Project Report:

MEMORY ISSUES IN PRET MACHINES

Nishant R. Shah (nrs2127)
Submitted on: 21st December, 2008.

1. ABSTRACT

In a processor design the premier issues with memory are (1) main memory allocation and (2) interprocess communication. These two mainly affect the performance of the memory system. The goal of this paper is to formulate a deterministic model for memory systems of PRET, taking into account all the intertwined parallelism of modern memory chips.

Studying existing memory models is necessary to understand the implications of these factors to realize a perfectly time predictable memory system.

2. INTRODUCUTION

1.1 MEMORY MANAGEMENT in PRET ARCHITECTURE

No large modern memories are pipelined; all of them are parallelized for maximum performance. Thus, to access the memory for thread-interleaved pipelined architecture has to be approached in a different way. For this purpose the use of window mechanism was conceptualized. Each thread would have a window slot in which it can access main memory. This provides predictable access to memory. This was called the Memory Wheel.

The PRET machine includes a six-stage thread-interleaved pipeline in which each stage executes a separate hardware thread to avoid the need for bypasses. Each thread has its own register file, local on-chip memory, and assigned region of off-chip memory. The THREAD CONTROLLER component is a simple round-robin thread scheduler, similar to time division multiplexing. Each thread occupies one pipeline stage at all times. To handle the stalls of the pipeline predictably a replay mechanism was introduced, that simply repeats the same instruction until the operation completes. Thus, the stalling of one thread does not affect any of the other threads. The round-robin execution of threads avoids memory consistency issues

The memory hierarchy consists of separate fast on-chip scratchpad memories (SPM) instead of cache memories due to the following reasons:

1. Non-deterministic behavior
2. Poor performance for multimedia application with regular data access patterns
3. Higher power consumption

SPMs are used for instruction and data, and also serve as large off-chip main memory. They are connected to a direct memory access (DMA) controller responsible for moving data between main memory and the SPMs.

If all threads were to access the main off-chip memory as and when they required, then the access times for each thread will vary according to the memory patterns and not the structure of the program. This would introduce non-determinism and unpredictability beating the whole purpose of the PRET design. Hence, to ensure time predictability, access to the off-chip main memory is only allowed through the memory wheel. Each thread is given a time slice to access the memory via the wheel. If a thread misses the window, then it blocks using the Replay instruction till it gets its chance again.

Each window slot lasts for 13 cycles and it is now very important to know that each access can take anywhere between 13 and 90 cycles. This number is now based on when the request is made and not on access patterns. Now, if a thread starts its access on the first cycle of its window, the access takes exactly 13 cycles. Otherwise, the thread blocks until its window reappears, this may take up to 77 cycles. A successful access after just missing the first cycle of its window results in 77 + 13 = 90 cycles.

What PRET expects from DRAM?

All DRAM memories are banked and designed to perform Burst transfers and yet have uniform latencies. Thus in PRET could advantage of this fact and design its off-chip memory using DRAM. It would be very convenient, if a block of data can be filled in a window of the wheel. Also as these memories are banked, we can give each thread a bank and this could increase performance with time predictability.

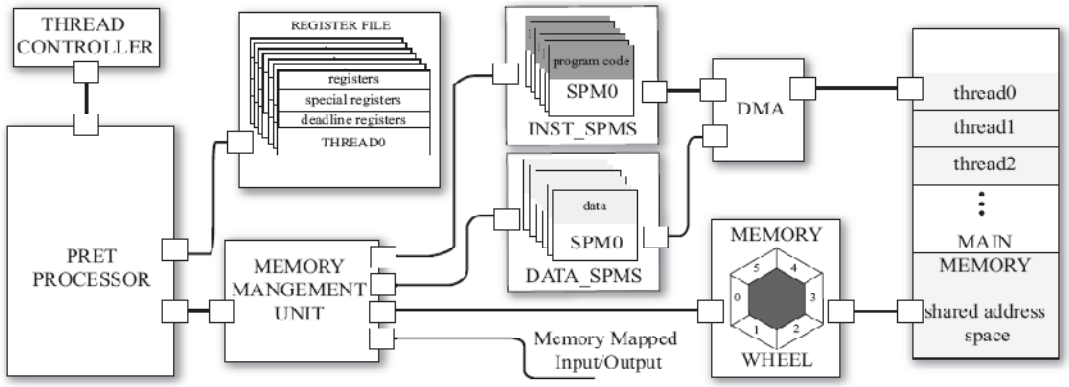


Figure 1: Block Diagram of PRET Architecture

3. UNDERSTANDING DDR2:

DDR-2 is the latest generation of computer memory, which has evolved from DDR (or called DDR-1) memory. DDR-2 has several advantages over DDR-1, including higher speeds, lower power consumption, smaller physical sizes, and greater MB module sizes available. Also DDR-2 comes with 8-bank memory interleaving and 4n-prefetch architecture.

Now for PRET machines we are trying to dedicate each bank of the RAM chip to a bank, we need minimum 6 banks and hence the safest option would be 8-bank memory. The available size for an 8-bank configuration is 1Gb and above. This we select the 16meg x 8 x 8 bank RAM configuration to implement main memory using DDR-2 for PRET machines.

3.1 BASIC WORKING

It operates from a differential clock (CK and CK#). Positive edge is when CK goes high and CK# goes low. All commands are registered at every positive edge of CK. Input data is registered on both edges of DQS whereas output data is registered on both edges of DQS as well as both edges of CK. DQS is a bidirectional data strobe used for data capture at the receiver.

Read and write accesses to the DDR2 SDRAM are burst oriented; accesses start at a selected location and continue for a burst length of four or eight in a programmed sequence. Accesses begin with the registration of an Active command, which is then followed by a Read or Write command. The address bits registered coincident with the active command are used to select the bank and row to be accessed (BA0-BA2 select the bank; A0-A15 select the row). The address bits registered coincident with the Read or Write command are used to select the starting column location for the burst access and to determine if the auto precharge command is to be issued. Prior to normal operation, the DDR2 SDRAM must be initialized.

As we already noticed, these memory chips are highly pipelined and its multibank architecture enables concurrent operations. This concurrent operation provides high effective bandwidth by hiding the overhead times for column and activation precharge.

The addressing, important pins/balls and functional block diagram of a DDR-2 chip are all as show below.

