AR] <- DR, [If (DR == 0) : PC <- PC + 1]) M[AR] = 013a, PC = 013a
# Time: 1810 | AR: 25e | PC: 013a | DR: 0001 | AC: 1234 | IR: e201 | I: 1
```



Third group of instructions

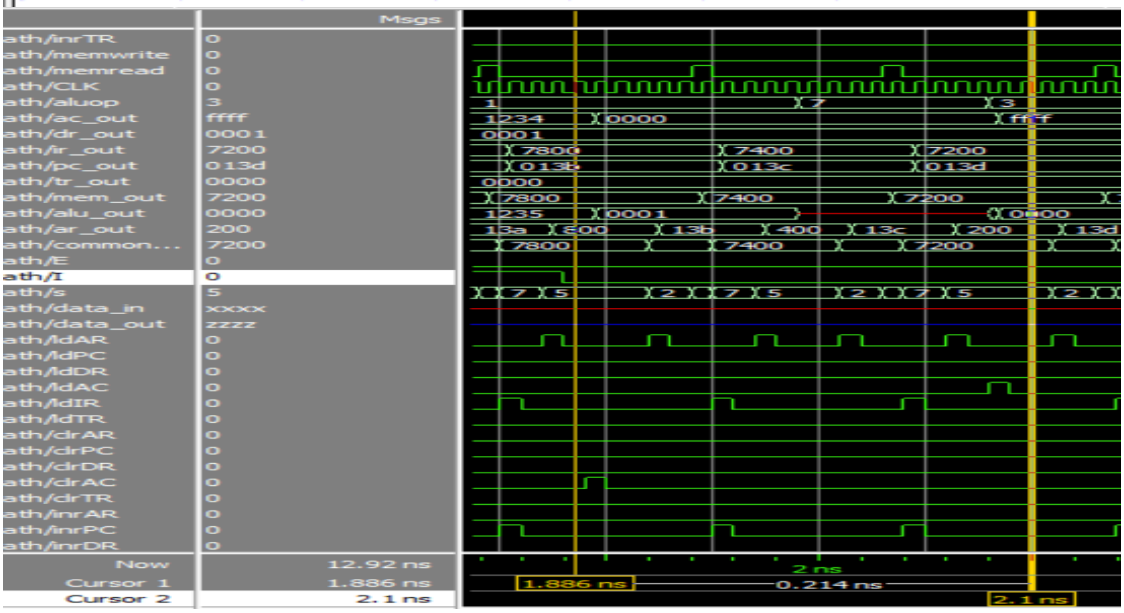
Register reference Instructions

The I Bit change :



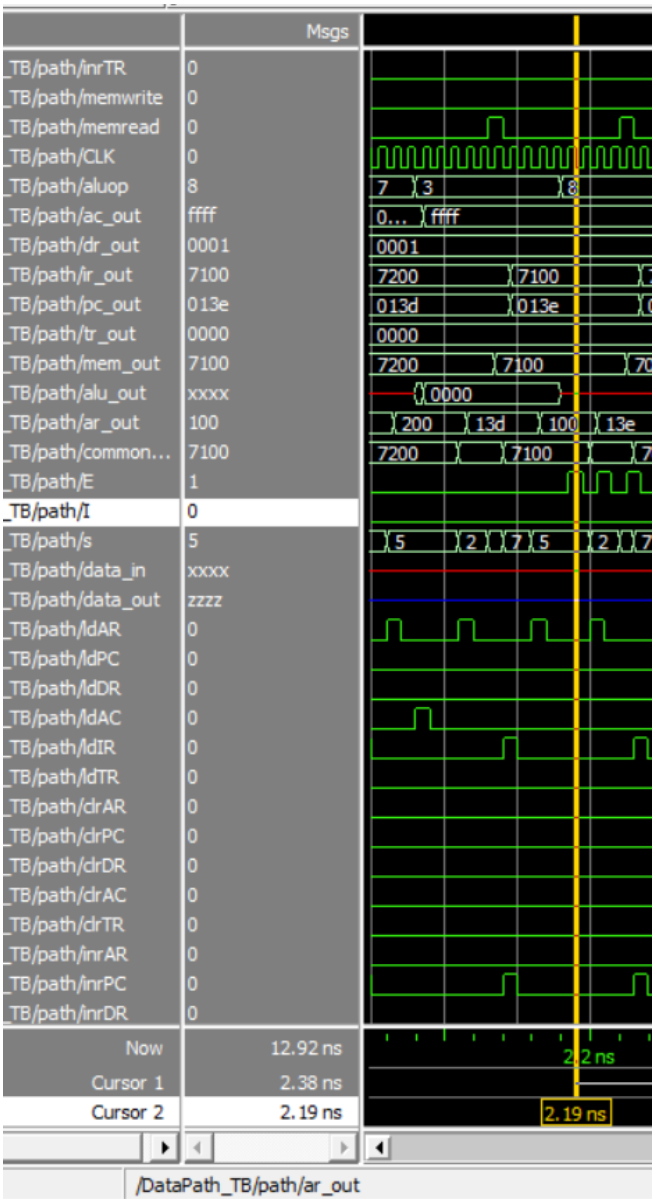
CMA Instruction

```
# CMA INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 13c, PC = 013c
#  T1: (IR <- M[AR], PC <- PC + 1) IR = 7200, PC = 013d
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 200
#  T3: (Register) I = 0
# Executing CMA instruction...
#  T4D7B9: (AC <- ~AC) AC = ffff
# Time: 2100 | AR: 200 | PC: 013d | DR: 0001 | AC: ffff | IR: 7200 | I: 0
```



