RTL DESIGN OF DMA CONTROLLER

Team Members:

- Pratik Avinash Narkhede
- Tanmay Nitin Patil
- Abdul Hasan Imroze Mohammed

AGENDA:

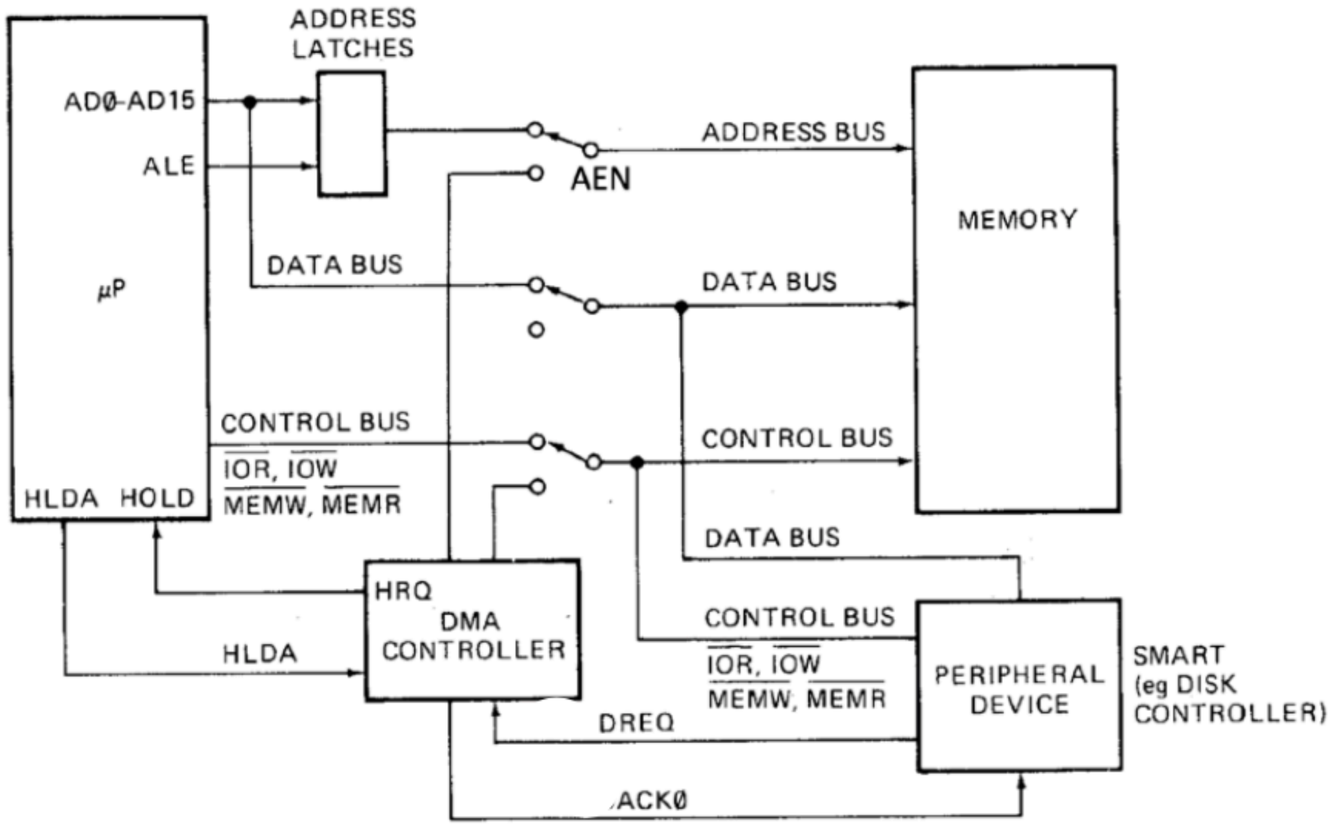
- Introduction to DMA Controller.
- Working of DMA Controller.
- High Level Design Architecture.
- High Level DMA Design.
- Priority Logic.
- Data Path Logic.
- Timing and Control Logic.
- SystemVerilog Constructs Used.
- Integration Hierarchy.
- Verification.
- Challenges Faced and Edification.
- Future Scope.
- References.

Introduction to DMA Controller:

- Direct Memory Access is a hardware device, which has a distinctive attribute in the system, which allow some of the hardware subsystem to access the main memory irrespective or independent of the Central Processing Unit (CPU).