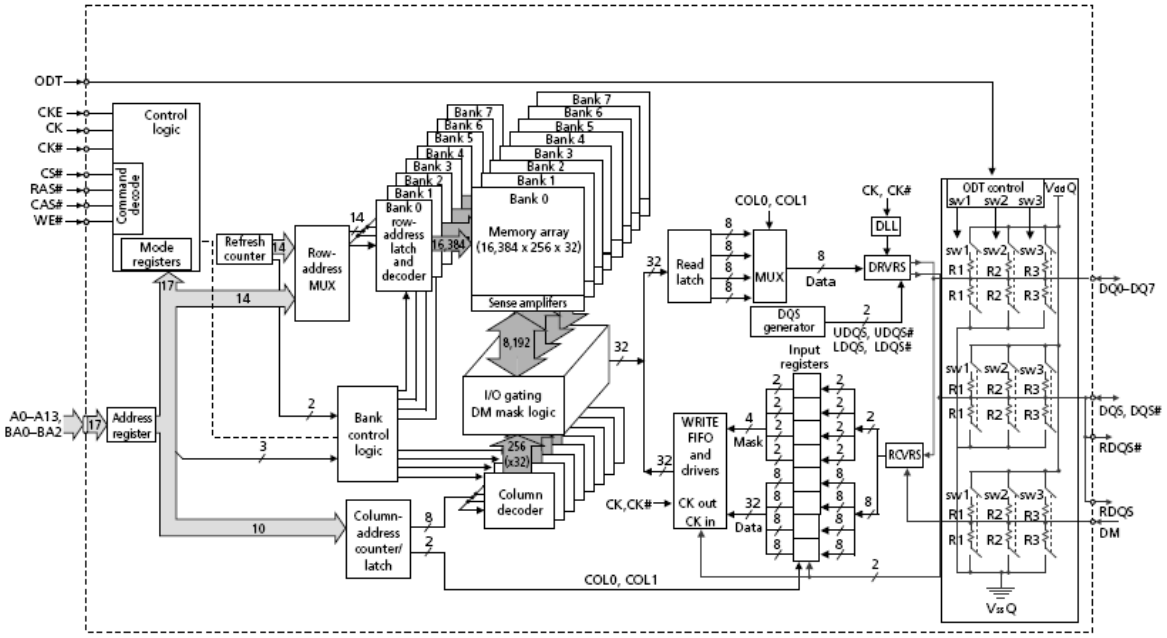
# of Banks	8
Bank Address	BA0-BA2
Auto Precharge	A10/AP
Row Address	A0-A13
Column Address	A0-A9
Page size	IKB

Table 1: Memory addressing

PIN/BALL	TYPE	FUNCTIONALITY
CK/CK#	Input	CK and CK are differential clock inputs
CKE	Input	CKE HIGH activates, and CKE LOW deactivates, internal clock signals and device input buffers and output drivers.
CS#	Input	Commands are masked when CS is registered HIGH. CS provides for external Rank selection on systems with multiple Ranks.
RAS#, CAS#, WE#	Input	Command Inputs: RAS, CAS and WE (along with CS) define the command being entered.
BA0-BA2	Input	Bank Address Inputs: BA0 - BA2 define to which bank an Active, Read, Write or Precharge command is being applied.
A0-A15	Input	Provide the row address for Active commands and the column address and Auto Precharge bit for Read/Write commands to select one location out of the memory array in the respective bank
DQ0-7	in/out	Data Input/ Output: Bi-directional data bus.

Table2: Important Pins/Balls of DDR2-2

Figure2: Functional Block Diagram of DDR-2 (128Meg x 8)



3.2 MODE REGISTER:

The mode register is used to define the specific mode of operation of the DDR2 SDRAM. This definition includes the selection of a burst length, burst type, CAS latency, operating mode, DLL RESET, write recovery, and power-down mode. Contents of the mode register can be altered by re-executing the LOAD MODE (LM) command. If the user chooses to modify only a subset of the MR variables, all variables must be programmed when the command is issued. The default value of the mode register is not defined; therefore the mode register must be programmed during initialization for proper operation

The LM command can only be issued (or reissued) when all banks are in the precharged state (idle state) and no bursts are in progress. The controller must wait the specified time tMRD before initiating any subsequent operations such as an ACTIVATE command. Violating either of these requirements will result in an unspecified operation.

The mode register is divided into various fields depending on functionality. Burst length is defined by A0 - A2 with options of 4 and 8 bit burst lengths. The burst length decodes are compatible with DDR SDRAM. Burst address sequence type is defined by A3, CAS latency is defined by A4 - A6. The DDR2 does not support half clock latency mode. A7 is used for test mode. A8 is used for DLL reset. A7 must be set to LOW for normal MRS operation. Write recovery time WR is defined by A9 - A11. Refer to the table below for specific codes.

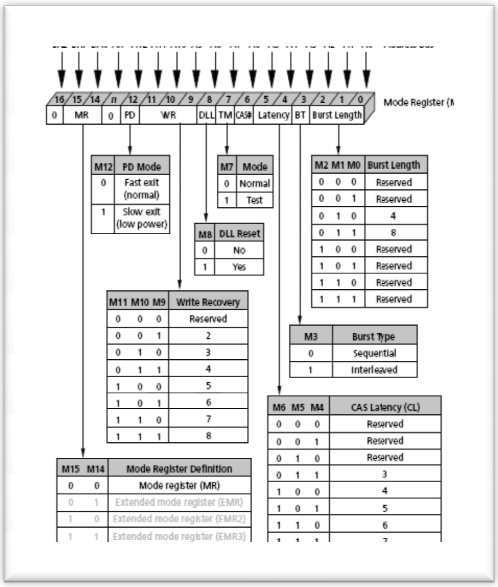


Figure 3: Mode Register

3.3 COMMANDS & MODES:

The various commands used for functioning of DDR RAMs are:

DESELECT

The DESELECT function (CS# HIGH) prevents new commands from being executed by the DDR2 SDRAM. The DDR2 SDRAM is effectively deselected. Operations already in progress are not affected.

NO OPERATION (NOP)

The NO OPERATION (NOP) command is used to instruct the selected DDR2 SDRAM to perform a NOP (CS# is LOW; RAS#, CAS#, and WE are HIGH). This prevents unwanted commands from being registered during idle or wait states. Operations already in progress are not affected.

LOAD MODE (LM)

The mode registers are loaded via bank address and address inputs. The bank address balls determine which mode registers will be programmed. The LM command can only be issued when all banks are idle, and a subsequent executable command cannot be issued until t_{MRD} is met.