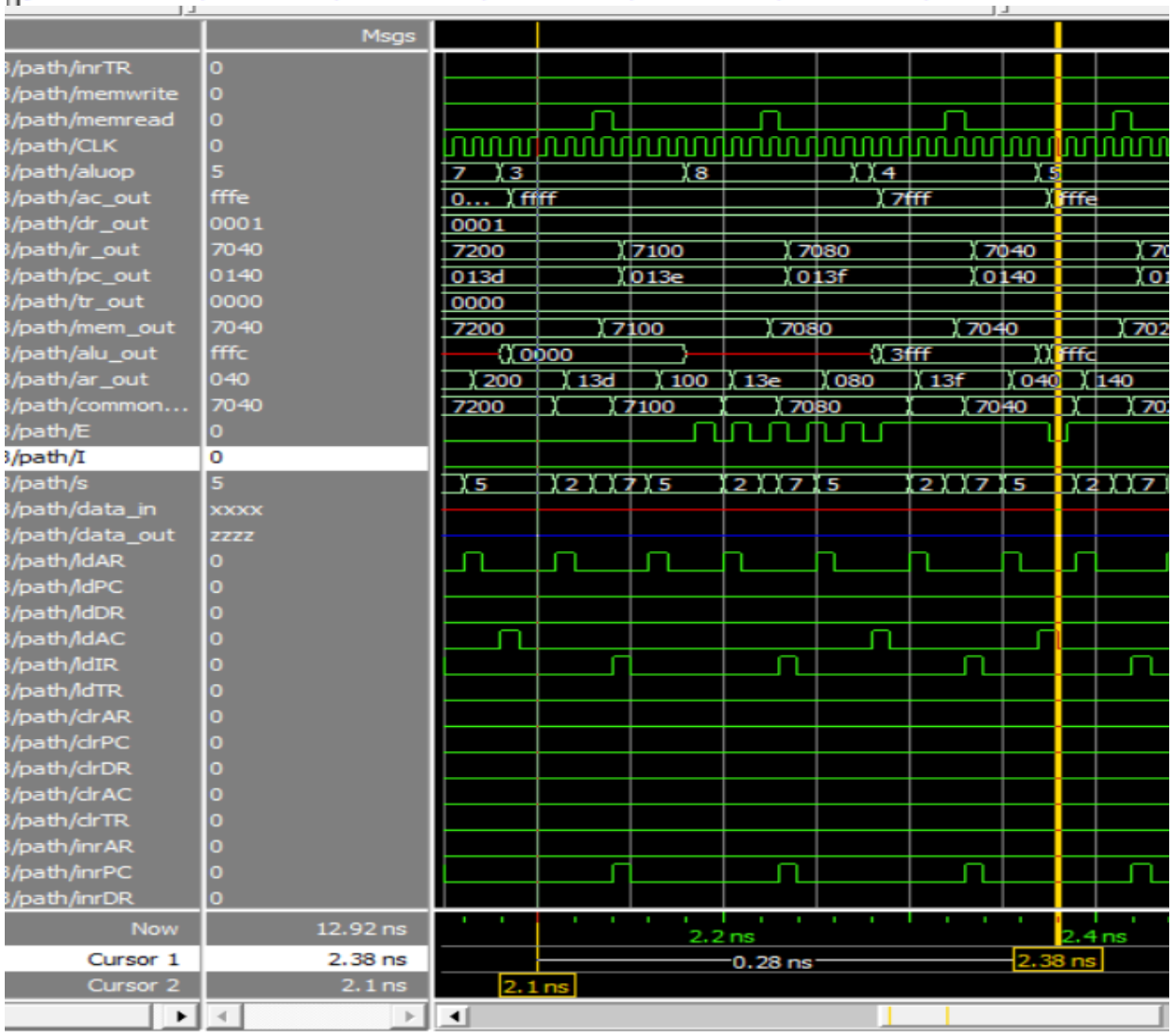
CME Instruction

```
#
# CME INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 13d, PC = 013d
#  T1: (IR <- M[AR], PC <- PC + 1) IR = 7100, PC = 013e
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 100
#  T3: (Register) I = 0
# Executing CME instruction...
#  T4D7B8: (E <- ~E) E = 1
# Time: 2190 | AR: 100 | PC: 013e | DR: 0001 | AC: ffff | IR: 7100 | I: 0
#
```



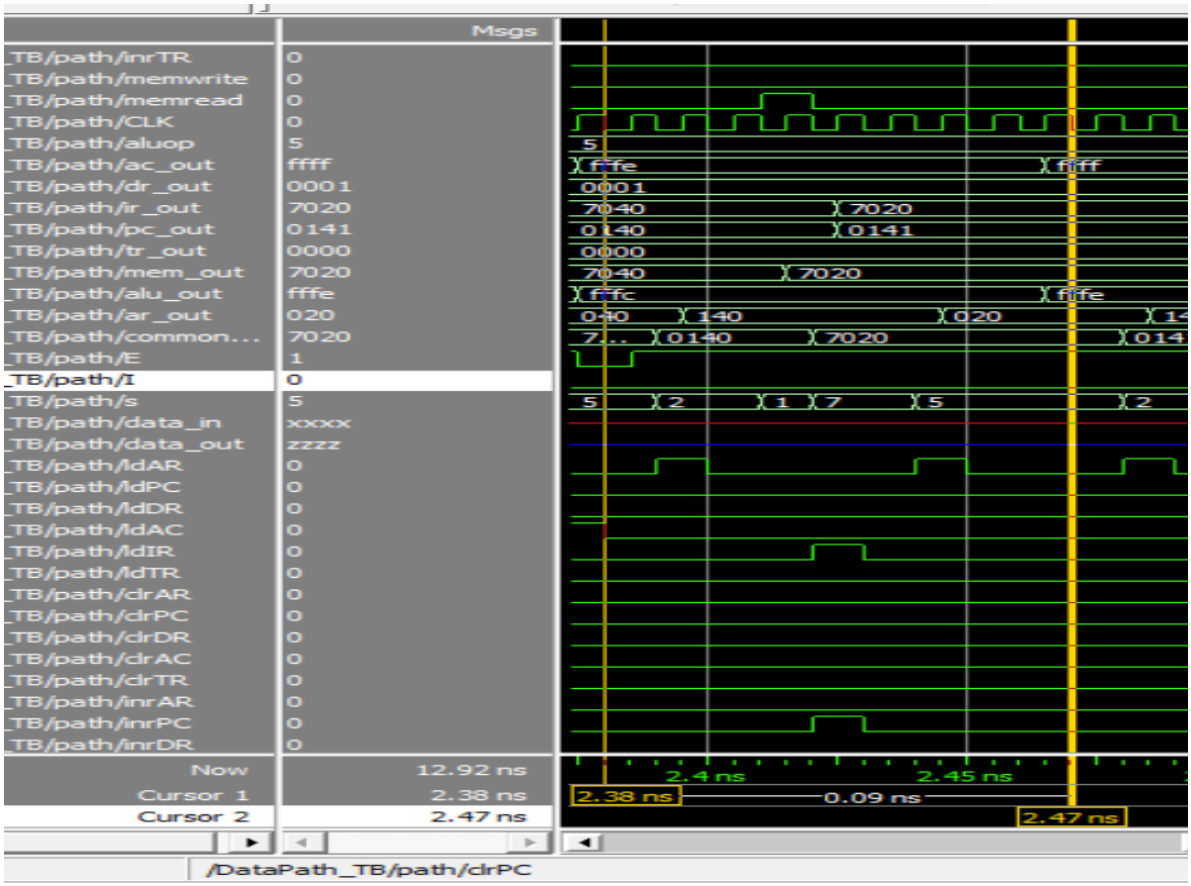
CIR AND CIL INSTRUCTIONS

```
#
# CIR INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 13e, PC = 013e
#  T1: (IR <- M[AR], PC <- PC + 1) IR = 7080, PC = 013f
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 080
#  T3: (Register) I = 0
# Executing CIR instruction...
# Time: 2290 | AR: 080 | PC: 013f | DR: 0001 | AC: 7fff | IR: 7080 | I: 0
#
# CIL INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 13f, PC = 013f
#  T1: (IR <- M[AR], PC <- PC + 1) IR = 7040, PC = 0140
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 040
#  T3: (Register) I = 0
# Executing CIL instruction...
# Time: 2380 | AR: 040 | PC: 0140 | DR: 0001 | AC: fffe | IR: 7040 | I: 0
```



INC INSTRUCTION

