The Design of a Hardware Memory Allocator Based on the Buddy System

Manyip Ngan
Edward F. Gehringer

July 1995

Technical report 95-10,
Department of Computer Science,
North Carolina State University
CONTENTS

PART A
1. Introduction 1
2. System Design 1
   2.1 Bit-Vector 2
   2.2 Memory Allocation 3
3. System Description 4
4. Conclusion 8

PART B
1. System Description 9
   1.1 General Functional Description 9
   1.2 Pin Description 9
   1.3 Simulation Block Diagram 9
2. Functional Block Description 10
   2.1 I/O Latches 10
   2.2 Rounded_up 10
   2.3 CBT 11
      2.3.1 Allocation/free Scheme 12
         2.3.1.1 Allocation Scheme 12
         2.3.1.2 Deallocation Scheme 17
      2.3.2 CBT Block Diagram 19
      2.3.3 Pin Description 20
   2.4 Selector 20
   2.5 MEM 22
   2.6 2-1 MUX 24
   2.7 4-16 Decoder 24
   2.8 16-4 Encoder 24
   2.9 Error Detection 24
3. System Data Flow 25
   3.1 Allocation Operation 25
      3.1.1 Functional Timing 25
      3.1.2 Data Flow 26
   3.2 Deallocation Operation 28
      3.2.1 Functional Timing 28
      3.2.2 Data Flow 29
4. Circuit Description 31
   4.1 CBT 31
      4.1.1 Generation of Availability 32
      4.1.2 Find the first ABRS (allocation block of requested size) 32
      4.1.3 Generation of flip bit signal 36


4.2 Other Blocks
   4.2.1 MEM 43
   4.2.2 Selector 44
   4.2.3 Rounded_up 47
   4.2.4 2-1 MUX 48
   4.2.5 16-4 Encoder 48
   4.2.6 4-16 Decoder 48
   4.2.7 Latches 48

5. Simulation Result 49
   5.1 Functional Timing 49
      5.1.1 Allocation/allocation Cycles 49
      5.1.2 Deallocation/deallocation Cycles 50
      5.1.3 Allocation/deallocaton/allocaion Cycles 50
      5.1.4 Allocation/nop/allocation Cycles 51
      5.1.5 Deallocation/nop/deallocation Cycles 51
   5.2 Gate Delay and Simulation Pins 51
      5.2.1 Gate Delay 52
      5.2.2 Simulation Pins 52
   5.3 Test Patterns and Simulation Waveforms 53
      5.3.1 Allocation Test 54
      5.3.2 Deallocaton Test 55
      5.3.3 Mixed Test 56
   5.4 Discussion of AC timing 56
   5.5 Discussion 58

Circuits 59
Simulation Waveforms 77
END 91
PART A

1. Introduction

Dynamic memory management has been an important topic in computer systems for many years. Much research on has been carried out on high-performance algorithms. Most of this research focuses on improving memory-management performance through software implementation.

The paper “A high-performance memory allocator based on the buddy system,” written by J. Morris Chang and Edward F. Gehringer, proposes a novel hardware implementation of the buddy system. This implementation uses bit-map tables to represent the allocation status of cache lines, and is capable of allocating one of these lines in a single cycle.

Using Viewlogic simulation tools, we designed the circuits for this hardware implementation. Our work includes architecture design, circuit design, debugging, and testing. In this report, we present the design concept of this system. The remainder of this report is organized as follows. Section 2 presents the basic idea of system design and section 3 presents the functional timing of allocation and deallocation examples. A brief conclusion will come after section 3. A detailed discussion of the system, its circuits, and simulation results will be covered in Part B.

2. System Design

Object-oriented systems create objects very frequently. Each object occupies a certain amount of memory. In an object-oriented system, memory allocation thus occurs very frequently. Most objects, however, do not “live” very long. That is, after a short period of time, most objects are no longer pointed to by a pointer anywhere in memory. They have effectively been abandoned, and the memory manager may reclaim them.

The memory allocator described in this document implements memory allocation and deallocation. This section addresses its design.

2.1. The bit-vector

In software allocation schemes, free memory blocks are often kept on a list. Each time memory allocation is requested, this list is searched to find an unused block large enough to allocate from. Though a list is an efficient structure to search by software, it is not very amenable to hardware manipulation. For searching by hardware, a bit-vector is a more appropriate structure.
Our memory allocator uses a bit-vector to represent the allocation status of each cache-line-size block of virtual memory. A bit has the value 1 if the corresponding block is allocated, and 0 if the corresponding block is free.

In order to avoid unnecessary memory reads, all blocks will be allocated in cache. That is, when an allocation occurs, the memory at the allocation address is not read into the cache; rather the target cache line is cleared. When an object dies in cache, the dead object is not written back to memory; its cache line(s) are simply marked as free. A simple example of using a bit-vector to manage $2^{11} = 2048$ blocks is shown in the figure below.

![Bit-vector Diagram](image)

* assume each block is 32 bytes

A program’s memory may consist of a very large number of cache-sized blocks. A very large bit-vector would be needed to represent the allocation status of each block. Since it is not very efficient to manipulate large bit-vectors in hardware, the virtual address space is divided into regions, each of which has its separate (smaller) bit-vector.

The bit-vector for a region is held in a series of registers. Each register contains a portion of the bit-vector that represents the allocation status of a particular page of memory. The reason for dividing the bit-vector into page-sized chunks is that most caches are accessed by physical address, while programs reference virtual addresses. A bit-vector represents the allocation status of a region of virtual memory, which is (usually) not contiguous in physical memory. So each page-sized bit-vector may refer to an area of physical memory which is discontiguous from the physical memory of its virtual neighbors. The diagram below shows how this works. In the diagram, each page contains $2^7 = 128$ cache-sized blocks. Thus, each $2^{11}$ region of virtual memory is represented by a set of 16 bit-vectors of size $2^7$. 
Two major functions of the memory allocator are memory allocation and deallocation. The following sections will describe how our bit-vectors can be used for that purpose.

2.2. Memory allocation

Allocating free memory space to a new object requires searching for an appropriate number of free bits in a bit-vector. In our system, we divide the bit-vector of a virtual-memory region into 16 bit-vectors, each representing one page of memory. The first step is a best-fit search to locate a page that has a free block large enough to allocate the requested block. Associated with each page in the region, there is an availability vector indicating the size of the largest free block in that page. The page is selected and its bit-vector is read out for use in the second step.

When the bit-vector is read out, it is passed through a set of logic gates that construct a complete binary tree (CBT). When a block of the requested size is located using the CBT, the allocation address is output and the bits corresponding to the allocated block are flipped. After the bit-vector is updated, the availability-vector field for this page is updated from the CBT. The figure below shows the hardware for finding a free block.
3. System Description

The memory allocator allocates or deallocates memory in one single clock cycle. In allocation operation, one can create an object by specifying the requested memory size (object size) and setting the operation mode to allocation. The external pins for user to input these two signals are:

- \( B[7:0] \) — memory size (for both allocation and deallocation operation)
- ALLOC — operation mode switch (ALLOC = '1' is allocation mode and ALLOC = '0' is deallocation mode)

When the system receives an allocation request and has enough free space to allocate the memory, it will send out the starting address of the allocated memory block without signaling an error (ERROR = '1'). If the allocation is not successful, the system will return an invalid address while signaling an error (ERROR = '0'). The external pins for outputting these two signals are:

- ADDR_OUT[10:0] — starting address of the allocated memory
- ERROR — error pin (ERROR = '0' when an error occurs, else ERROR = '1')
For example, if user creates an object of size = 7 lines, he needs to:

1. set B[7:0] = 7 = 07 in hexadecimal
2. set ALLOC = 1

If the allocation process is successful, the system will return the starting address of the allocated memory and set ERROR pin to '1'. In this example, we assume the starting address is 018 in hexadecimal. CLK1 and CLK2 are two non-overlapping clock signals used to synchronize the internal circuits. The functional timing of this allocation cycle is shown below:

![Allocation Cycle Diagram]

In this allocation cycle, the requested memory size (B[7:0] = 07) and operation mode (ALLOC = 1) are input at the rising edge of CLK1. The system outputs the starting address of newly allocated memory (ADDR_OUT[10:0] = 018) without error (ERROR = 1) at the rising edge of CLK2 in the same cycle.