ACTIVATE

The Bank Activate command is issued by holding CAS# and WE# HIGH with CS# and RAS# LOW at the rising edge of the clock. The bank addresses BA0- BA2 are used to select the desired bank. The row address A0 through A15 is used to determine which row to activate in the selected bank. The Bank Activate command must be applied before any Read or Write operation can be executed. Immediately after the bank active command, the DDR2 SDRAM can accept a read or write command on the following clock cycle. If a Read/Write command is issued to a bank that has not satisfied the t_{RCD}min specification, then additive latency must be programmed into the device to delay when the Read/Write command is internally issued to the device. The additive latency value must be chosen to assure t_{RCD}min is satisfied. Additive latencies of 0, 1, 2, 3, 4 and optionally 5 are supported. Once a bank has been activated it must be precharged before another Bank Activate command can be applied to the same bank. The bank active and precharge times are defined as t_{RAS} and t_{RP}, respectively. The minimum time interval between successive Bank Activate commands to the same bank is determined by the RAS cycle time of the device (t_{RC}). The minimum time interval between Bank Activate commands is t_{RRD}.

In order to ensure that 8 bank devices do not exceed the instantaneous current supplying capability of 4 bank devices, certain restrictions on operation of the 8 bank devices must be observed. There are two rules. One for restricting the number of sequential ACT commands that can be issued and another for allowing more time for RAS precharge for a Precharge All command. The rules are as follows:

- 8 bank device Sequential Bank Activation Restriction: No more than 4 banks may be activated in a rolling t_{FAW} window.

• 8 bank device Precharge All Allowance: t_{RP} for a Precharge All command for an 8 Bank device will equal to t_{RP} +1 x t_{CK}.

3.3 READ and WRITE ACCESS MODES

After a bank has been activated, a read or write cycle can be executed. This is accomplished by setting RAS HIGH, CS and CAS LOW at the clock's rising edge. WE must also be defined at this time to determine whether the access cycle is a read operation (WE HIGH) or a write operation (WE LOW). The DDR2 SDRAM provides a fast column access operation. A single Read or Write Command will initiate a serial read or write operation on successive clock cycles. The boundary of the burst cycle is strictly restricted to specific segments of the page length.

A new burst access must not interrupt the previous 4 bit burst operation in case of BL = 4 setting. However, in case of BL = 8 setting, two cases of interrupt by a new burst access are allowed, one reads interrupted by a read, the other writes interrupted by a write with 4 bit burst boundary respectively. The minimum CAS to CAS delay is defined by t_{CCD}, and is a minimum of 2 clocks for read or write cycles.

DDR2 SDRAM allows a CAS read or writes command to be issued immediately after the RAS bank activate command (or any time during the RAS-CAS-delay time, t_{RCD}, period). The command is held for the time of the Additive Latency (AL) before it is issued inside the device. The Read Latency (RL) is controlled by the sum of AL and the CAS latency (CL). Therefore if a user chooses to issue a Read/Write command before the t_{RCD}min, then AL (greater than 0) must be written into the EMR (1). The Write Latency (WL) is always defined as RL - 1 (read latency -1) where read latency is defined as the sum of additive latency plus CAS latency (RL=AL+CL).

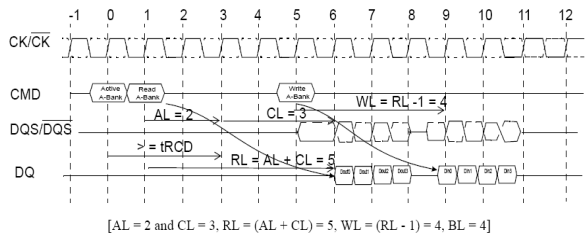


Figure 4: Example of a data transfer in DDR-2

Burst mode operation is used to provide a constant flow of data to memory locations (write cycle), or from memory locations (read cycle). The parameters that define how the burst mode will operate are burst sequence and burst length. DDR2 SDRAM supports 4 bit burst and 8 bit burst modes only. For 8 bit burst mode, full interleave address ordering is supported, however, sequential address ordering is nibble based for ease of implementation. The burst type, either sequential or interleaved, is programmable and defined by MR[A3], which is similar to the DDR SDRAM operation. Seamless burst read or write operations are supported. Unlike DDR devices, interruption of a burst read or writes cycle during BL = 4 mode operations are prohibited. However in case of BL = 8 mode, interruption of a burst read or write operation is limited to two cases, reads interrupted by a read, or writes interrupted by a write. Therefore the Burst Stop command is not supported on DDR2 SDRAM devices.

BURST READ OPERATION

The Burst Read command is initiated by having CS and CAS LOW while holding RAS and WE HIGH at the rising edge of the clock. The address inputs determine the starting column address for the burst. The delay from the start of the command to when the data from the first cell appears on the outputs is equal to the value of the read latency (RL). The data strobe output (DQS) is driven LOW one clock cycle before valid data (DQ) is driven onto the data bus. The first bit of the burst is synchronized with the rising edge of the data strobe (DQS). Each subsequent data-out appears on the DQ pin in phase with the DQS signal in a source synchronous manner. The RL is equal to an additive latency (AL) plus CAS latency (CL). The CL is defined by the Mode Register (MR), similar to the existing SDR and DDR SDRAMs. The AL is defined by the Extended Mode Register (1)(EMR(1)). DDR2 SDRAM pin timings are specified for either single ended mode or differential mode depending on the setting of the EMR "Enable DQS" mode bit; timing advantages of differential mode are realized in system design. The method by which the DDR2 SDRAM pin timings are measured is mode dependent. In single ended mode, timing relationships are measured relative to the rising or falling edges of DQS crossing at VREF. In differential mode, these timing relationships are measured relative to the cross point of DQS and its complement, DQS. This

distinction in timing methods is guaranteed by design and characterization. Note that when differential data strobe mode is disabled via the EMR, the complementary pin, DQS, must be tied externally to V through a 20 Ω to 10 kΩ resistor to ensure proper operation.

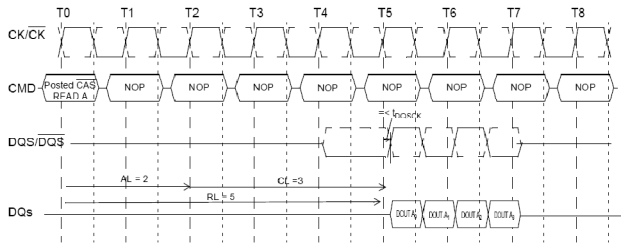


Figure 5: Example of a burst read

The seamless burst read operation is supported by enabling a read command at every other clock for BL = 4 operation, and every 4 clock for BL = 8 operation. This operation is allowed regardless of same or different banks as long as the banks are activated.