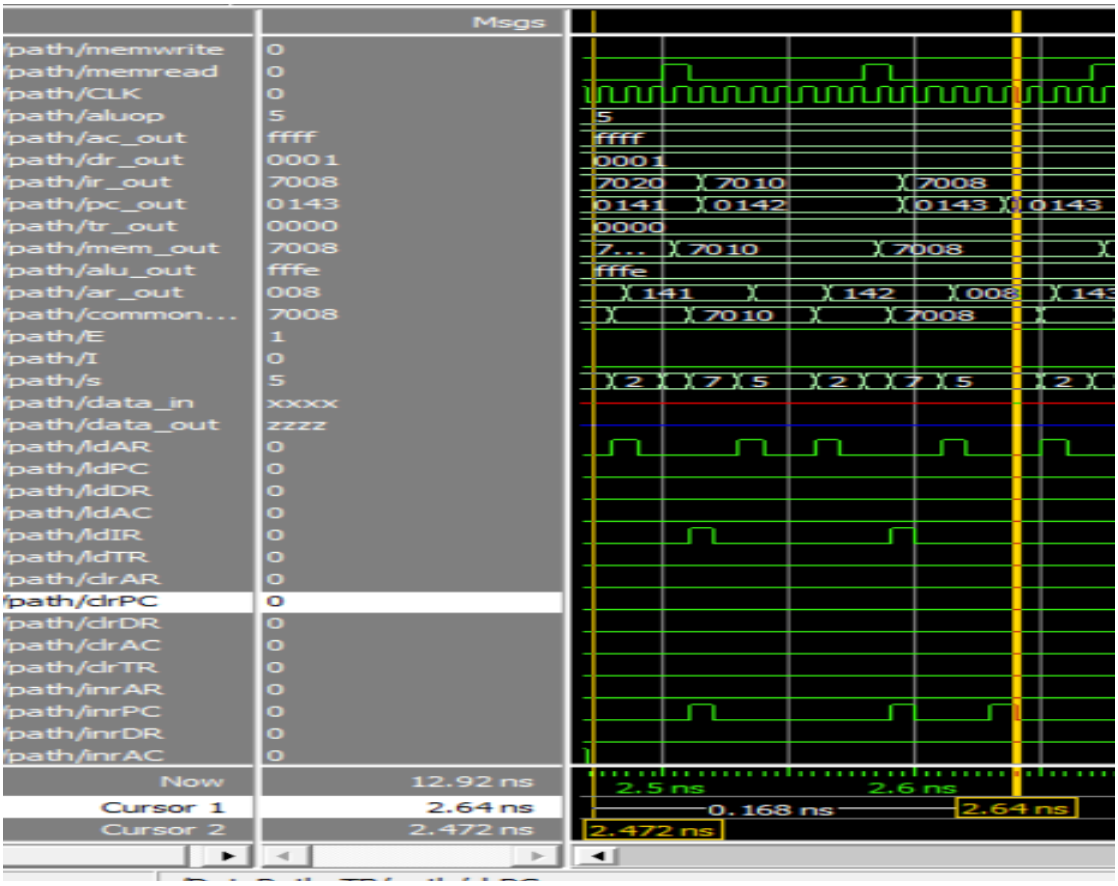
```
# INC INSTRUCTION
# Fetching instruction...
#   T0: (AR <- PC) AR = 140, PC = 0140
#   T1: (IR <- M[AR], PC <- PC + 1) IR = 7020, PC = 0141
# Decoding instruction...
#   T2: (AR <- IR[0-11], I <- IR[15]) AR = 020
#   T3: (Register) I = 0
# Executing INC instruction...
#   T4D7B5: (AC <- AC + 1) AC = ffff
# Time: 2470 | AR: 020 | PC: 0141 | DR: 0001 | AC: ffff | IR: 7020 | I: 0
```



SPA AND SNA INSTRUCTIONS

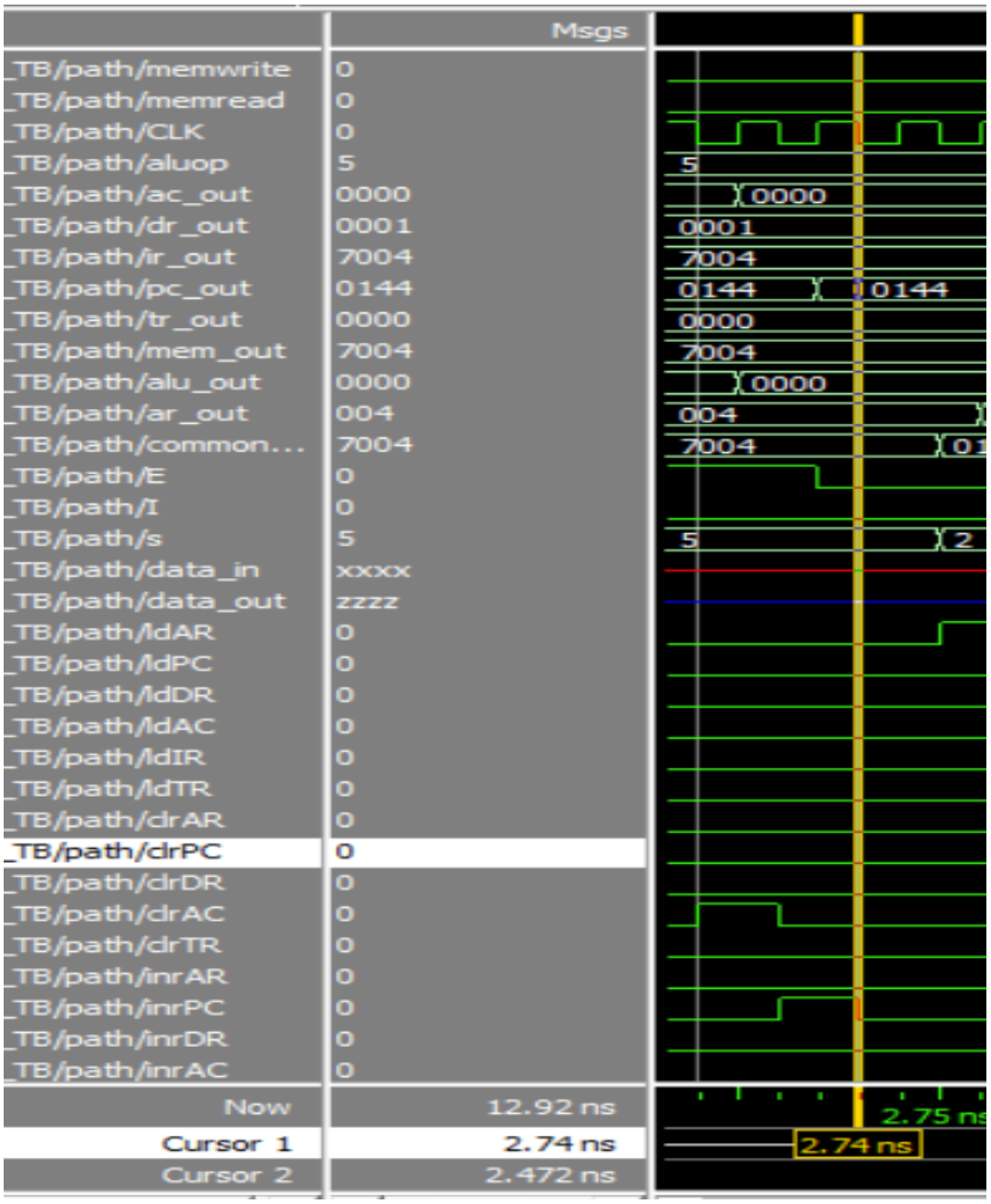
```
#
# SPA INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 141, PC = 0141
#  T1: (IR <- M[AR], PC <- PC + 1) IR = 7010, PC = 0142
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 010