In deallocation operation, one can free an object by specifying the deallocation memory size (size of object), the address of the memory, and the operation mode at the beginning of the cycle. The external pins for these three signals are B[7:0], ADDR_IN[10:0], and ALLOC. At the end of the cycle, the system will set ERROR to '1' when operation is successful and set ERROR to '0' when the deallocation fails.

For example, if user frees an object located at address 010 and had a size of 5 blocks, he needs to:

1. set B[7:0] = 5 = 05 in hexadecimal
2. set ADDR_IN[10:0] = 010
3. set ALLOC = 0
The functional timing for this example is shown below:

```
CLK1
CLK2
ALLOC '0' means deallocation
B[7:0]
ERROR '1' means no error

The system will return ERROR = '1' at the rising edge of CLK2 in the same cycle when the deallocation is successful.

All external pins are listed below:

<table>
<thead>
<tr>
<th>Pin name</th>
<th>In/output</th>
<th>Function</th>
<th>Used in which mode?</th>
</tr>
</thead>
<tbody>
<tr>
<td>B[7:0]</td>
<td>input</td>
<td>memory size</td>
<td>alloc. and dealloc.</td>
</tr>
<tr>
<td>ALLOC</td>
<td>input</td>
<td>operation mode switch: ALLOC = '1' for allocation, ALLOC = '0' for deallocation</td>
<td>alloc. and dealloc.</td>
</tr>
<tr>
<td>ADDR_IN[10:0]</td>
<td>input</td>
<td>starting address of memory requested to be freed</td>
<td>deallocation only</td>
</tr>
<tr>
<td>ADDR_OUT[10:0]</td>
<td>output</td>
<td>starting address of memory requested to be allocated</td>
<td>allocation only</td>
</tr>
<tr>
<td>ERROR</td>
<td>output</td>
<td>error signal: ERROR = '1' when no error exists, ERROR = '0' when error exists</td>
<td>alloc. and dealloc.</td>
</tr>
</tbody>
</table>
```

During development, three test patterns are used for the AC timing test. The system works well within a 75ns clock rate. The testing includes:

- Allocation / Allocation cycles
- Deallocation / Deallocation cycles
- Allocation / Deallocation / Allocation cycles
- Allocation / Nop / Allocation cycles
- Deallocation / Nop / Deallocation cycles

Note: Nop is the “no-operation cycle”
The simulation is done in the Viewlogic environment with the gate delay set to the following value:

<table>
<thead>
<tr>
<th>Gate Type</th>
<th>TPLH, TPHL</th>
<th>TPLZ, TPHZ, TPZL, TPZH</th>
<th>CLKtoQ0, CLKtoQ1, CLKtoQN0, CLKtoQN1</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>0.5 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BUF</td>
<td>1.0 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>TRIBUF</td>
<td>1.2 ns</td>
<td>1.2 ns</td>
<td>—</td>
</tr>
<tr>
<td>NAND</td>
<td>0.8 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>0.8 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>1.0 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>NOR</td>
<td>0.8 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>0.8 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>1.0 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>AND</td>
<td>0.8 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>0.8 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>1.0 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>OR</td>
<td>0.8 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>0.8 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>1.0 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>LATCH</td>
<td>—</td>
<td>—</td>
<td>7.5 ns</td>
</tr>
<tr>
<td>T_F/F</td>
<td>—</td>
<td>—</td>
<td>3.5 ns</td>
</tr>
<tr>
<td>D_F/F</td>
<td>—</td>
<td>—</td>
<td>7.5 ns</td>
</tr>
</tbody>
</table>

Legend:
- TPLH — low to high propagation time
- TPHL — high to low propagation time
- TPLZ — low to high-impedence propagation time
- TPHZ — high to high-impedence propagation time
- TPZL — high-impedence to low propagation time
- TPZH — high-impedence to high propagation time
- CLKtoQ0 — delay time from CLK trigger to Q output = '0'
- CLKtoQ1 — delay time from CLK trigger to Q output = '1'
- CLKtoQN0 — delay time from CLK trigger to Q' output = '0'
- CLKtoQN1 — delay time from CLK trigger to Q' output = '1'
4. Conclusion

The system works well with a clock cycle of 75 ns. It cannot work faster than this speed because the whole architecture is designed to generate the output within one clock cycle. There are two possible ways to improve our system speed.

1. Use a pipelined architecture to separate the system into several smaller segments. This allows several allocation or deallocation requests to be handled at the same time and increases the throughput.

2. Reduce the delay time by optimizing the circuits.

As we are entering the world of object-oriented programming, further hardware design should be done to enhance the speed of creating or destroying objects. This is an interesting research topic with large potential market. Our design is a good attempt to start this trend.
PART B

1. System Description

1.1 General Function Description

The memory allocator is a hardware implementation based on the paper “A high-performance memory allocator based on the buddy system.” Information about the allocation status of memory blocks is held in a binary tree structure formed by AND/OR/FLIPPER cells. These cells propagate information about allocation status and realize the allocation and deallocation operations. The system can allocate or deallocate an arbitrary-sized block in a single clock cycle. The bit-map for allocating memory is maintained by a set of 16 bit-vectors, each of which contains 128 mapping bits.

1.2 Pin Description

The pin functions are listed in table 1.1.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK1</td>
<td>Clock 1 — Cooperates with CLK2 to provide two nonoverlapped clock signals to the internal circuits.</td>
</tr>
<tr>
<td>CLK2</td>
<td>Clock 2 — Cooperates with CLK1 to provide two nonoverlapped clock signals to the internal circuits.</td>
</tr>
<tr>
<td>CLR</td>
<td>Clear pin — Clears the previous circuit’s status. Reset the circuits to the initial state.</td>
</tr>
<tr>
<td>ALLOC</td>
<td>Operation mode switch — Set system to allocation mode when ALLOC = '1'. Set system to deallocation mode when ALLOC = '0'.</td>
</tr>
<tr>
<td>B[7:0]</td>
<td>Memory size — Specify the number of memory blocks needed to be allocated or deallocated.</td>
</tr>
<tr>
<td>ADDR_IN[10:0]</td>
<td>Input address — Specify the starting address of the deallocated memory blocks.</td>
</tr>
<tr>
<td>ADDR_OUT[10:0]</td>
<td>Output address — Output the starting address of the allocated memory blocks.</td>
</tr>
<tr>
<td>ERROR</td>
<td>Error pin — When the requested value (B[7:0]) is larger than 128 or the system has not enough memory to be allocated, the ERROR pin will be set 'L'. In normal operation, it is always set to 'H'.</td>
</tr>
</tbody>
</table>

1.3 Block Diagram of circuits

The block diagram is shown on Fig C1 on page 59.
2. Functional Block Description

The whole system is composed of the following functional blocks.