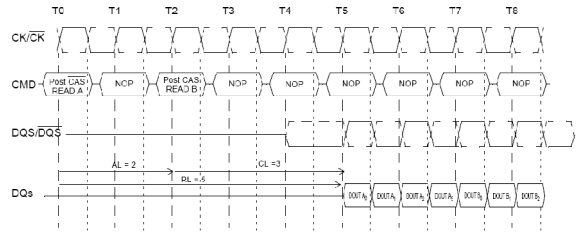


Figure 6: Example of a seamless burst read

BURST WRITE OPERATION

The Burst Write command is initiated by having CS, CAS and WE LOW while holding RAS HIGH at the rising edge of the clock. The address inputs determine the starting column address. Write latency (WL) is defined by a read latency (RL) minus one and is equal to (AL + CL -1); and is the number of clocks of delay that are required from the time the write command is registered to the clock edge associated to the first DQS strobe. A data strobe signal (DQS) should be driven LOW (preamble) nominally half clock prior to the WL. The first data bit of the burst cycle must be applied to the DQ pins at the first rising edge of the DQS following the preamble. The tDQSS specification must be satisfied for each positive DQS transition to its associated clock edge during write cycles. The subsequent burst bit data are issued on successive edges of the DQS until the burst length is completed, which is 4 or 8 bit burst. When the burst has finished, any additional data supplied to the DQ pins will be ignored. The DQ Signal is ignored after the burst write operation is complete. The time from the completion of the burst write to bank precharge is the write recovery time (WR). DDR2 SDRAM pin timings are specified for either single ended mode or differential mode depending on the setting of the EMR "Enable DQS" mode bit; timing advantages of differential mode are realized in system design. The method by which the DDR2 SDRAM pin timings are measured is mode dependent. In single ended mode, timing relationships are measured relative to the rising or falling edges of DQS crossing at the specified AC/DC levels. In differential mode, these timing relationships are measured relative to the cross point of DQS and its complement, DQS. This distinction in timing methods is guaranteed by design and characterization.

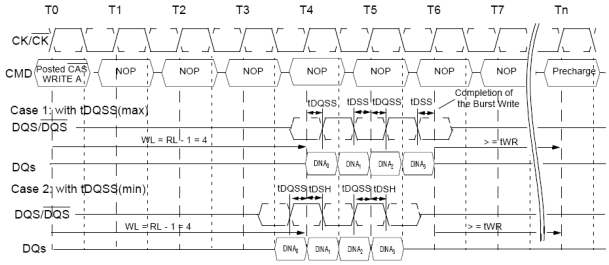


Figure 7: Example of a burst write

The seamless burst write operation is supported by enabling a write command every other clock for BL = 4 operation, every four clocks for BL = 8 operation. This operation is allowed regardless of same or different banks as long as the banks are activated.

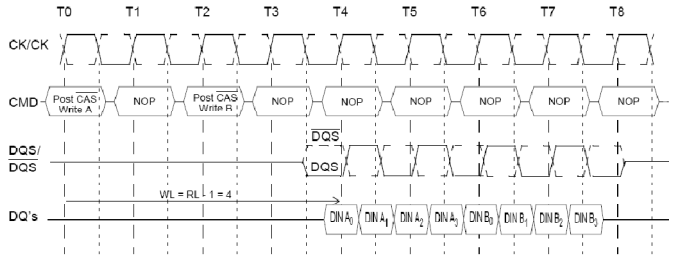


Figure 8: Example of a seamless write burst data-transfer

PRECHARGE

The PRECHARGE command is used to deactivate the open row in a particular bank or the open row in all banks. The bank(s) will be available for a subsequent row activation a specified time (t_{RP}) after the PRECHARGE command is issued, except in the case of concurrent auto precharge, where a READ or WRITE command to a different bank is allowed as long as it does not interrupt the data transfer in the current bank and does not violate any other timing parameters. After a bank has been precharged, it is in the idle state and must be activated prior to any READ or WRITE commands being issued to that bank. A PRECHARGE command is allowed if there is no open row in that bank (idle state) or if the previously open row is already in the process of precharging. However, the precharge period will be determined by the last PRECHARGE command issued to the bank.

From Command	To Command	Minimum Delay between "From Command" to "To Command"
Read	Precharge (to same Bank as Read)	AL + BL/2 + max(RTP, 2) - 2
	Precharge All	AL + BL/2 + max(RTP, 2) - 2
Read w/AP	Precharge (to same Bank as Read w/AP)	AL + BL/2 + max(RTP, 2) - 2
	Precharge All	AL + BL/2 + max(RTP, 2) - 2
Write	Precharge (to same Bank as Write)	WL + BL/2 + tWR
	Precharge All	WL + BL/2 + tWR
Write w/AP	Precharge (to same Bank as Write w/AP)	WL + BL/2 + WR
	Precharge All	WL + BL/2 + WR
Precharge	Precharge (to same Bank as Precharge)	1
	Precharge All	1
Precharge All	Precharge	1
	Precharge All	1

REFRESH

REFRESH is used during normal operation of the DDR2 SDRAM and is analogous to CAS#-before-RAS# (CBR) REFRESH. All banks must be in the idle mode prior to issuing a REFRESH command. This command is non-persistent, so it must be issued each time a refresh is required. The addressing is generated by the internal refresh controller. This makes the address bits a "Don't Care" during a REFRESH command.

SELF REFRESH

The SELF REFRESH command can be used to retain data in the DDR2 SDRAM, even if the rest of the system is powered down. When in the self refresh mode, the DDR2 SDRAM retains data without external clocking. All power supply inputs (including V_{ref}) must be maintained at valid levels upon entry/exit *and* during SELF REFRESH operation.

The SELF REFRESH command is initiated like a REFRESH command except CKE is LOW. The DLL is automatically disabled upon entering self refresh and is automatically enabled upon exiting self refresh.

To summarize DDR-2 RAMs, it would be helpful to look at all the different timing considerations which affect the working of DDR-2 RAMs.

CAS Latency

The CAS latency is the delay, in clock cycles, between sending a READ command and the moment the first piece of data is available on the outputs.

t_{WR} - Write Recovery Time:

t_{WR} is the number of clock cycles taken between writing data and issuing the precharge command. t_{WR} is necessary to guarantee that all data in the write buffer can be safely written to the memory core.

t_{RAS} - Row Active Time:

t_{RAS} is the number of clock cycles taken between a bank active command and issuing the precharge command.

t_{RC} - Row Cycle Time:

The minimum time interval between successive ACTIVE commands to the same bank is defined by t_{RC}.