#  T3: (Register) I = 0
# Executing SPA instruction...
#  T4D7B4: ([if(AC[15] == 0) : PC <- PC + 1]) PC = 0142, AC[15] = 1
# Time: 2550 | AR: 010 | PC: 0142 | DR: 0001 | AC: ffff | IR: 7010 | I: 0
```

```
#
# SNA INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 142, PC = 0142
#  T1: (IR <- M[AR], PC <- PC + 1) IR = 7008, PC = 0143
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 008
#  T3: (Register) I = 0
# Executing SNA instruction...
#  T4D7B3: ([if(AC[15] == 1) : PC <- PC + 1]) PC = 0144, AC[15] = 1
# Time: 2640 | AR: 008 | PC: 0144 | DR: 0001 | AC: ffff | IR: 7008 | I: 0
#
```



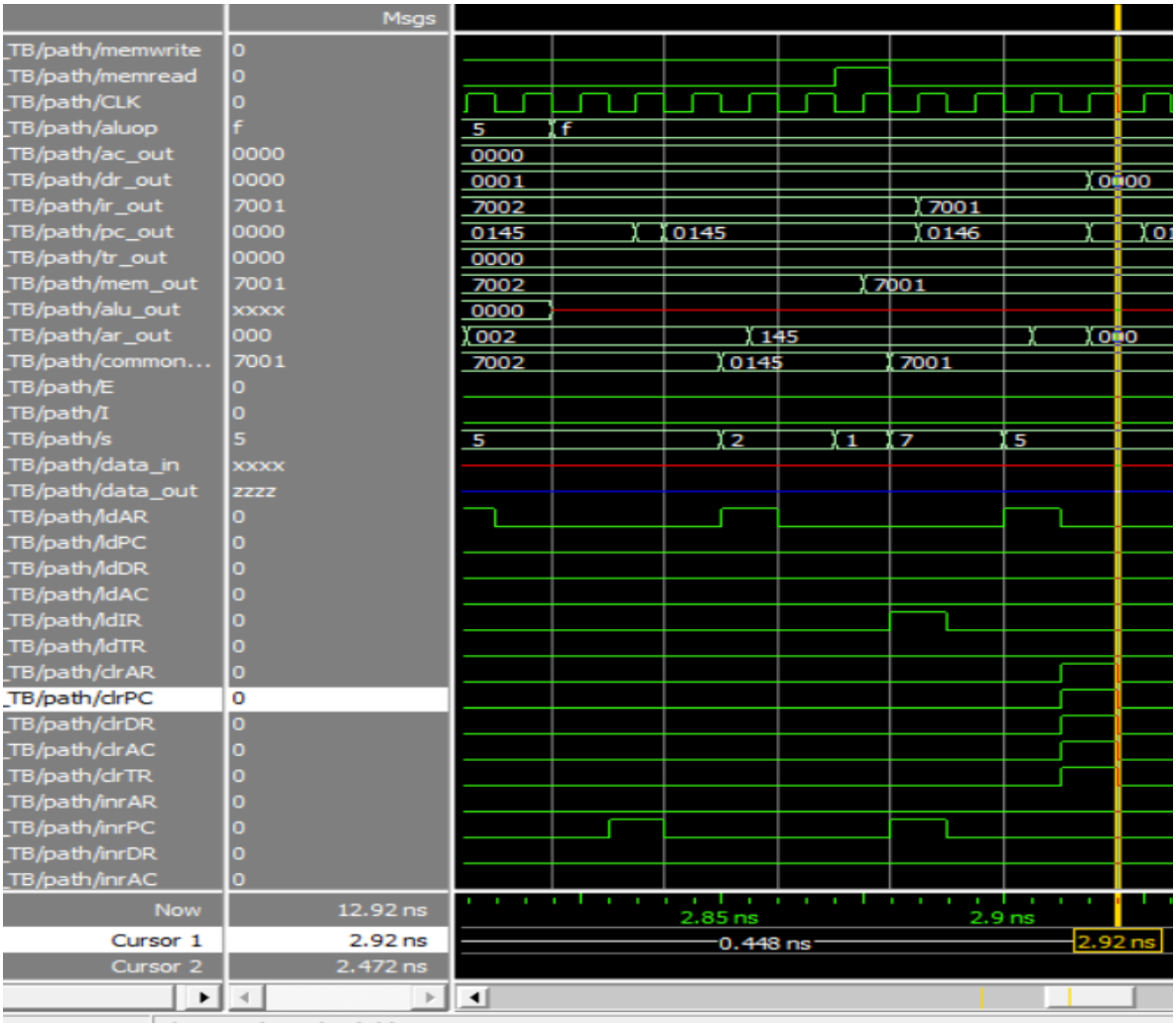
SZA INSTRUCTION