2.1 I/O Latches

I/O latches synchronize the input and output signal/data with the CLK1 and CLK2 signals. There are five I/O latches in all, which are classified as input and output latches:

The input latches are all triggered by the rising edge of CLK1. They include:

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LATCH_A</td>
<td>Latches in the memory size from the external signal B[7:0].</td>
</tr>
<tr>
<td>LATCH_B</td>
<td>Latches in the starting address of the memory to be freed from the external signal ADDR_IN[10:0].</td>
</tr>
<tr>
<td>LATCH_C</td>
<td>Latches in the operation mode signal from external pin ALLOC.</td>
</tr>
</tbody>
</table>

The output latches are all triggered by the rising edge of CLK2. They include:

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LATCH_D</td>
<td>latches out the starting address of allocation memory to external pins ADDR_IN[10:0].</td>
</tr>
<tr>
<td>LATCH_E</td>
<td>latches out the error signal to external pin ERROR.</td>
</tr>
</tbody>
</table>

2.2 ROUND_UP

Block Diagram:
The size of the requested block $B$ is rounded_up to generate the internal signal $B_{\text{rounded up}}$. $B_{\text{rounded up}}$ is defined as:

$$B_{\text{rounded up}} = 2^{\left\lfloor \log_2 B \right\rfloor}$$

If the size of the requested block is so large that all 128 of the bits need to be flipped, the FLIP_ALL signal will be asserted. Here are some examples of $B_{\text{rounded up}}$, FLIP_ALL, and ERROR generated from a requested block size of 8 bits:

<table>
<thead>
<tr>
<th>$B_{[7:0]}$</th>
<th>$\left\lfloor \log_2 B \right\rfloor$</th>
<th>$2^{\left\lfloor \log_2 B \right\rfloor}$</th>
<th>FLIP_ALL</th>
<th>ERROR (low-active)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000100</td>
<td>2</td>
<td>00000100 ($2^2$)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00010000</td>
<td>4</td>
<td>00010000 ($2^4$)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01000000</td>
<td>6</td>
<td>01000000 ($2^6$)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00000011</td>
<td>2</td>
<td>00000100 ($2^2$)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00001010</td>
<td>4</td>
<td>00010000 ($2^4$)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00010010</td>
<td>5</td>
<td>00100000 ($2^5$)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00101101</td>
<td>6</td>
<td>01000000 ($2^6$)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10000000</td>
<td>7</td>
<td>10000000 ($2^7$)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01000001</td>
<td>7</td>
<td>10000000 ($2^7$)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10000010</td>
<td>exceeds limit</td>
<td>not valid</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The maximum number of blocks requested at a time is 128, which is the total number of mapping bits in each bit-vector. If the requested block size is larger than 128, the system cannot handle this request, and ERROR will be set to '0'. The $B_{\text{rounded up}}$ is not valid when ERROR is '0'. Otherwise, if the requested block size is exactly equal to 128, the FLIP_ALL pin will become '1' to inform the CBT to allocate all 128 mapping bits in the selected bit-vector.

The $B_{\text{rounded up}}$ is used by SELECTOR block to select the appropriate bit-vector and also by the CBT to allocate/deallocate the requested mapping bits.

### 2.3 CBT

The CBT (Complete Binary Tree) is a combinational logic circuits capable of allocating and deallocating the requested mapping bits on the selected bit-vector. It is composed of or-tree, and-tree, and flipper-tree.
2.3.1 Allocation/free scheme

The CBT can allocate arbitrary requested block size when it has located sufficient free mapping bits in the bit-vector under the following schemes:

2.3.1.1 Allocation scheme:

1. We need to find an allocation block of the requested size (ABRS). When the requested allocation block size = B, the size of ABRS is:
\[
\text{size of ABRS} = 2^{\lceil \log B \rceil} = 2^n
\]

2. In the ABRS, the \(2^n\) mapping bits must be contiguous and share the common level-\(n\) node on the complete binary tree (CBT) structure.

For example, for a 8-mapping-bit CBT.

The mapping bits are the nodes on level 0. A level-1 node has two level-0 children. We say the two children share the same level-1 node. Similarly a level-2 node has four level-0 children. Therefore, these four children are sharing the same level-2 node.

Here are the ways for locating ABRS in the mapping bits of size \(2^n\).

For \(n = 0\), the mapping bits found must share the same level-0 node. Each node and its children form an ABRS. As there are eight level-0 nodes, there are eight possible ABRSs.

For \(n = 1\), there are four level-1 nodes, so there are four possible ABRSs.
For $n = 2$, there are two level-2 nodes, so there are two possible ABRSs.

For $n = 3$, there is one level-3 node, so there is only one possible ABRS.

If more than one possible ABRS exists, the CBT will choose the one with the lowest starting address, which is the first one on the left in the diagram.

To illustrate the allocation scheme, we use an allocation example on a 8-bit bit-vector. Assume the user allocates memory five times and the requested block size are 1, 2, 3, 1 respectively. Assume all bits (mapping bits) on the bit-vector are ‘0’ (free) initially.

Allocate a memory block of size 1.

As the requested size = 1 = 2$^0$, the size of ABRS is also 2$^0$. There are eight possible ABRSs; the CBT will choose the first one on the left.
Allocate memory of size 2.

As the requested size = 2 = 2\(^1\), the size of ABRS is also 2\(^1\). There are 3 possible ABRSs. The CBT will choose the first one on the left.

Allocate memory of size 3.

As the requested size = 3, the size of ABRS is also 2\(^2\). There is only one possible ABRS, and the CBT must use this one.
Allocate memory of size 1.

As the requested size $= 1 = 2^0$, the size of ABRS is also $2^0$. There are 2 possible ABRSs, and the CBT will choose the first one on the left.

After the allocations, the mapping bits in the bit-vector change from 00000000 to 11111110.

There will be an error if there is no possible ABRS found for an allocation request. Therefore, ROUND_UP and SELECTOR blocks will check if a possible ABRS exists or not. If at least one exists, the CBT will be enabled (ENABLE pin is set to ‘1’). If no possible ABRS is found, CBT is disabled (ENABLE pin is set to ‘0’) and no mapping bit will be affected.

In a successful allocation, the starting address of the allocated blocks is output. It is actually the starting address of the ABRS that is chosen.
2.3.2. Deallocation scheme

For deallocation operation, the user must supply the size and the starting address of the memory blocks he wants to free. The CBT will free the blocks that are allocated through the allocation scheme discussed in the previous part. This means that the address supplied must be the starting address of a possible ABRS for that size. Otherwise, an error may occur.

For example, for an 8-bit bit-vector:

Free 1 memory block at address $a_3$ (assume the initial bit-vector is 11111111).

For size = 1, there are eight possible ABRSs. The starting address for these ABRSs are $a_0, a_1, a_2, a_3, a_4, a_5, a_6,$ and $a_7$. The user can assign any one of them as
the starting address. In this example, the user addresses $a_3$. The resulting level-0 bit-vector is $11101111$.

Free 2 memory blocks at address $a_4$ (assume the initial bit-vector is $11111111$).

For size = 2, there are four possible ABRSs. The starting address for these ABRSs are $a_0$, $a_2$, $a_4$, and $a_6$. User can assign any one of them as the starting address. In this example, the user addresses $a_4$. The resulting bit-vector is $11110011$.

Free 3 memory blocks at address $a_4$ (assume the initial bit-vector is $11111111$).

For size = 3, there are two possible ABRSs. The starting address for these ABRSs are $a_0$, and $a_4$. User can assign any one of them as the starting address. In this example, the user addresses $a_4$. The resulting bit-vector is $11110001$. 
Free 5 memory blocks at address $a_0$ (assume the initial bit-vector is 1111111).

For size = 5, there only one possible ABRS. The starting address must be $a_0$. In this example, the user addresses $a_0$ correctly. The resulting bit-vector is 00000111.