$t_{RC} = t_{RAS} + t_{RP}$

t_{RCD} - Row Address to Column Address Delay:

t_{RCD} is the number of clock cycles taken between the issuing of the active command and the read/write command. In this time the internal row signal settles enough for the charge sensor to amplify it.

t_{RP} - Row Precharge Time:

t_{RP} is the number of clock cycles taken between the issuing of the precharge command and the active command. In this time the sense amps charge and the bank is activated.

t_{RRD} - Row Active to Row Active Delay:

The minimum time interval between successive ACTIVE commands to the different banks is defined by t_{RRD}.

t_{WTR} - Internal Write to Read Command Delay: t_{WTR} is the delay that has to be inserted after sending the last data from a write operation to the memory and issuing a read command.

To put it all on one piece of paper, the following state diagram can be studied:

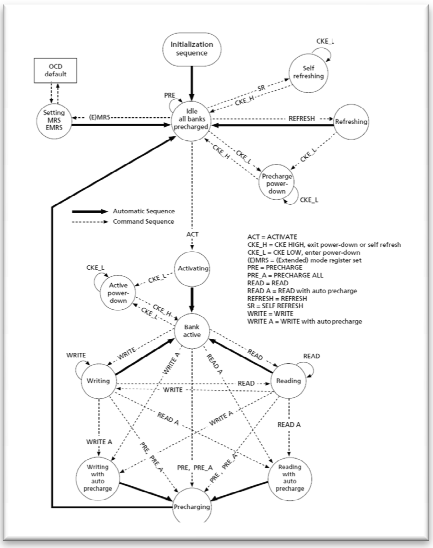


Figure 9: State Diagram of DDR-2

Now after knowing all the details about DDR-2 RAMs we can see the block diagram and figure out that as soon as a bank is Activated and it knows the row and column address, it fetches 4 words per cycle, hence 4n-prefetch architecture, and releases it also at a similar rate in case of the read cycle. So now knowing the basics of DDR-2 RAM let us look at the PRET environment and how this random memory can be used here and made predictable.

4. DDR-2 as MAIN MEMORY for PRET MACHINES

A complete understanding of the working and structure of existing system was necessary in order to use DDr-2, in PRET machines. The DDR-2 is made for modern day PC applications and its sole purpose was to reduce the overhead latency and provide more parallelism for faster and faster use and hence reducing the level-2 or level-3 cache miss penalty. It is made to have more memory at faster speeds ranging from 200MHz - 533MHz.

Now the real challenge is to make this random access memory predictable in time. The need of the hour is to make this RAM behave in a manner which will reduce the parallelism and make time predictability achievable using the DDR-2. This will surely reduce the performance but time predictability has always been PRETS forte.

Now as we have seen that in PRET it would be best if each thread is given an individual bank to use and the DDR-2 has 8 banks; making this a match. Each thread will be given a bank and the rest of the memory will be used as shared space, Monitor program and memory I/O mapping. Our primary focus here is to make each thread work with the main memory predictably within the window of time allotted to it.

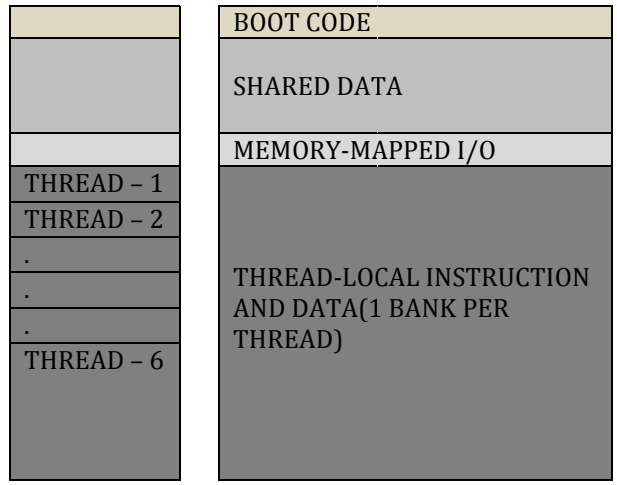


Figure 10: Memory map of DDR-2 for PRET machine

DDR-2 can be used to realize this type of memory. The only matter that needs to be addressed is the address limitations of each thread and also no tolerance for mistaken addressing; it will cause data sharing issues and copies of that data will lie in the whole system causing inconsistency.

Now the most pressing issue is that of accessing the data within the allotted window time, by the wheel. If we try and access data in all slots then we can have the following problems:

1. The activation command comes in the last cycle of the window and the column stays like that till the next time we give the read/ write command. This is not

acceptable as only four banks can be activated at a time, so somewhere down the line one thread will stall for no reason.

2. The bank is transferring data and window ends, this would lead to un-predictability as we will not know how much of it was left and again the same problem of bank activation continues to linger around.
3. If Al !=0 we have issues of seamless transfers, where data of thread one will continue even when it is not suppose to and only then data of thread 2 will start.

These are some of the major problems that could be encountered if normal access were made in each window. So to make it more controlled and predictable a method is devised for using the memory wheel in a controlled environment and using the memory's commands more deterministically.

So the idea is to realize that the following steps happen while a memory is being accessed:

1. ACTIVATION
2. READ/WRITE command
3. PRECHARGE

Now we propose to use auto precharge as it takes care of the precharging the row within the window allotted. During auto-precharge, a Read command will execute as normal with the exception that the active bank will begin to precharge on the rising edge which is CAS latency (CL) clock cycles before the end of the read burst. Auto-precharge is also implemented during Write commands. The precharge operation engaged by the Auto-precharge command will not begin until the last data of the burst write sequence is properly stored in the memory array.

The main idea is to perform each memory access instruction in two window slots. Al is kept at '0' so no seamless transfers can happen. Now consider the following steps:

First cycle:

1. Identify if the thread-1 instructions needs memory access or no. If yes than decide if it's a read or write and calculate its latency, say X. Now calculate Y = No. of clock cycles in a window – X. Now after Y cycles of the clock put the Activate command along with the column address and bank address on the address lines of the ddr-2 chip.
2. In the next cycle, the row address along with the read/write command are put on the bus.