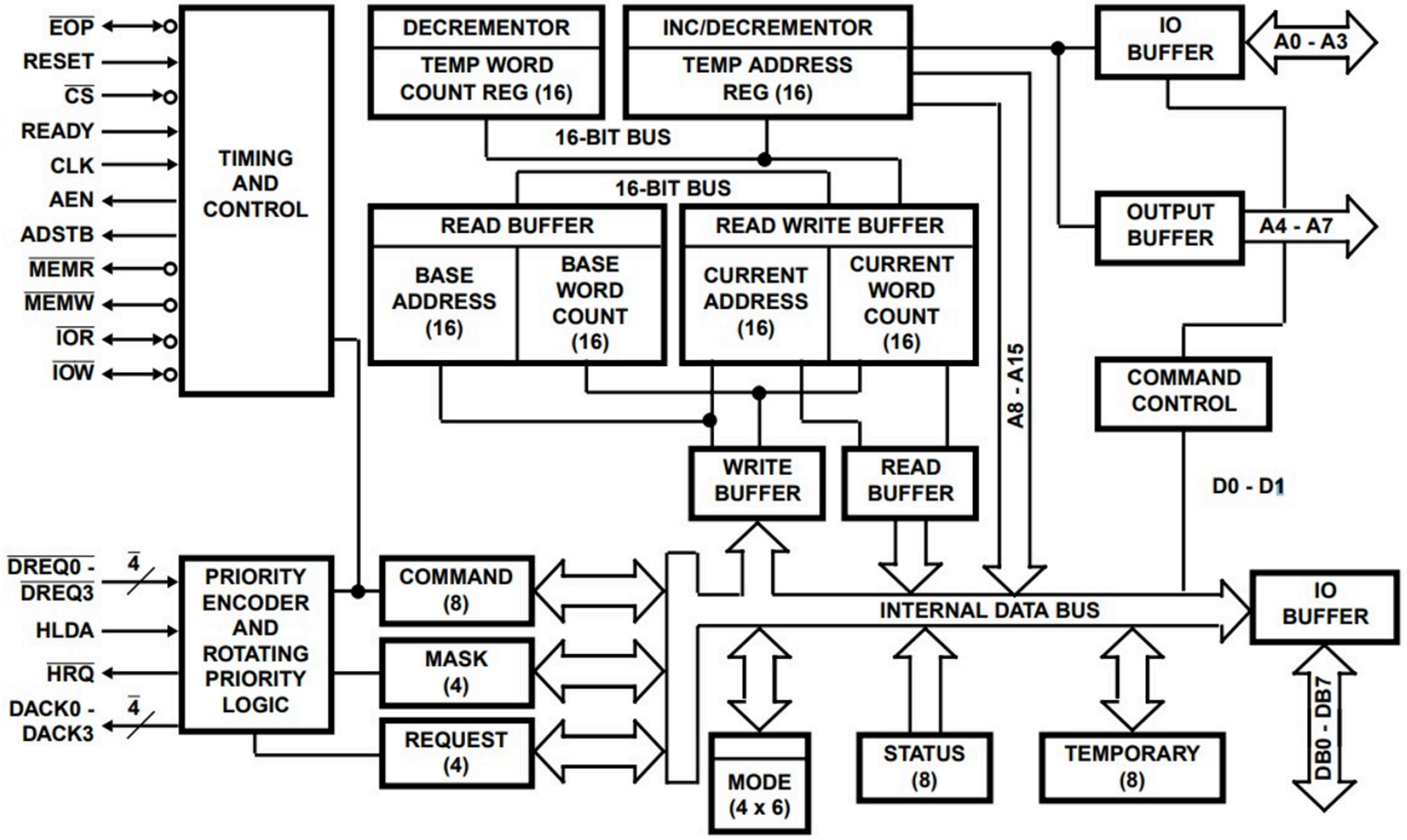
- The main objective of the DMA Controller is to enhance the system performance by directly data from I/O device to the main memory through DMA Controller without the interference of the CPU.

Working Of DMA Controller:

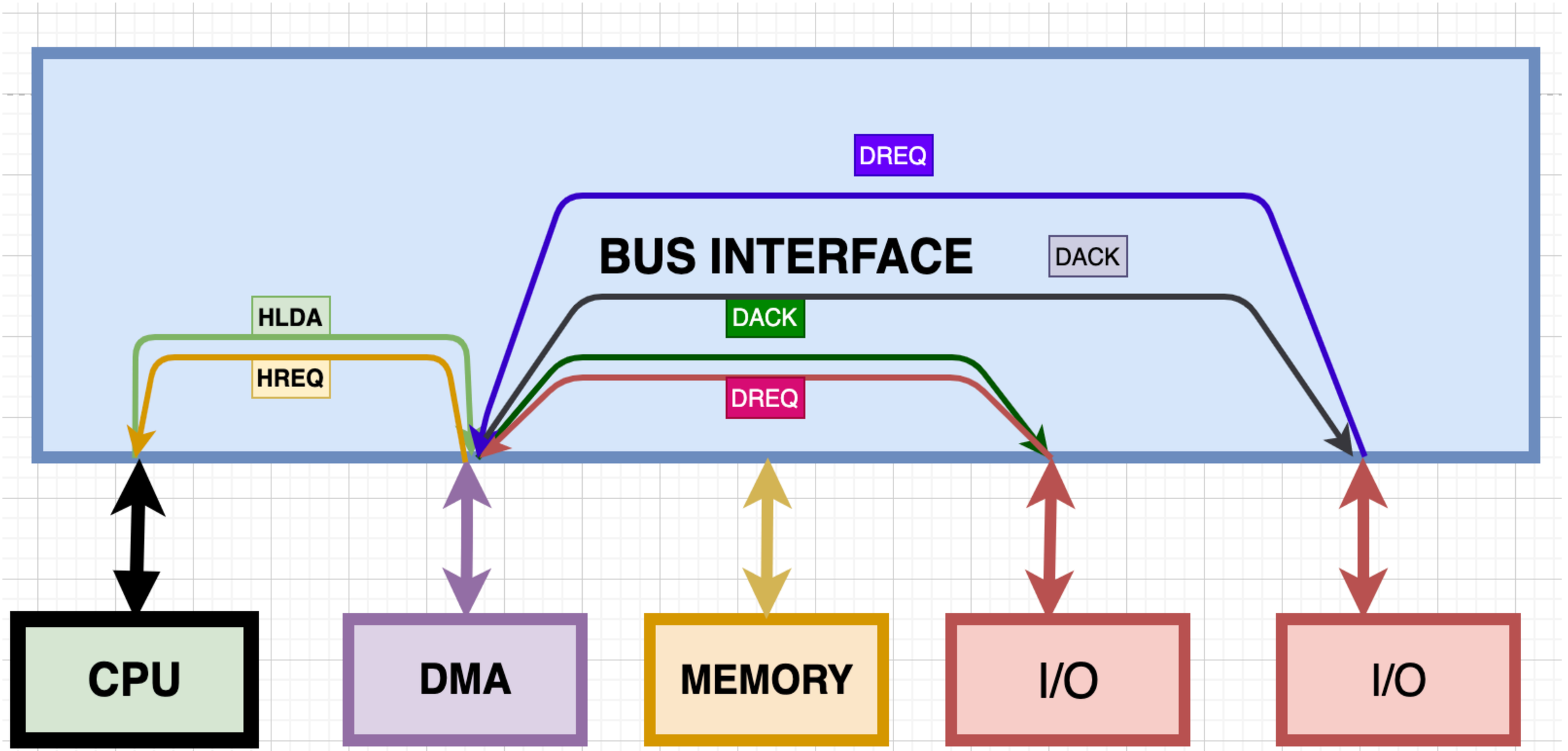


Block Diagram of DMA Controller interacting with Microprocessor, Memory and Peripheral Devices.

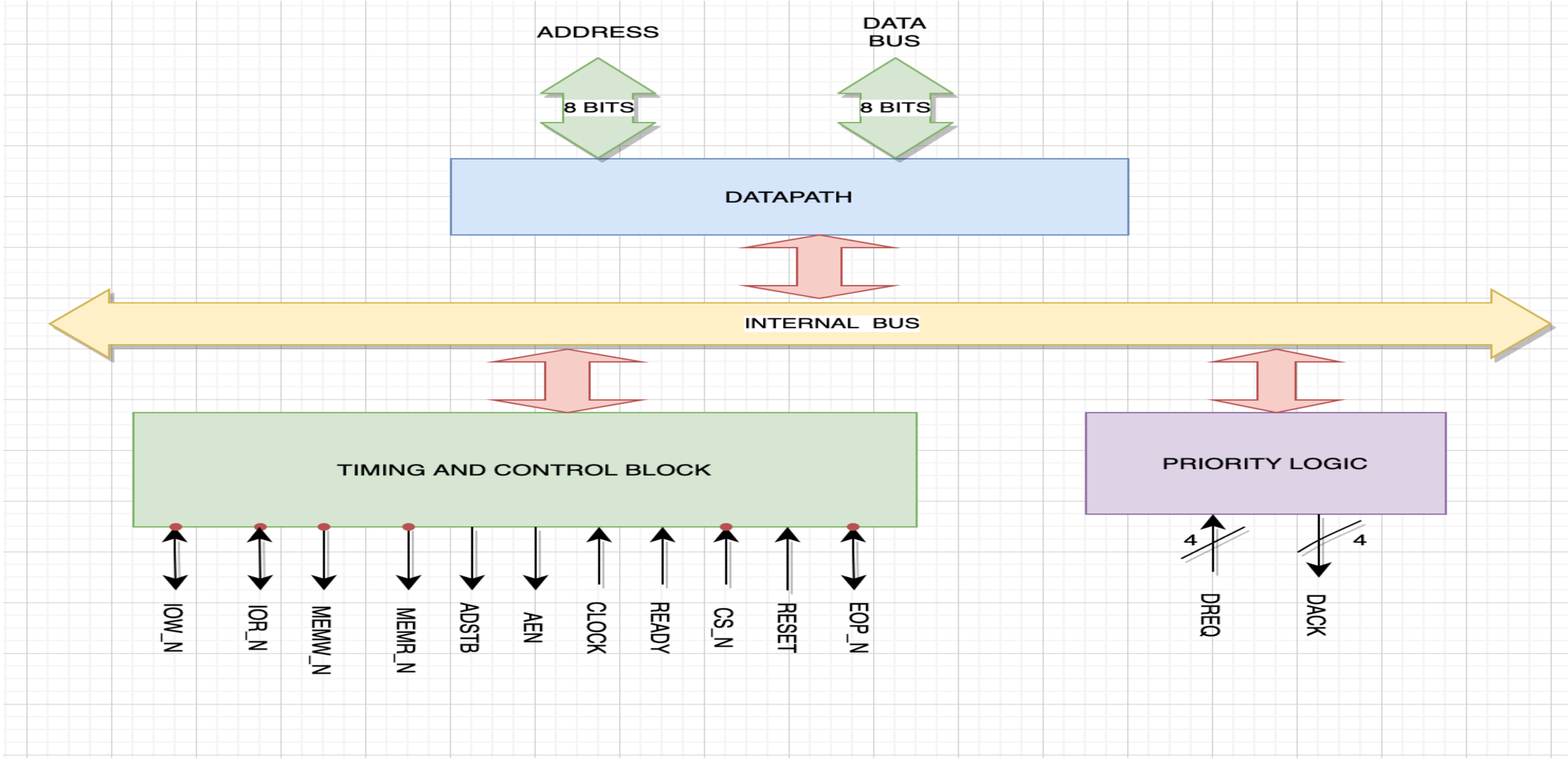
Working Of DMA Controller:(Continuation)