```
# SZA INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 143, PC = 0143
#  T1: (IR <- M[AR], PC <- PC + 1) IR = 7004, PC = 0144
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 004
#  T3: (Register) I = 0
# Executing SZA instruction...
#  T4D7B2: ([if(AC == 0) : PC <- PC + 1]) PC = 0145, AC = 0000
# Time: 2740 | AR: 004 | PC: 0145 | DR: 0001 | AC: 0000 | IR: 7004 | I: 0
```

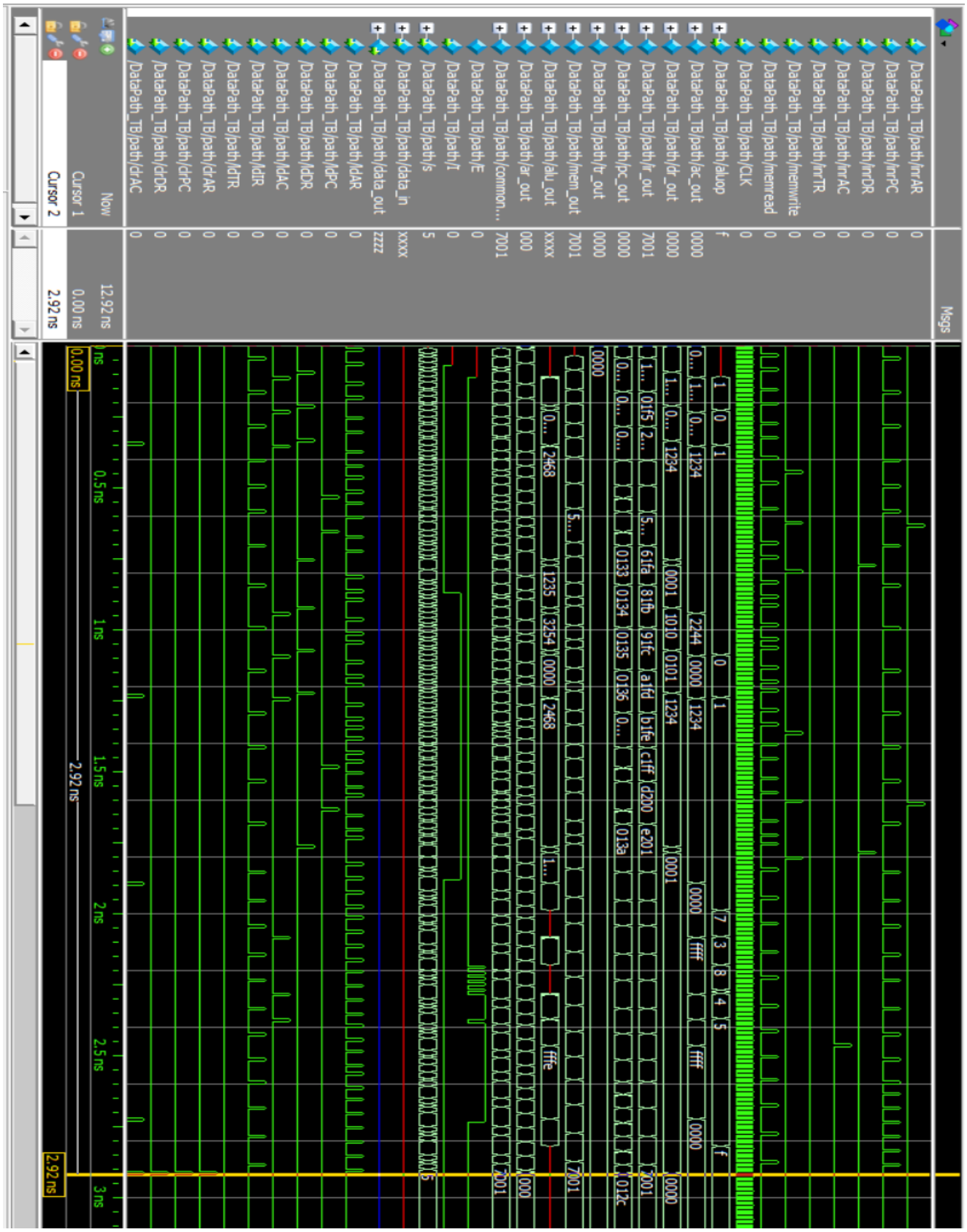


SZE AND HALT INSTRUCTIONS

```
#
# SZE INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 144, PC = 0144
#  T1: (IR <- M[AR], PC <- PC + 1) IR = 7002, PC = 0145
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 002