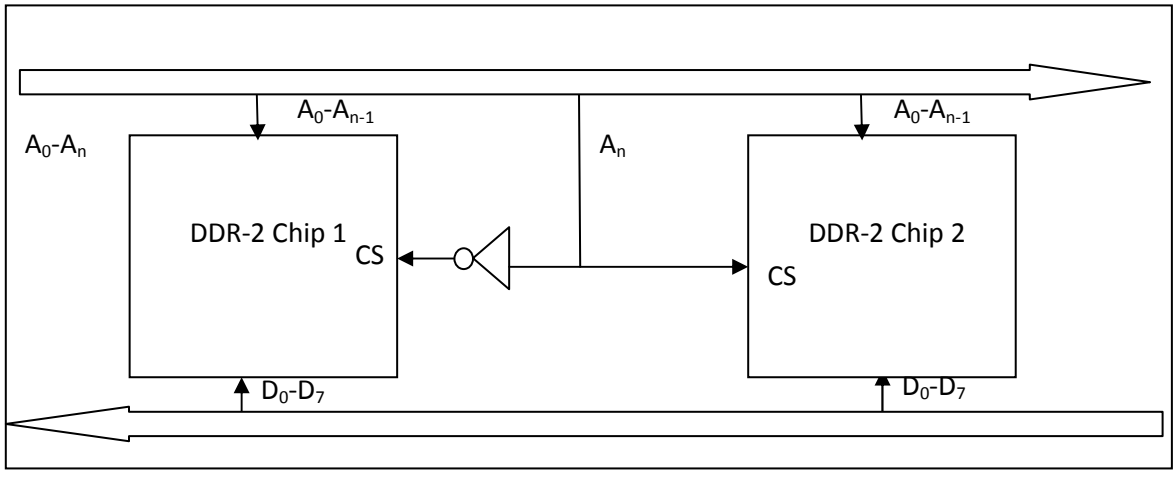
Do the above to steps for every thread.

Second Cycle:

1. Data starts flowing in to data bus for every thread from the first cycle of the window slot.
2. Each bank is precharged and all 4/8 length burst data is transferred before the window ends.

The above happens for every thread which put in a memory access. Thus in the first cycle only the banks are activated and in the second one real data transfer takes place. This effort to un-parallelize the memory operations give predictability to the memory system using DDR-2.

The problem occurs because at a time only 4 banks on the DDR-2 chip can be activated hence using the 8 bank chip will not give us the functionality it was chosen for. The solution of this problem is that we use 2 chips with 4 or 8 banks each and dedicate 1 or 2 banks to each thread. As shown



below, we use the highest address bit to select the correct chip. This means that one of the chips will have memory banks allotted to 2 threads and the other will have banks for the other four threads. The address translation would not be affected and hence this idea seems feasible.

This scheme will give you the predictability required for PRET environment. Also this scheme gives you a data bus of 8-bits only. To get larger data busses, we must use pairs of chip in parallel.

5. INTERPROCESS COMMUNICATION:

Each Thread has its own private context, which includes the register values, PC registers and scratch pad. Even though each thread has its own context, it requires data form threads or tasks or functions and this would than require them to share data among all threads of the system. Now to share data would be simply done if both threads have access to the same variable. This can be easily achieved if the variable is defined in one thread and declare it extern in the other. Now consider the following example to understand sharing data and the problems it can create.

```
struct.
{
   long ltanklevel;
   long ltimeupdates;
} tank[max_tanks];

//button task

void vrespondtobutton(void) //high priority
{
  int i;
  while(true)
    {
       //block untill user pushes button
       i=//id of button pressed;
       printf("\ntime:%08ld  level:508ld",
       tankdata[i].ltimeupdated,
       tankdata[i]/ltanklevel);
    }
}
```

```
//levels task
void vcalculatetanklevels(void) //low priority
{
  int i=0;
  while(true)
  {
     //read levels of floats in tank i
     //do more interminable calculation
     //store the results
     tankdata[i].ltimeupdated = //current time
     //between these two instructions is a bad  place
           to task switch
     tankdata[i].ltanklevel = //result of calculation
     //figure out which tank to do next
      i=//something new
  }

}
```

As we see, that vrespondtobutton task prints out some data that is maintained by the vcalculatetanklevels task. Both tasks can access the tank data array of structs just as they would if this system was free of shared data issues.

In the above example, the RTOS might stop vcalculatetanklevels at any time and run vrespondbutton. Remember, that's what we want the RTOS to do, so as to get good response. However, the RTOS might stop vcalculatetanklevel is right in the middle of setting data in the tankdata array (which is not a atomic operation), and vrespondbutton might then read that half-changed data.

The above code seemed simple enough to run, two tasks sharing data. Unfortunately it is these kinds of programs with shared data issues that blow up space shuttles after their launch. These bugs show up at the wrongest of times, especially on friday evenings, just after your product landed on mars or when no debugging instrument is attached to the system or worst of them all, during a demo. Now, as these bugs often show up very rarely and are therefore difficult to find, it pays to avoid putting these bugs into your code into the first place.

To eliminate the shared data bug, we employ some RTOS techniques, namely:

1. Message Queues
2. Mail Boxes and pipes
3. Semaphores

5.1 MESSAGE QUEUES

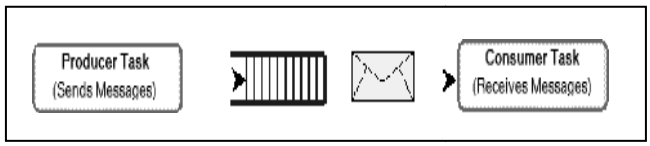


Figure 12: Message Queue

Suppose we have two task, task1 and task2, each of which has a number of high priority, urgent things to do. Suppose also that from time to time each have to report an error condition. Whenever task1 and task2 discover an error, it reports that error to error_task and then continues its process. Now the error reporting process undertaken by error_task does not delay the other tasks.

To implement such a system, a message queue is used. Whenever task1 or task2 reports an error to the error_task it calls a function error_log. This function puts the error on a queue of errors for error task to deal with.

Now two other functions have to be defined, namely add_to_queue and read_from_queue. Each time the system calls read_from_queue, it gets the next error, even if the switch from error_task to taks1 t task1 and again in the middle of the call occurs.