High Level Design Architecture:

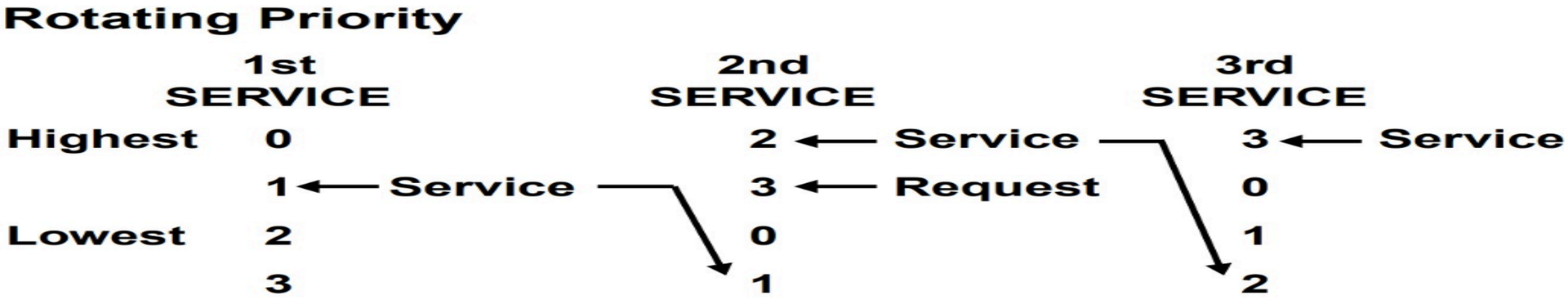


High Level DMA Design:



Priority Logic:

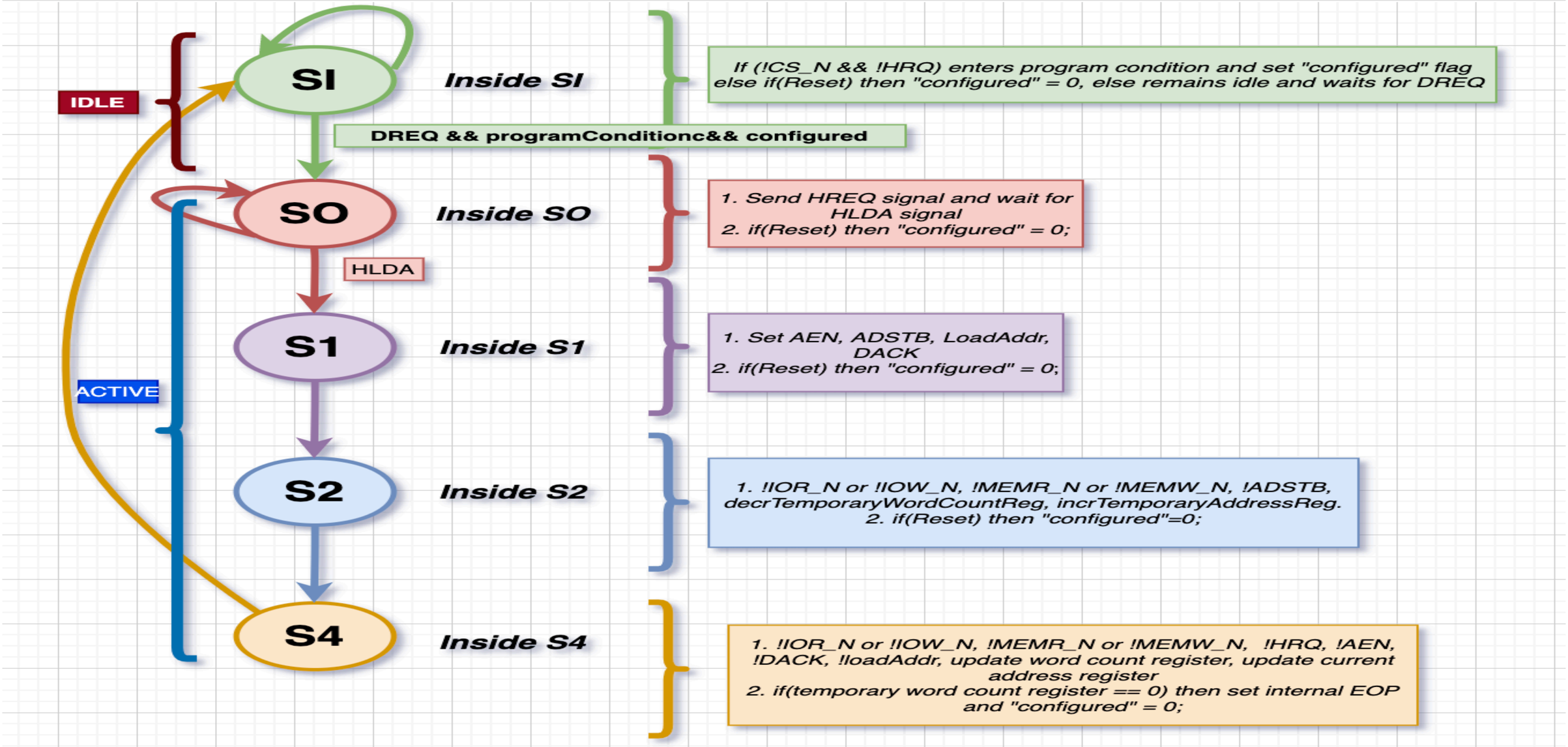
- The 8237A DMA Controller has to types of priority logic.
- **Fixed Priority:** Every I/O device has an unique rank. Channels are given fixed priority based on the descending value of their number.
- **Rotating Priority:** The Last channel to get acknowledgement gets the lowest priority with others rotating accordingly.