#  T3: (Register) I = 0
# Executing SZE instruction...
#  T4D7B1: ([if(E == 0) : PC <- PC + 1]) PC = 0146, E = 0
# Time: 2840 | AR: 002 | PC: 0146 | DR: 0001 | AC: 0000 | IR: 7002 | I: 0
#
# HLT INSTRUCTION
# Fetching instruction...
#  T0: (AR <- PC) AR = 145, PC = 0145
#  T1: (IR <- M[AR], PC <- PC + 1) IR = 7001, PC = 0146
# Decoding instruction...
#  T2: (AR <- IR[0-11], I <- IR[15]) AR = 001
#  T3: (Register) I = 0
# Time: 2920 | AR: 000 | PC: 0000 | DR: 0000 | AC: 0000 | IR: 7001 | I: 0
```



THE WHOLE PROCESS WAVE FORM



- NOTES :

- **We faced multiple issues in the code and fixed it using the divide and conquer method discussed in class, tracing and finding the bugs, such as the capital letters difference, wire and register definitions, common bus wrong connections, delay and timing issues, alu operations not executed properly, for example : in the LDA instruction the AC was not loaded in multiple tries, so we traced the code to find out that the opposite operand in the ALU code was getting the value, so we changed it and it worked as it should. So we applied debugging and found all the possible bugs we could and tried to give the best output we had.**

- **In the Datapath code, you will find a Data_in and Data_out variables that are not used, these data lined are defined for synthesis purposes in the future segments when we deal with the FPGA .**