2.3.2 The CBT block diagram
2.3.3 Pin description

<table>
<thead>
<tr>
<th>Name</th>
<th>Input/Output</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRE_RDUP[6:0]</td>
<td>input</td>
<td>Pre_roundup requested block size</td>
</tr>
<tr>
<td>POST_RDUP[6:0]</td>
<td>input</td>
<td>Post_roundup requested block size. The PRE_RDUP and POST_RDUP are used to find out the location of bits expected to be allocated or deallocated.</td>
</tr>
<tr>
<td>ADDR_IN[6:0]</td>
<td>input</td>
<td>Input address. This is the starting address of the deallocating memory blocks specified by user.</td>
</tr>
<tr>
<td>ADDR_OUT[6:0]</td>
<td>output</td>
<td>Output address. This is the starting address of the allocated memory block.</td>
</tr>
<tr>
<td>FLIP_ALL</td>
<td>input</td>
<td>Flip_all pin. When this pin is ‘1’, the CBT will allocate all 128 mapping-bits in the bit vector.</td>
</tr>
<tr>
<td>ENABLE</td>
<td>input</td>
<td>Enable pin. The CBT will be disabled when ENABLE = ‘0’ and will be activated when ENABLE = ‘1’.</td>
</tr>
<tr>
<td>ADDR_SEL</td>
<td>input</td>
<td>Address-select pin. When ADDR_SEL = ‘1’, the CBT runs in allocation mode and allocates the memory blocks specified by the starting address generated internally. When ADDR_SEL = ‘0’, the CBT runs in deallocation mode and frees the memory blocks specified externally from ADDR_IN[6:0].</td>
</tr>
<tr>
<td>IN[127:0]</td>
<td>input</td>
<td>IN bus. This inputs the 128 mapping bits of the bit-vector from the MEM block.</td>
</tr>
<tr>
<td>FLIP[127:0]</td>
<td>output</td>
<td>FLIP bus. This contains 128 bits. The CBT sets the flip bit to ‘1’ when it allocates or deallocates the corresponding mapping bit. Otherwise, the flip bit is always set to ‘0’.</td>
</tr>
</tbody>
</table>

2.4 SELECTOR

There are 16 bit-vectors in the MEM block. Each bit-vector contains 128 mapping bits, which represent 128 cache memory blocks. The allocation status of each mapping bit in the bit-vectors changes frequently during the normal allocation/deallocation operations. When a new allocation request arrives, the SELECTOR has to select the appropriate bit-vector that has an ABRS available. SELECTOR selects the bit-vector by sending the Selected Vector Address to the MEM block. The MEM will then send out the selected bit-vector to the CBT.
Scheme of operation:

The SELECTOR chooses one of the 16 bit-vectors by the following algorithm:

if  (there are availability vectors that have a run of exactly as many zero
bits as specified by Brounded_up)
{
    if  the number of such availability vectors is more than one
        Choose the one with the lowest address.
    else
        Select that vector.
}
else if there are availability vectors that have a run of zero bits larger than
specified by Brounded_up)
{
    if number of such availability vectors is more than one
        Choose the one with the lowest address.
    else
        Select that vector.
}
else
    Report error.
The selected vector address is output to the MEM to select the bit-vector for allocation. If no vector has a large enough block available for such allocation request, SELECTOR will set the ERROR pin to '0'.

### 2.5 MEM (Memory) Block

A MEM block has 16 vectors. Each vector in turn is composed of a bit-vector and an availability vector.

A bit-vector has 128 mapping bits. Each mapping bit represents the allocation status of the corresponding cache memory block.

An availability vector has 7 availability bits. It represents the size of the largest contiguous run of free mapping bits in that bit-vector.

The vector is selected by the vector address given by SELECTOR, and its data is output to CBT for allocation/deallocation. The data flow between MEM and CBT is shown in the following steps:

***Need to fix grammar in figure
The CBT determines which mapping bits in bit-vector should be allocated or freed by considering:
- current status of the mapping bits
- in allocation or deallocation mode
- B value
- Brounded_up value
- starting address of the block to be deallocated

There are 128 flip bits on the flip bus. Each bit is corresponding to the mapping bit on the same address. If CBT wants to allocate or deallocate a mapping bit, it will set the corresponding flip bit to '1'. The other flip bit are all '0'.

The CBT will calculate the availability of the new mapping bits in bit-vector and output it through availability bus.

1. The MEM outputs the 128 mapping bits of the bit-vector through IN bus. The bit-vector is specified by the selected vector address (SEL) sent from SELECTOR.

2. At the rising edge of CLK2, the MEM will toggle the value of mapping bits that have the corresponding flip bits being set to '1'. After the toggling, the new mapping bits are sent through IN bus to CBT.

3. At the falling edge of CLK2, the MEM will update the availability value into the availability vector.
2.6 2-1 MUX

The 2-1 MUX is a 2-to-1 multiplexer.

During allocation operation, ALLOC = '1'. The 2–1 MUX will select and output the SELECTOR’s data onto the SEL bus which will select a suitable bit-vector for allocation.

In deallocation operation, the user will specify the block size and the starting address of the memory block. The starting address will go through the LATCH_C and come to the 4–1 DECODER. During deallocation operation, ALLOC = '0' and the 2–1 MUX will select the starting address from the 4–16DECODER and output it to select the bit-vector.

2.7 4-16 DECODER

This 4-to-16 decoder converts from a 4-bit address to 16 bits of selection information. The 16-bit value will go through the 2–1MUX to MEM and select the desired bit-vector for deallocation operation.

2.8 16-4 ENCODER

This 16-to-4 encoder converts from 16 bits of selection information to a 4-bit address. During allocation operation, the address of selected bit-vectors generated from SELECTOR will go through the 2–1MUX and 16–4 ENCODER and is finally output to the user.

2.9 ERROR

ERROR will be set to '0' only when an “error” occurs.

In allocation operation, possible errors are:

- User requests an allocated block size that is larger than 128 (number of all mapping bits on a bit-vector).
- SELECTOR cannot find any bit-vector with possible possible ABRS (allocation block of the requested size).

In deallocation operation, a possible error is:

- The user requests to free a memory with size larger than 128 blocks.
3. System Data Flow

The memory allocator can allocate or deallocate an arbitrary number of requested blocks (mapping bits) in a single clock cycle. How can it allocate or free the mapping bits? We now illustrate the operations through the data flow diagrams.

3.1 Allocation operation

3.1.1 Functional timing

The input data, ALLOC and B[7:0], must be valid before the rising edge of CLK1. ADDR_OUT[10:0] (starting address of allocated block) and ERROR will be latched at the rising edge of CLK2. The timing for sequential allocation operations is:

![Timing Diagram]

---

---
3.1.2 Data Flow

The data flow sequence is listed below:

After rising edge of CLK1

step 1. Triggered by the rising edge of CLK1, the B[7:0] (requested allocation-block size) and the ALLOC (operation mode) are latched in through the LATCH_A and the LATCH_C.

step 2. The B[7:0] value is rounded_up for availability comparision and CBT processing.

step 3. The COMPARATOR compares the availability value of the 16 bit-vectors with the Brounded_up to select a suitable bit-vector. This bit-vector must have ABRS for allocation.
step 4. The selected vector address is sent out through the 2–1MUX and finally to the MEM.

step 5. The bit-vector selected by selected vector address is sent to the CBT through the IN bus.

step 6. The CBT calculates the location of the requested allocated block from the information of B[7:0], Brounded_up[7:0], and IN[127:0]. It then sets the corresponding flip bit to '1'. The 128 flip bits are sent out through FLIP[127:0].

At the same time, the starting address of the allocated block is generated and passed out to the LATCH_D.

After rising edge of CLK2

step 7. Triggered by the rising edge of CLK2, the MEM uses the flip-bit value to update the corresponding bit-vector. The updated bit-vector is sent out to the CBT.