Data Path Logic:

- Handles all the Internal Registers of the DMA.
- In ideal cycle Command Register, Mode Register, Base Address Register, Base Word Count Register, Current Address Register, Current Word Count Registers are programmed.
- Increments the Address Register, decrements the word count, updates the status register in the active cycle.
- Makes the internal flip flop to a known state to read/write the address/word count register.

Timing and Control Logic:



SystemVerilog Constructs Used:

- Assertions
- Interfaces
- Package
- Priority Case and Unique Case
- Structures
- Enumerated types

Interface Construct:

```
interface dmaInternalSignalsIf(input logic CLK, RESET);

  logic programCondition;
  logic loadAddr;

  logic assertDACK;
  logic deassertDACK;

  logic intEOP;
  logic updateCurrentWordCountReg;
  logic updateCurrentAddressReg;
  logic decrTemporaryWordCountReg;
  logic incrTemporaryAddressReg;

  modport timingAndControl(
    output assertDACK,
    output deassertDACK,
    output intEOP,
    output loadAddr,
    output programCondition,
    output updateCurrentWordCountReg,
    output updateCurrentAddressReg,
    output decrTemporaryWordCountReg,
    output incrTemporaryAddressReg
  );
```

Package Construct:

```
1   package dmaRegConfigPkg;
2
3   parameter ADDRESSWIDTH = 16;
4   parameter DATAWIDTH = 8;
5   parameter CHANNELS = 4;
6   parameter REGISTERADDRESS = 4;
7   parameter PERIPHERALS = 2;
8
9   endpackage
10
```

Priority Case and Unique Case Construct:

```
//Fixed Priority
if(!intRegIf.commandReg.priorityType && intSigIf.assertDACK)
  begin
    priority case(1'b1)
      busIf.DREQ[0] : busIf.DACK = 4'b0001 << 0;
      busIf.DREQ[1] : busIf.DACK = 4'b0001 << 1;
      busIf.DREQ[2] : busIf.DACK = 4'b0001 << 2;
      busIf.DREQ[3] : busIf.DACK = 4'b0001 << 3;
    endcase
  end
```

```
//Next state Logic
always_comb
  begin
    nextState = state;
    unique case (1'b1)
      state[SIIndex]: if (|PLbusIf.DREQ && intSigIf.programCondition == 1'b0 && configured)
        nextState = S0;
      state[S0Index]: if (PLbusIf.HLDA )
        nextState = S1;
      state[S1Index]:
        nextState = S2;
      state[S2Index]:
        nextState = S4;
      state[S4Index]:
        nextState = SI;
    endcase
  end
```

Assertions Construct:

```
//assertion to check if Command Register holds valid data
property writeCommandRegister_p;
  @(posedge busIf.CLK)
  disable iff (busIf.RESET)
  (ldCommandReg) |=> (intRegIf.commandReg == $past(ioDataBuffer));
endproperty

writeCommandRegister_a : assert property (writeCommandRegister_p);


  //assertion to check if Command Register is zeroed on Reset
property commandRegZeroOnReset_p;
  @(posedge busIf.CLK)
    (busIf.RESET) |=> (intRegIf.commandReg == '0);
endproperty

commandRegZeroOnReset_a : assert property (commandRegZeroOnReset_p);


    //assertion to check if Mode Register is zeroed on Reset
property modeRegZeroOnReset_p;
  @(posedge busIf.CLK)
    (busIf.RESET) |=> (intRegIf.modeReg[0] == '0 and intRegIf.modeReg[1] == '0 and intRegIf.modeReg[2] == '0 and intRegIf.modeReg[3]
== '0);
    endproperty

modeRegZeroOnReset_a : assert property (modeRegZeroOnReset_p);
```