Here is how this can be implemented.

```
void task1()
{
  ...
  if(!!error arises)
  error_log(error_type1);

  //other high priority tasks
  ....
}

void task2()
{
  ...
  if(!!error arises)
  error_log(error_type2);

  //other high priority tasks
  ...
  }
```

```
 void error_log()
{
  add_to_queue(int error_type);
}

void error_task()
{
  int error_types;
  while(1)
    {
       read_from_queue(&error_type);
       error_queue_no++;
      //send error_queue_no and
error_type
     }
```

This approach is primitive and not implemented in RTOSs anymore. In our error case the message is a constant sized message but that may not be the case in all queues. Hence the time it takes to transfer this message depends on the size of the message, this increases non-determinism in the system making it impossible to predict.

An approach that avoids this non-determinism and also accelerates performance, is to have the operating system copy a pointer to the message and deliver that pointer to the message-receiver task without moving the message contents at all.

5.2.1 MAILBOXES:

Mailboxes are very similar to queues. The system function to write, read and delete them exists along with a function that checks whether the mailbox is empty or no in order to destroy it and free the space. The number of message that can be store in a mailbox can be chosen by the user while creating it. We can also have a mailbox with unlimited space. Sometimes in a mailbox we can prioritize the messages. In this case the higher-priority messages will be read before the low priority one, regardless of the order in which they are written into the mailbox. This would be of help in the error logging example. A more serious error should be serviced before the others.

5.2.2 PIPES

Pipes are much like queues, we can write to them, read from them and so on. Some pipes allow variable length messages unlike the message queues and mailboxes. The pipe can be completely byte oriented. A thread can write any number of bytes and another can pick up the number bytes it requires from the pipe.

The rest id left on the pipe for some other pipe or program to read later.

5.2.2 PTIFALLS

Above described queues, pipes and mailboxes seem to be quite easy to share data but are prone to introducing bugs into the system. Here are a few types of bugs they can introduce:

1. There is usually no restriction on who reads or writes to these structures, so it is the users responsibility to ensure thread uses the correct one message each time.
2. A message which is 'int' type can be treated as a 'char' or a pointer and hence the bug will corrupt the system even without us noticing it.
3. Running out of space is also a possibility.
4. Message passing through all these structures is not a bad option but it causes a lot of 'wait' time which reduces performance and also is non deterministic as each thread will have data in its queue which could be wanted by another thread and will have wait indefinitely till the producer thread passes it.

5.3 SEMAPHORES

In a multitasking environment there is often a requirement to synchronize the execution of various tasks or ensure one process has been completed before another begins. This requirement is facilitated by the use of a software switch known as a Semaphore or a Flag. The function of this is to work in much the same way a railway signal would; only allowing one train on the track at a time.

A semaphore object is also a synchronization object that maintains a count between zero and a specified maximum value. The count is decremented each time a thread completes a wait for the semaphore object and incremented each time a thread releases the semaphore. When the count reaches zero, no more threads have to wait to be signaled or wait for the semaphore. The state of a semaphore is set to signaled when its count is greater than zero, and non-signaled when its count is zero.

The semaphore object is useful in controlling a shared resource that can support a limited number of users. It acts as a gate that limits the number of threads sharing the resource to a specified maximum number. For example, an application might place a limit on the number of windows that it creates. It uses a semaphore with a maximum count equal to the window limit, decrementing the count whenever a window is created and incrementing it whenever a window is closed. The application specifies the semaphore object in call to one of the wait function before each window is created. When the count is zero — indicating that the window limit has been reached — the wait function blocks execution of the window-creation code.

There are many variants of semaphores. Let's discuss the binary semaphore and then the other ones like Mutex or a counting semaphore or resource semaphore.

Each task can call two RTOS functions, TakeSemaphore and ReleaseSemaphore. When task A has called TakeSemaphore and has not called ReleaseSemaphore, task b which calls TakeSemaphore is asked to wait. Task B is blocked till task A calls ReleaseSemaphore. Only one task can have the semaphore at a time. Now let's use these to RTOS functions to solve the shared data problem in the tank program.

In the above case, updating or using the shared data is the critical region and is protected by sempahore. In the above way every critical section can have a semaphore haence there can be multiple semaphores. Release of a semaphore by taskA will not affect taskB if A and B are independent and using different semaphores.

```
//levels task
void vcalculatetanklevels(void) //low priority
{
  int i=0;
  while(1)
   {
      //read levels of floats in tank i
      //do more interminable calculation
      //store the results
      TakeSemaphore()
      tankdata[i].ltimeupdated = //current time
       //between these two instructions is a bad
place to  task  switch
       tankdata[i].ltanklevel = //result of calculation
       ReleaseSemaphore();
      //figure out which tank to do next
      i=//something new
   }
}
```

```
struct.
{
  long ltanklevel;
  long ltimeupdates;
} tank[max_tanks];

//button task
void vrespondtobutton(void) //high priority
{
  …
  if(!!error arises)
  error_log(error_type2);

  //other high priority tasks
  …
  }

int i;
  while(true)
   {
      //block untill user pushes button
      i=//id of button pressed;
      TakeSemaphore()
      printf("\ntime:%08ld  level:508ld",
      tankdata[i].ltimeupdated,
      tankdata[i]/ltanklevel);
     ReleaseSemaphore();
   }
```

Some systems offer semaphores that can be taken multiple times. Essentially, such semaphores are integers; taking them decrements the integer and releasing them increments the integer. If a task tries to take the semaphore when the integer is equal to zero, then the task will block. These semaphores are called counting semaphores, and they were the original type of semaphores.

Now the thought that semaphore is the best solution for shared data problem is not entirely true as semaphore brings along inherent problems of its own. Thus to design a very robust system, less number of semaphore should be used. Semaphores works well only if used carefully and wisely. The number ways a semaphore won't work are:

1. Forgetting to take the semaphore
2. Forgetting to release the semaphore
3. Taking the wrong semaphore
4. Holding the semaphore for too long
5. Causing a deadlock
6. Unbound Priority Inversions

The semaphore variants that are most interesting are mutexes and counting semaphores.

5.4 MUTEX

A ‘mutex’ is a special kind of semaphore used particularly to provide mutual exclusion, hence its name. The operations on a mutex are called ‘lock’ and ‘unlock’, with meanings which are pretty much intuitive. It can eliminate the unbound priority inversion problem.

A mutex can be thought of as being in either a locked or an unlocked state. A task can use the lock operation to take the mutex from the unlocked to the locked state; and if it succeeds in this operation, the task becomes the “owner” of the mutex while it remains locked. If the mutex is already in the locked state, a task trying to use the lock operation will be prevented from running, until the mutex is unlocked and becomes available. Even after it becomes available, the RTOS decides the next owner of the MUTEX. Unlike traditional semaphores, mutexes should not be thought of as having a count. Instead, they’re best thought of as being in either a locked or an unlocked state.