At the same time, the starting address and error signal are latched out through LATCH_D and LATCH_E.

step 8. The CBT generates the availability value of the updated bit-vector and send it to the MEM through AVAIL[7:0].

After falling edge of CLK2

step 9. Triggered by the falling edge of CLK2, the AVAIL[7:0] value is updated to the availability vector.
3.2 Deallocation operation

3.2.1 Functional timing

Timing for sequential free operations:

The input data: ALLOC, ADDR_IN[10:0] and B[7:0], must be valid before the rising edge of CLK1. ERROR will be latched at the rising edge of CLK2.
3.2.2 Data Flow

The data flow sequence is listed below:

After rising edge of CLK1

step 1. Triggered by the rising edge of CLK1, B[7:0] (requested allocated block size), ADDR_IN[10:0] (starting address of the block being freed), and ALLOC (operation mode) are latched in through LATCH_A, LATCH_B, and LATCH_C.

step 2. The B[7:0] value is rounded_up for CBT processing.

        4–16 ENCODER converts the ADDR_IN[10:7] to the selected vector address.

step 3. The bit-vector selected by selected vector address is sent to the CBT through the IN bus.

step 4. The CBT calculates the location of the block being freed from the information of B[7:0], Brounded_up[7:0], ADDR_IN[7:0], and IN[127:0]. It then sets the corresponding flip bit to '1'. The 128 flip bits are sent out through FLIP[127:0].
After rising edge of CLK2

step 5. Triggered by the rising edge of CLK2, the MEM uses the flip bits value to update the corresponding bit-vector. The updated bit-vector is sent out to the CBT.

step 6. The CBT generates the availability value of the updated bit-vector and sends it to the MEM through AVAIL[7:0].

After falling edge of CLK2

step 9. Triggered by the falling edge of CLK2, the AVAIL[7:0] value is updated to the availability vector.
4. Circuit Description

4.1 CBT

The MEM connects the selected bit-vector and availability vector to the CBT. This connection is controlled by the selecting address from SELECTOR in allocation mode or from the user in deallocation mode. The CBT will perform the following operations in the various modes:

a. Allocation mode:

   The CBT enters allocation mode when ALLOC pin = '1'.

   step 1. Receives the requested block size B from LATCH_A.
           Receives the rounded up size $B_{\text{roundup}}$, from ROUND_UP.
           This is equal to the size of the ABRS (allocation block of the requested size).
           Receives the selected bit-vector from MEM.

   step 2. Finds out the first available ABRS.

   step 3. Generates the starting address of the first available ABRS.
           Finds out the position of the mapping bits intended to be allocated. Sets the corresponding flip bits to '1'.

   step 4. Generates the availability value when it receive the new bit-vector from the MEM.

b. Deallocation mode:

   The CBT enters deallocation mode when ALLOC pin = '0'.

   step 1. Receives the requested block size B from LATCH_A.
           Receives the starting address of the deallocating block from user through LATCH_B.

   step 2. Finds out the position of the mapping bits intended to be freed. Sets the corresponding flip bits to '1'.

   step 3. Generates the availability value when it receives the new bit-vector from the MEM.

How can the CBT perform the above functions? We will illustrate the operation by looking at the mechanisms for:

- generating the availability vector
- finding the first available ABRS
- generating the flip-bit signal
- generating the starting address of the allocated block
4.1.1 Generation of availability value

Let’s look at an example of a CBT for an 8-bit bit-vector.

Fig 4.1

<table>
<thead>
<tr>
<th>Size</th>
<th>Availability</th>
<th>Node level</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The AV[3:0] are the availability lines of this CBT. Each AV line is connected to a pull-high resistor and is pulled down to '0' whenever one or more connecting node-to-node lines (the horizontal lines in the diagram) are '0' (low-active pull-low wired-OR). Any '0' on the node-to-node line will pull the same-level AV line to '0'. If no '0' exists, the AV line on the same level will remain '1'. In this example, AV0=AV1=AV2='0'. This means that contiguous runs bits of length 2^0, 2^1, and 2^2 exist. The system can know how many contiguous free mapping bits are present by checking the availability value at any time.

4.1.2 Find the first available ABRS (allocation block of the requested size)

How to calculate the size of ABRS?

\[
\text{size of ABRS} = 2^{\left\lfloor \log B \right\rfloor}
\]

In the or-tree, if the output of an level-\(n\) or-gate is '0', the children below this or-gate must all be '0'. Therefore, if size of ABRS is \(2^n\), the CBT will look on level \(n\) to search for the first or-gate with a '0' output value.

There is another tree structure called the and-tree. It can search for the first '0' in a CBT. Let’s look at an example of seaching for the first '0' in 8 blocks.
Let's look at the possible input/output relations in an and-gate:

Whenever there is at least one '0' on the input pins, the output will be '0'. Therefore, if we look down from the top of and-gate and see a '0', we can be sure that there is at least one '0' among its children. As we are only concerned with the side the first '0' is located on, we can derive a simple relationship from the above three and-gates.

- When the left input pin is '1', the first '0' must be on the left.
- When the left input pin is '0', the first '0' must be on the right.

Let's come back to look at the and-tree. In the following sections, we will use an and-node to represent an and-gate and or-node to represent an or-gate. We can start a searching path coming down from the output of the topmost
**and-node** \( a \). Whenever the search path comes to an **and-node**, it will check the value on the left input pin. A value of '1' means that the first '0' must be on the right, so the search path will proceed down to the right child. A value of '0' means the first '0' must be on the left, so the search path will go down to the left child.

In this example, the search path will go down in this way:

**Fig 4.4**

<table>
<thead>
<tr>
<th>node</th>
<th>value of left input pin</th>
<th>turning direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>0</td>
<td>left</td>
</tr>
<tr>
<td>( b )</td>
<td>0</td>
<td>left</td>
</tr>
<tr>
<td>( d )</td>
<td>1</td>
<td>right</td>
</tr>
</tbody>
</table>

It finally reaches the first '0' at \( a_1 \).

In this example, we are searching for the first '0' on level 0. We can select the first '0' on a higher level by dynamically ignoring the **and-node**s below that level.

The **or-tree** can show us if a block of \( 2^n \) contiguous '0's exists or not. The **and-tree** leads us to find the first '0' on a particular level. We can combine these two functions to find the first ABRS.

Assume we want to allocate \( 2^0 \) block. We need to search for the first ABRS of size \( 2^0 \). Let's connect the **or-tree** and **and-tree** together in this way:
Therefore, if we have to find the first '0' on level 0, we can connect all the level-0 pins to the and-tree as in the diagram above. The search path will go down through each and-node and finally locate the first ABRS at address $a_1$.

Assume we have to allocate three blocks. We need to find the first ABRS of size $2^2$. Therefore we search the first '0' on level 2.
The **and**-tree can easily see that the first '0' is located on the right. This means the first ABRS is on the right side of the **and**-tree.

The connection between the **and**-tree and the **or**-tree is set up dynamically based on the ABRS size. The connection diagram is shown below:

**Fig 4.7**

![Diagram showing connection between **and**- and **or**-trees based on ABRS size]

The ABRS size, held in $s_3s_2s_1s_0$, is used to control the ON/OFF of the array switches. ($s_0$ is not shown in the diagram because there are no child nodes below it.) For example, if size = $2^2 = 0100_2$.

- $a_3 = 0$: sw3a = OFF, sw3b = ON
- $a_2 = 1$: sw2a = ON, sw2b = OFF
- $a_1 = 0$: sw1a = OFF, sw1b = ON
- $a_0 = 0$: nil, nil

The connection will then be as shown in Fig 3.6.