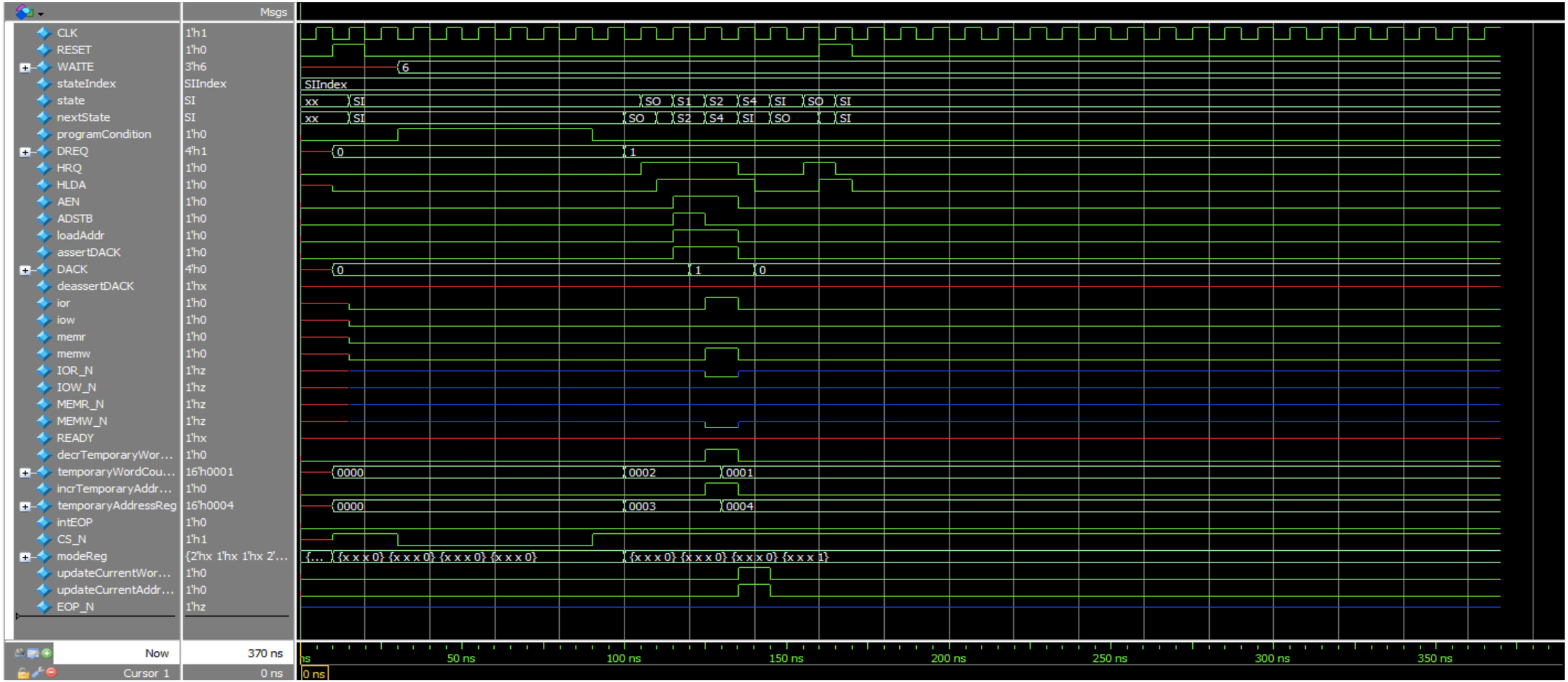
Structure Construct:

```
struct packed{
  logic dackSense     ; //0-DACK sense active low, 1-DACK sense active high
  logic dreqSense     ; //0-DREQ sense active high, 1-DREQ sense active low
  logic writeSelection; //0-Late Write Selection, 1-Extended Write Selection, X-if CompressedTiming=1
  logic priorityType  ; //0-Fixed Priority, 1-Rotating Priority
  logic timing        ; //0-Normal Timing, 1-Compressed Timing
  logic controller    ; //0-Controller Enable, 1-Controller Disable
  logic c0AddressHold ; //0-Channel 0 address hold disabled,  1-Channel 0 address hold enable, X-if MemToMem=0
  logic memToMem      ; //0 - Memory to Memory disable, 1 - Memory to Memory enable
} commandReg;
```

Enumerated Types Construct:

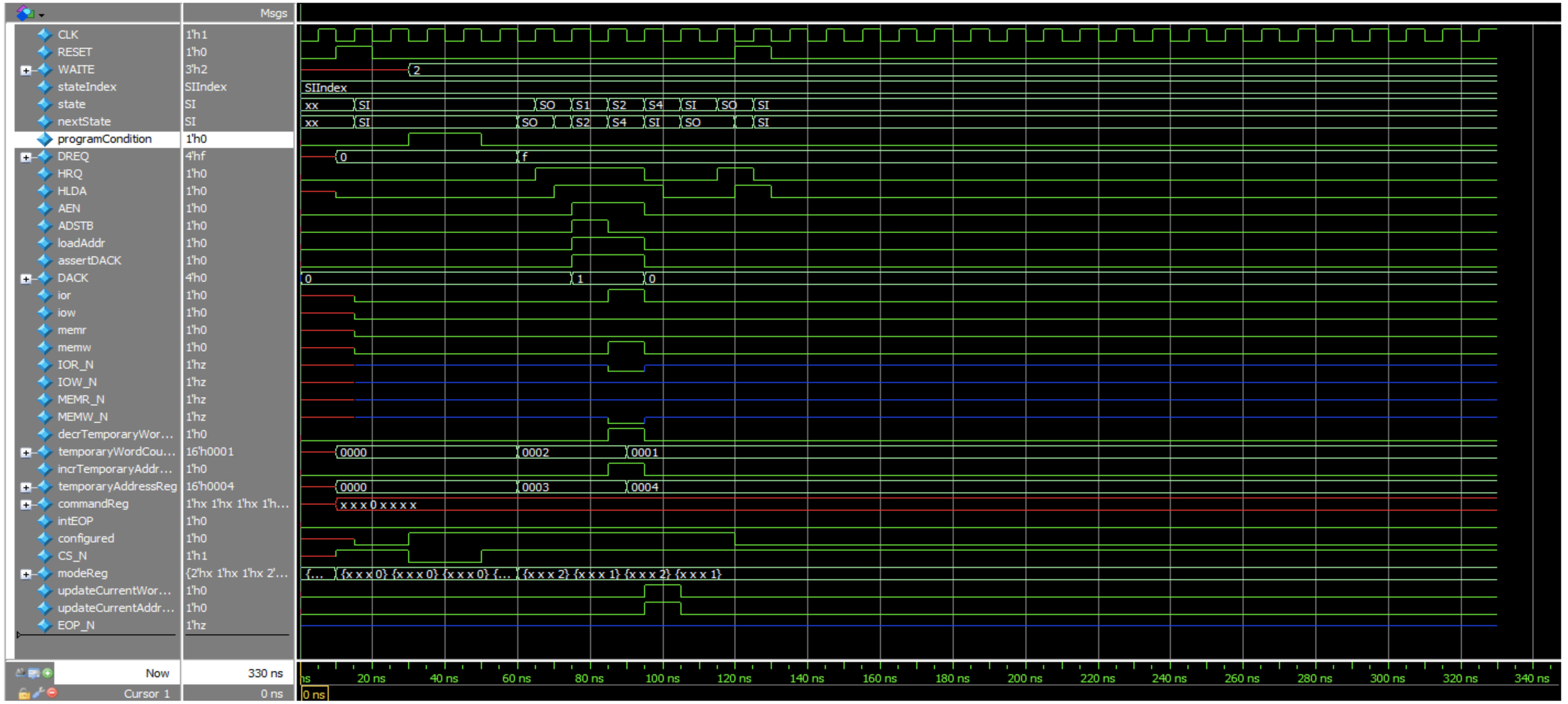
```
enum {SIIndex = 0,
      SOIndex = 1,
      S1Index = 2,
      S2Index = 3,
      S3Index = 4,
      S4Index = 5} stateIndex;

enum logic [5:0] {SI = 6'b000001 << SIIndex,
                  SO = 6'b000001 << SOIndex,
                  S1 = 6'b000001 << S1Index,
                  S2 = 6'b000001 << S2Index,
                  S3 = 6'b000001 << S3Index,
                  S4 = 6'b000001 << S4Index} state, nextState;
```

Integration Hierarchy:



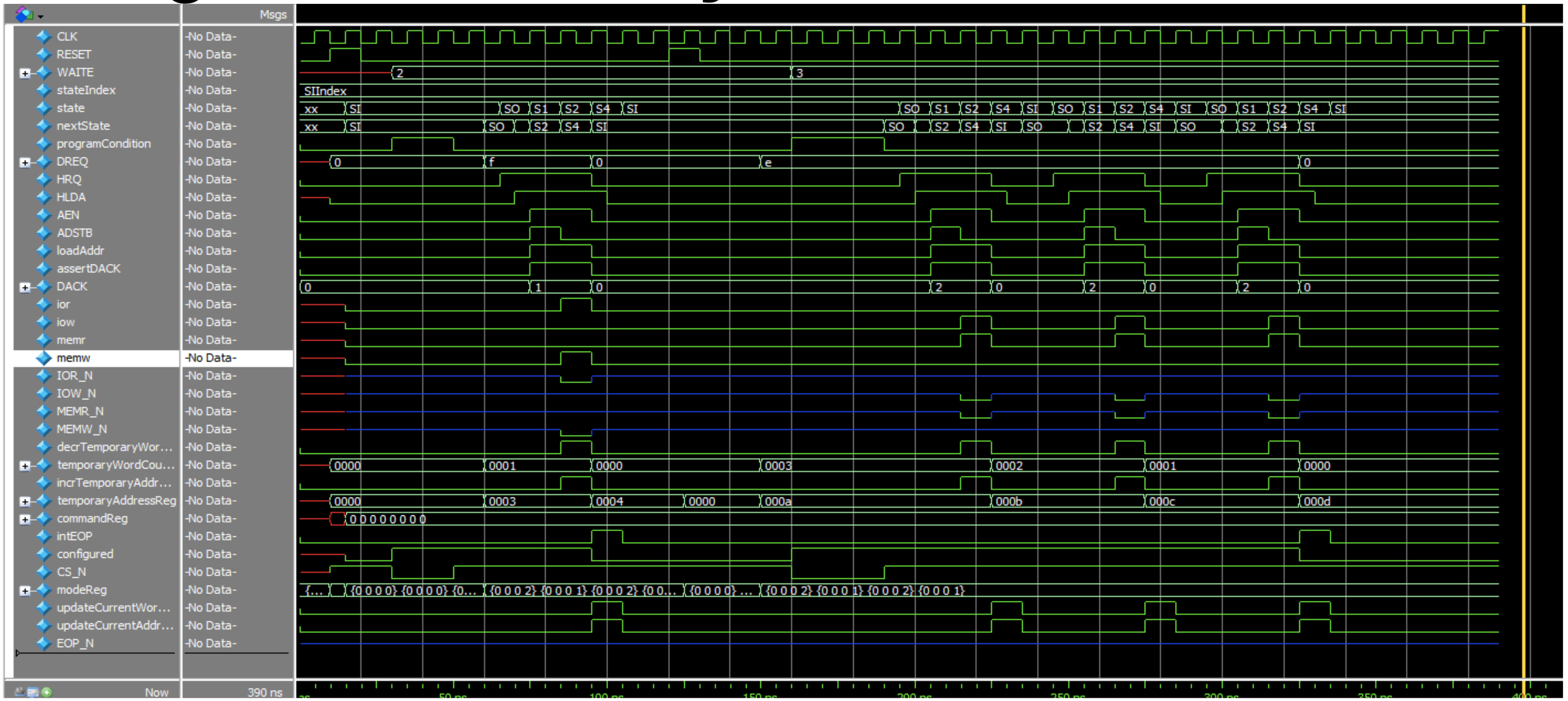
Timing and Control FSM waveform

Integration Hierarchy:



Priority Logic and Timing and Control Integration Waveform.

Integration Hierarchy:



Independent DMA waveform.

Verification:

- **Unit Level Testing:**
- Priority Logic: Applied directed test cases.
- Timing and Control Logic: Applied directed test cases.
- Datapath Logic: Applied directed test cases.
- Used assertions.
- Used bottom-up approach to test the design.
- **System Level Testing:**
- Applied directed test cases at the system level.
- Debugging was done mostly with the help of printing $display, waveforms and error messages.

Task Used for Register Configuration:

```
task ldCommandReg(logic [DATAWIDTH-1:0] cmdData);
  //command register
  cpu.CS_N ='0;
  {A3, A2, A1, A0} = 4'h8;
  IOR_N = '1;
  IOW_N ='0;
  DB = cmdData;
endtask

task ldModeReg(logic [DATAWIDTH-1:0] modeData);
  //mode register
  cpu.CS_N ='0;
  IOR_N = '1;
  IOW_N ='0;
  {A3, A2, A1, A0} = 4'hB;
  DB = modeData; //single transfer channel 0 write mode
  //DB<= 8'h48; //single transfer channel 0 read mode
endtask


task ldBaAddrLB(logic [REGISTERADDRESS-1:0] channelAddr, logic [(ADDRESSWIDTH/2)-1:0] lowerByte);
  //write base address & current address lower byte
  cpu.CS_N  = '0;
  IOR_N ='1;
  IOW_N ='0;
  {A3, A2, A1, A0} = channelAddr; //channel 0
  //{A3, A2, A1, A0} <= 4'h2; //channel 1
  DB =lowerByte;
endtask
```

Challenges Faced and Edification:

- System level Integration and environmental integration took a lot of time debugging simulation time errors.
- Comprehension of the data sheet for correct implementation of the design
- Modelling Bi-directional Signals.
- Identifying issues in the design with aid of test bench.

Future Scope and Intent:

- Run Back-end simulation, work on the design from synthesis to GDSII.
- Assertion Based Verifications.
- Improve the design to support Block Transfer Mode, Demand Transfer and Cascade transfer mode.

Source Code:

- Github Link.

References:

- 8237 Data Sheet(Intel).
- ECE 585 Basic I/O Lecture Slides.
- Direct Memory Access - (Wikipedia).
- Round Robin Arbitration source-1.
- Round Robin Arbitration source-2.