Only the task that “owns” the mutex at a given time, can perform the ‘unlock’ operation on the mutex. This will take the mutex from the locked to the unlocked state. This will also end the “ownership” of the mutex by that task.

The concept of a ’token’ is also not relevant for mutexes. With traditional semaphores, the 'take' operation can be thought of as “taking” or “getting” a token, which can then be passed from

task to task if necessary. With a mutex, the 'lock' operation can be thought of as obtaining "ownership" of the mutex. But the "owner" task is not permitted to pass the "ownership" directly to another task.

Probably the most significant difference between traditional semaphores and mutexes, is that the problem of unbounded priority inversion can not easily be solved using traditional semaphores. This is because of their lack of a notion of "ownership" in a traditional semaphore.

When tasks share resources, as they often do/must, strange things can and will happen. Priority inversions can be particularly difficult to anticipate. A basic understanding of the problem is the key. All the various techniques or tools discussed related to semaphores are used to share data through a shared address space either in the common memory or as a part of the thread's personal address space.

All the above described methods for resolving shared memory problems and enable interprocess communication in PRET environment will not be sufficient. Each method described has some kind of non-determinism embedded in it. Like the mutex or the semaphore, otherwise most widely used methods, are inherently non-deterministic. So it cannot be used in PRET machines. The message passing through queues, mailboxes or pipes requires a lot of RTOS support, and in a system where the attempt is to keep the work of RTOS as minimal as possible they are not a good idea. These methods when used also require some memory and all that put with the parallel behavior of DDR-2 RAMs will be a very difficult process.

6. DIRECTORY BASED IPC:

In the PRET machines we surely know the occurrence of memory accesses of each thread and we also know the time when it is going to occur. This is possible because of the instructions like deadline and replay present in the PRET environment.

Although, the standard IPC methods don't give us a good solution, we can try and fill the non-deterministic holes and innovate to find a solution that can be used in the PRET environment. Using those concepts and the deterministic behavior of the PRET machine we can consider the following as a possible solution to IPC problem in PRET machines.

Consider a directory based system which keeps a record of which memory bank/location is being used by each thread. The directory will have a special type of counter, which will keep a count of the time for which the particular thread will be using the current memory location/bank. Now this counter will be a decrement counter and will be loaded by the time-count for which the thread will be using a particular memory/bank for. There will one such counter for every thread. The directory would be as shown below:

Memory Address	Thread #	Semaphore Count	Counter
	1		
	2		
	3		
	4		
	5		
	6		

Figure 13: Directory for IPC

Now every time a thread wants to access the memory, it searches through the small directory and the hit or miss condition would be known to the user beforehand due to the deterministic nature of PRET. Now if it is a miss, then normal memory access takes place. But in case of a hit, the thread cannot access the memory and has to use the replay instruction for adding NOPs to avoid pipeline stall. It then loads the count form thread using the memory onto its own counter. As soon as its counter becomes zero, it executed its own access. This now provides a safe and precise IPC.

In the case when more than one thread have the need to access the same shared memory space, each successive thread would increase the semaphore counter number and add all the timings of the executing and waiting threads to decide when it needs to wait till. In the worst case scenario, all six threads will want access. The

process will be slow, performance will take a hit, but determinism in IPC will surely be achieved.

To understand this better consider the following example: Thread1 is accessing 'xyz' part of the memory. It will require a total sum of 90cycles to complete the transaction. Now say another thread2 wants to access the same location and will take 180 cycles to complete its transaction. So it first waits for 90 cycles, when its counter becomes zero and starts its own access.

Now, consider a thread3 puts in a request to use the same location when 35 cycles of thread1 were remaining. In this case the counter of thread3 would be the addition of the count of the counter of thread1 and thread2. Thus thread3 would have to wait for 35+180 = 215 cycles before it begins its own instruction.

All shared data access will be done directly to the main memory to avoid data inconsistency. The process of checking the directory and waiting in the directory causes some overhead. This overhead is a small price we pay for Time Predictability which is the focus of PRET machines.

7. FUTURE WORK

In the future, we would like to design a Programmable Memory Wheel. This would allow us to have a very precisely timed program and also resolve all our memory related issues. The programmable memory would be based on the Timed Triggered Protocol.

The programmable memory wheel will allow each thread to request for banks and/or bandwidth. This would be possible only statically, i.e. only during the initialization process. If threads make conflicting requests then the RTOS present on one of the threads, will throw an exception. Also interprocess communication is possible if during the initialization process along with allocation for each thread, RTOS can also allow two or more threads to access one bank depending on the available bandwidth. This whole concept is very important for solving the memory issues with timing predictability.

Also so far in PRET machine we have only spoken of a single-core, but designing memory system for multi-core PRET machine would be more challenging with more issues like inter-core communications and long latencies due structural hazards.

8. CONCLUSION

The paper addresses different memory models and in specific DDR-2's compatibility with PRET machines From the work done thus far various decisive conclusions have been made. A clear understanding has reached of the implications thrown by DDR-2 memory and ways to resolve are being worked upon.

Also to resolve the problems introduced by inter-process communication, the existing memory wheel structures in the PRET machines also have to be changed to suit the needs of both the processor and the DDR-2 memory chips.

9. REFERNCES

[1] B. Lickly, I. Liu, S. Kim, H. Patel, S. Edwards and E. Lee. Predictable Programming on a Precision Timed Architecture. In proceedings for Conference on Compliers, Architecture and Synthesis of Embedded Systems (CASES '08), Atlanta, Georgia, USA, October , 2008.

[2] JESD79-2E, DDR-2 SDRAM Specification, JEDEC Standard, JEDEC Technology Solid State Association, April 2008.

[3] O.Ozturk, M Kandemir and I. Kolcu. Shared Scratch-Pad Memory Space Management.

[4] J Leverich, H. Arakido, A. Solomatnikov, A Firoozshahian, M. Horowitz and C. Kozyrakis. Comparing Memory Systems for Chip Multiprocessors.

[5] B. Jacobs. Cache Design for Embedded Real-Time Systems.

[6] D. Kalinsky. Basics of Real Time Operating Systems. Nov. 2003.

[7] D. Kalinsky, Mutexes Battle Priority Inversions.

[8] D. Simon. An Embedded Software Primer. 2nd Edition, 12th Indian Reprint.

[9] Christ of Pitter and M. Schoeberl. Time Predictable CPU and DMA Shared Memory Access.