### 4.3. Generation of the flip-bit signal
How can the CBT locates the position of the bits allocated and set the corresponding flip-bit to '1'?

Assume we have a flipper-tree which works together with the and-tree. The direction data (value of the left input pin of each and-node) for searching is passed to the flipper-nodes.

Fig 4.8

In the flipper-tree, we have an enable path going down from the top. It is similar to the search path in the and-tree. The enable path can find the first ABRS, which is equivalent to the first '0' at the corresponding level, by checking the direction data from the and-tree.

From the and-tree diagram Fig 4.4, the direction data from each and-node is used by the flipper-node in the corresponding location in the figure below. By the same mechanism, the enable path can easily find the first '0'. Assume we use the value of the left output pin on each and-node in diagram Fig 3.5 and put them into the flipper-tree:

Fig 4.9
When the direction data from the and-node is '0', the enable path goes down to the left by setting the left output pin to '1'. If it is '1', enable path descends to the right by setting the right output pin to '1'. Finally, a continuous enable path forms going from top to the first '0' at address \(a_1\).

How can the flipper-tree generate the flip-bit signal? The enable path is formed by setting to '1' all the lines it passes. The lines being set to '1' are activated. The other lines are all '0'; they are not activated. The enable path has the useful characteristic that it always points to the first ABRS. If we want to allocate \(2^n\) blocks (flip \(2^n\) mapping bits), we can send a signal to the level-\(n\) flipper-node at which the enable path is pointing. This signal is called the flip control and the flipper-node that receives a flip control will propagate a flip signal down to the lowest-level bits. Here, we use a new line, called the flip line, to propagate the flipping signal downward. Looking from the site of a flipper-node, the flip line coming from the parent is called flip input, while the flip lines going to the children are called flip outputs.

Fig 4.10
A flip line is activated by being set to '1' and deactivated by being set to '0'. For example, in case of allocated block size $= 2 = 2^1$, we can set the flip control of level-1 nodes to '1'. This will cause the flip output of level-1 nodes to be '1' if the flip output is going out with the enable path. The flip output signal will be propagated down and eventually used as the flip-bit signal for an allocation or deallocation operation. In this example, all four flipper-nodes on level-1 receive the same flip control value. However, the flip outputs are set according to the following conditions:

If the flip input of a flipper-node = '1'
   set the flip outputs to both left and right children to '1' \hspace{1cm} \ldots \text{(Cond. 1)}

   if the flip input of a flipper-node = '0'
   {
     \begin{align*}
     \text{if the flip control = '1'} & \{
     \text{if the left eable output = '1'} & \text{set the left flip output to '1'} \hspace{1cm} \ldots \text{(Cond. 2)}
     \text{if the right enable output = '1'} & \text{set the right flip output to '1'} \hspace{1cm} \ldots \text{(Cond. 3)}
     \}
     \}
   }
   \}
   \}

In all other conditions, the flip outputs are kept '0' \hspace{1cm} \ldots \text{(Cond. 4)}

If the allocated block size $= 4 = 2^2$ and the enable line is pointing down to $a_0$, the flipper-tree will look like this:

Fig 4.11

* When the Enable or Flip line is active, it = '1'. Else, it = '0'.
The left flip input of node a is set '1' because of condition 2. The flip outputs of node a, d, and e are all set to '1' because of condition 1. The flip control for each level l is actually the value of bit l of the requested block size. A requested block size of 4 (100 in binary) will set the flip control on level-2 nodes to '1' while size of 6 (110 in binary) will set the flip control on all level-2 and level-1 nodes to '1'.

How can we allocate blocks of a size that is not exactly equal to $2^k$? We do this by adding another feature to the enable line. From Fig 4.9, we know that direction data from and-tree will direct the enable line to propagate down to the left child or to the right child. We call this data “address control.” With the address-control signal, the flip-node becomes:

![Fig 4.12](image)

In the diagram Fig 4.9, we have:

if the enable input = '1'
  if the address control = '1'
    set right enable output to '1'
  else
    set left enable output to '1'

We now change these conditions to:

if the enable input = '1'
  if the flip control = '0'
    if the address control = '1'
      Set right enable output to '1'    … (Cond. 5)
    else
      Set left enable output to '1'    … (Cond. 6)
  else
else
    Set both right and left enable outputs to '1' ... (Cond. 7)
    However, the right flip output will be kept '0' (Cond. 3
does not hold)
}
else
    All the enable outputs are kept '0'

Suppose the allocated block size = 5, then the flip control = 5 (101 in binary).
Assume we have a bit-vector of value 00000000 in binary.

Fig 4.13

Thus, the flip outputs on $a_0$ to $a_4$ are set to '1'. Here different nodes follow
different conditions.

<table>
<thead>
<tr>
<th>node</th>
<th>conditions for enable line</th>
<th>conditions for flip line</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>Cond. 7</td>
<td>Cond. 2</td>
</tr>
<tr>
<td>$b$</td>
<td>Cond. 6</td>
<td>Cond. 1</td>
</tr>
<tr>
<td>$c$</td>
<td>Cond. 6</td>
<td>Cond. 4</td>
</tr>
<tr>
<td>$d$</td>
<td>Cond. 6</td>
<td>Cond. 1</td>
</tr>
<tr>
<td>$e$</td>
<td>Cond. 8</td>
<td>Cond. 1</td>
</tr>
<tr>
<td>$f$</td>
<td>Cond. 6</td>
<td>Cond. 2</td>
</tr>
<tr>
<td>$g$</td>
<td>Cond. 8</td>
<td>Cond. 4</td>
</tr>
</tbody>
</table>

The conditions 1 to 8 determine all the operations on a flipper-node. The
flipper-node is implemented by circuits shown in Fig. C2 (on page 60). We
call the circuits the *flip cell* (*flip_c*). A flip cell will be connected with *or-node*
and-node, and other auxiliary logic to form a CBT cell (cbt_c), which is a basic unit of the CBT block. The logic of the CBT cell is shown in Fig. C3 (on page 61). In Fig. 4.13, the enable outputs on the level-0 flipper-node are not used because they are the lowest-level nodes. In Fig. C4 (page 62) is shown a simplified version of a flip node called a ground-level flip cell (gflip_c), which does not have the enable output pins. Similarly, in the CBT cell, the switching circuits between or-nodes and and-nodes are not necessary on level 0, and are therefore omitted. This yields a simplified version called a ground-level CBT cell (gcbt_c). The logic of gcbt_c is shown in Fig. C5 (page 63).

The cbt_c and gcbt_c are the basic units of the complete binary tree (CBT). Each cell contains an or-node, an and-node, and a flipper-node. The cells can be easily used to construct a CBT of various sizes. In our application, we build a CBT with 128 flip bits (128 flip outputs from the lowest-level nodes). The whole circuit is shown in Fig. C6 (page 64). The lowest level is composed of four 32-bit CBTs. The logic for a 32-bit CBT is shown in Fig. C7 (page 65).

How can one generate the starting address of the allocated blocks?

The enable path has a useful characteristic that it always goes down and reaches the starting bit of allocated block when it arrives the lowest level. We can easily extract the address from the enable path by wired-orring the left enable output pin of every flipper-node. As there are no enable outputs on the level-0 flipper-node, we can wired-or the left flip output pins and this will give us the same result.

For example, if the enable path goes down to address \( a_4 \) and two bits are allocated, the flipper-tree will look like this:

Fig 4.14
All the OUT_ADDR lines are connected to the pulled-high resistor. Whenever the enable path goes to the left, it will pull the OUT_ADDR line to '0'. If it goes down to right, the OUT_ADDR will remain '1'. In this example, the address generated is 100, which is the address of a4.

Deallocation operation:

In the CBT, a free operation is similar to an allocation operation except that the location of block being freed is specified externally by the user. In this case, the address control for flipper-node is supplied by the user, not from the and-node. In both the CBT cell and ground-level CBT cell, we have a selecting pin (SELECT) to choose the address from either the internal and-node or the external ADDR_IN signal. In allocation operation, the SELECT pin is set to '1'. Otherwise, it is set to '0'.

4.2. Other blocks

4.2.1 MEM

The MEM is composed of 8 vectors. Each vector in turn is composed of a 7-bit availability vector and a 128-bit bit-vector. The vector is selected by the vector address from SELECTOR and its bit-vector is output to the CBT for allocation/deallocation. Then, the updated availability value from the CBT
will be loaded into the availability vector in the same MEM vector from which it was obtained.

Fig 5.1

The logic of MEM is shown in Fig C8 (page 66). The MEM is separated into two parts, the toggle-memory (tmem) and the D-memory (dmem). The toggle-memory contains eight 128-bit bit-vectors and is shown in Fig C9 (page 67). The D-memory consists of eight 7-bit availability vectors and is shown in Fig. C10 (page 68). There is a CLR pin to initialize the vectors when the system is started. In our application, two memory blocks are used together to supply 16 vectors for the whole system.

4.2.2 SELECTOR

The SELECTOR compares the rounded_up requested block size $B_{\text{rounded}_\text{up}}$ with the unallocated memory sizes that are available (from the availability vector) in the MEM. It then chooses the appropriate vector for allocation. The choice is based upon the following algorithm:
The SELECTOR chooses one of the 16 bit-vectors by the following algorithm:

```plaintext
if there are availability-vectors that have exactly the same values as the \texttt{Brounded_up}
{
    if the number of such availability-vectors is more than one
        Choose the one with the lowest address.
    else
        Select that vector.
}
else if there are availability vectors with values larger than \texttt{Brounded_up}
{
    if the number of such availability-vectors is more than one
        Choose the one with the lowest address.
    else
        Select that vector.
}
else
    Report error.
```

Each SELECTOR is composed of eight comparators (\texttt{comp\_c}) to compare the availability value and one priority selector (\texttt{priority\_sel}) to select the right vector address.

\textit{Comparator (\texttt{comp\_c})}: This is an 8-bit comparator. Here, we use a 4-bit comparator to illustrate the concept.

If the rounded\_up requested Size = 4 (0100 in binary). We want to have an equal (\texttt{EQ}) bit and greater-than-or-equal (\texttt{GE}) bit to indicate whether the availability-vector value is \(=\), or \(\geq\) the rounded-up request size.

<table>
<thead>
<tr>
<th>rounded_up requested size</th>
<th>availability vector value</th>
<th>expected \texttt{EQ} value</th>
<th>expected \texttt{GE} value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0100</td>
<td>0001</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0100</td>
<td>0011</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0100</td>
<td>0111</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0100</td>
<td>1111</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Let’s investigate the logic to calculate the expected \texttt{EQ} and \texttt{GE} value for all possible availability-vector values. In the table below, \texttt{A[3:0]} is the rounded-up request size, and \texttt{B[3:0]} is the availability-vector value. Note that only a single bit of \texttt{A} can be 1.
For **GE**:

\[
\begin{array}{|c|c|c|c|}
\hline
A[3:0] & B[3:0] & Ci = Ai \times Bi & GE=C_0+C_1+C_2+C_3 \\
\hline
0100 & 0000 & 0000 & 0 \\
0100 & 0001 & 0000 & 0 \\
0100 & 0011 & 0000 & 0 \\
0100 & 0111 & 0100 & 1 \\
0100 & 1111 & 0100 & 1 \\
\hline
\end{array}
\]

where \( \times \) is **and** operator
\( + \) is **or** operator

For all possible values of \( B \), only the last two \( B \) value can have the GE set to '1'. Therefore, GE can be obtained by first performing \( Ci = (Ai \times Bi) \) followed by oring all the C bits together.

For **EQ**:

\[
\begin{array}{|c|c|c|c|c|}
\hline
\hline
0100 & 0000 & 0000 & 0000 & 0 \\
0100 & 0001 & 0000 & 0000 & 0 \\
0100 & 0011 & 0000 & 0000 & 0 \\
0100 & 0111 & 0100 & 0100 & 1 \\
0100 & 1111 & 0100 & 0000 & 0 \\
\hline
\end{array}
\]

where \( \times \) is **and** operator
\( + \) is **or** operator

\[
D_0 = C_0 \times (\neg B_1)
\]
\[
D_1 = C_1 \times (\neg B_2)
\]
\[
D_2 = C_2 \times (\neg B_3)
\]
\[
D_3 = C_3 \times 1
\]

Only in the fourth row of the table is EQ set to '1'. This is because only in the fourth case is the requested block size rounded up to exactly the size of the largest available block. The circuit’s implementation in an 8-bit comparator is shown in Fig. C11 (page 69). This circuit is called a comparator cell (comp_c).

The GEs and EQs from the eight comparator cells are sent to the priority selector to select the vector address according to the conditions given above (on page 45). The logic of the priority selector is shown in Fig. C12 (page 70). When all the EQs and GEs are '0', it will raise an error by setting the ERROR pin to '0'. The whole SELECTOR circuit is shown in Fig. C13 (page 71). The EQIN, EQOUT, GEIN, GEOUT, and CASCADE_IN are pins specially used for
cascading two 8-vector SELECTORs to form a 16-vector SELECTOR. The SELECTOR with the higher priority on its vector address will connect the CASCADE_IN pin to '0'. The one with lower priority will connect the CASCADE_IN pin to '1'. A diagram of connecting two SELECTORs and the MEM blocks is shown below: GEIN, GEOUT below

4.2.3 ROUND_UP

ROUND_UP converts the requested block size $B$ to the rounded_up block size $B_{\text{rounded_up}}$ according to the following relation:

$$B_{\text{rounded_up}} = 2 \left\lceil \log B \right\rceil$$

The maximum requested block size in our system is 128. ROUND_UP will set ERROR to '0' if $B$ is larger than 128. If $B$ is exactly equal to 128, ROUND_UP will issue a FLIP_ALL signal to the CBT to allocate all 128 mapping bits in a faster way. The circuits for ROUND_UP is shown on Fig C14 (page 72).

4.2.4 2-1 MUX

A 2-to-1 multiplexer selects the vector address from the external ADDR_IN or from the internal SELECTOR according to the value of the operation mode
pin (ALLOC). 2–1MUX will output the vector address from the source to the MEM.

\[
\text{If } \text{ALLOC} = '1' \quad \text{source is the SELECTOR} \\
\text{else} \quad \text{source is the external ADDR_IN.}
\]

The circuits are shown on Fig C15 (page 73).

### 4.2.5 16-4 ENCODER

A 16-to-4 encoder converts vector converts 16 address lines to a 4-bit ADDR_OUT[10:7]. The circuit is shown in Fig. C16 (page 74).

### 4.2.6 4-16 DECODER

A 4-to-16 decoder converts a 4-bit ADDR_IN[10:7] to 16 bits for internal processing. The circuit is shown on Fig C17 (page 75).

### 4.2.7 LATCHES

The are five latches in this system. They are all triggered by the positive edge of a clock and cleared by CLR. The input latches are all triggered by CLK1 while output latches are triggered by CLK2. The logic of LATCH_C is shown on Fig C18 (page 76) as an example.

These signals are latched in or out:

<table>
<thead>
<tr>
<th>Latch</th>
<th>Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>LATCH_A</td>
<td>B[7:0] (the size of the requested allocation block size or the block to be freed)</td>
</tr>
<tr>
<td>LATCH_B</td>
<td>ADDR_IN[10:0] (starting address of block to be freed)</td>
</tr>
<tr>
<td>LATCH_C</td>
<td>ALLOC (operation mode)</td>
</tr>
<tr>
<td>LATCH_D</td>
<td>ADDR_OUT[10:0] (starting address of block that is allocated)</td>
</tr>
<tr>
<td>LATCH_E</td>
<td>ERROR (error signal )</td>
</tr>
</tbody>
</table>

### 5. SIMULATION RESULTS
Three simulation patterns are used to debug the circuits and measure the timing characteristics during the final development stage. In this section, several items will be discussed:

- the functional timing.
- the gate delay and the simulation pins.
- three test patterns and the simulation waveforms.
- the AC timing.

**5.1 FUNCTIONAL TIMING**

The system can allocate or free a block of arbitrary size in a single clock cycle. Five different combinations of operation timings are described below:

**5.1.1 Allocation/allocation cycles**

The timing for sequential allocation operations:

For correct operation, the ALLOC and B[7:0] must be valid before the rising edge of CLK1.
5.1.2 Deallocation/deallocation cycles

The timing for sequential deallocation operations:

For correct operation, the ALLOC and B[7:0] must be valid before the rising edge of CLK1.

5.1.3 Allocation/deallocation/allocation cycles

For correct operation, the ALLOC, ADDR_IN[10:0], and B[7:0] must be valid before the rising edge of CLK1.
5.1.4 Allocation/nop/allocation cycles

A cycle is a “nop” when no operation is done. When the requested block size B[7:0] (in either allocation or deallocation operation) is zero, the system will assume the user is not using the memory allocator and will enter nop mode. The ERROR pin remains '0' during this cycle.

5.1.5 Deallocation/nop/deallocation cycles

5.2 GATE DELAY AND SIMULATION PINS
5.2.1 Gate Delay

In the simulation, we use the following delay time for different logic gates.

<table>
<thead>
<tr>
<th>Gate Type</th>
<th>TPLH, TPHL</th>
<th>TPLZ, TPHZ, TPZH</th>
<th>CLKtoQ0, CLKtoQ1, CLKtoQN0, CLKtoQN1</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>0.5 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BUF</td>
<td>1.0 ns</td>
<td>1.2 ns</td>
<td>—</td>
</tr>
<tr>
<td>TRIBUF</td>
<td>1.2 ns</td>
<td>1.2 ns</td>
<td>—</td>
</tr>
<tr>
<td>NAND 2 inputs</td>
<td>0.8 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>NAND 3 inputs</td>
<td>0.8 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>NAND 4 inputs</td>
<td>1.0 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>NOR 2 inputs</td>
<td>0.8 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>NOR 3 inputs</td>
<td>0.8 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>NOR 4 inputs</td>
<td>1.0 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>AND 2 inputs</td>
<td>0.8 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>AND 3 inputs</td>
<td>0.8 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>AND 4 inputs</td>
<td>1.0 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>OR 2 inputs</td>
<td>0.8 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>OR 3 inputs</td>
<td>0.8 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>OR 4 inputs</td>
<td>1.0 ns</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>LATCH</td>
<td>—</td>
<td>—</td>
<td>7.5 ns</td>
</tr>
<tr>
<td>T_F/F</td>
<td>—</td>
<td>—</td>
<td>3.5 ns</td>
</tr>
<tr>
<td>D_F/F</td>
<td>—</td>
<td>—</td>
<td>7.5 ns</td>
</tr>
</tbody>
</table>

5.2.2 Simulation pins

External pins:

<table>
<thead>
<tr>
<th>Name</th>
<th>Input/output</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK1</td>
<td>input</td>
<td>Clock 1</td>
</tr>
<tr>
<td>CLK2</td>
<td>input</td>
<td>Clock 2</td>
</tr>
<tr>
<td>CLR</td>
<td>input</td>
<td>Clear pin. This is the signal for initialization</td>
</tr>
<tr>
<td>B[7:0]</td>
<td>input</td>
<td>Requested block size (allocation or deallocation operation)</td>
</tr>
<tr>
<td>ALLOC</td>
<td>input</td>
<td>Operation mode</td>
</tr>
<tr>
<td>ADDR_IN[10:0]</td>
<td>input</td>
<td>Starting address of the requested freeing block</td>
</tr>
<tr>
<td>ADDR_OUT[10:0]</td>
<td>output</td>
<td>Starting address of the allocated block</td>
</tr>
<tr>
<td>ERROR</td>
<td>output</td>
<td>Error signal</td>
</tr>
</tbody>
</table>
### Internal pins:

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN[127:0]</td>
<td>The bit-vector outputed from the MEM to the CBT.</td>
</tr>
<tr>
<td>FLIP[127:0]</td>
<td>The flip control bits output from the CBT to MEM.</td>
</tr>
<tr>
<td>PRE[7:0]</td>
<td>The pre_roundup value of the requested block size.</td>
</tr>
<tr>
<td>POST[7:0]</td>
<td>The post_roundup value of the requested block size.</td>
</tr>
<tr>
<td>FLIP_ALL</td>
<td>When user allocates 128 mapping bits, FLIP_ALL will be ‘1’.</td>
</tr>
<tr>
<td>ALLOC_</td>
<td>The ALLOC signal after being latched in.</td>
</tr>
<tr>
<td>SE0[15:0]</td>
<td>The selected vector address outputed from the SELECTOR.</td>
</tr>
<tr>
<td>DEO[15:0]</td>
<td>The output of 4-16 DECODER.</td>
</tr>
<tr>
<td>ERROR_</td>
<td>The error signal before being latched out.</td>
</tr>
<tr>
<td>AVAIL[7:0]</td>
<td>The availability value of circuit processing Bit-vector. It is generated from CBT.</td>
</tr>
<tr>
<td>SEL[15:0]</td>
<td>The selected vector address going into the MEM.</td>
</tr>
<tr>
<td>AV[15:0]</td>
<td>The availability vector values of vector[15:0].</td>
</tr>
<tr>
<td>OUT_ADDR[6:0]</td>
<td>The ADDR_OUT signal before latched out.</td>
</tr>
</tbody>
</table>

#### 5.3 TEST PATTERNS AND SIMULATION WAVEFORMS

Three different test patterns are used for allocation, deallocation, and mixed operations.

The clock cycle used is 75ns.
5.3.1 Allocation test

a. Test patterns:

<table>
<thead>
<tr>
<th>operation</th>
<th>ADDR_</th>
<th>B</th>
<th>availability</th>
<th>ADDR_</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td>reset</td>
<td>reset</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 FF</td>
<td>—</td>
</tr>
<tr>
<td>alloc</td>
<td>1</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 0001 7F</td>
<td>000</td>
</tr>
<tr>
<td>alloc</td>
<td>2</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 000D 7F</td>
<td>002</td>
</tr>
<tr>
<td>alloc</td>
<td>3</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007D 7F</td>
<td>004</td>
</tr>
<tr>
<td>alloc</td>
<td>4</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007D 7F</td>
<td>008</td>
</tr>
<tr>
<td>alloc</td>
<td>5</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>010</td>
</tr>
<tr>
<td>alloc</td>
<td>6</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>018</td>
</tr>
<tr>
<td>alloc</td>
<td>7</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>020</td>
</tr>
<tr>
<td>alloc</td>
<td>8</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 FF7F 3F1F 0F7D 7F</td>
<td>028</td>
</tr>
<tr>
<td>alloc</td>
<td>9</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>030</td>
</tr>
<tr>
<td>alloc</td>
<td>10</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>040</td>
</tr>
<tr>
<td>alloc</td>
<td>15</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>050</td>
</tr>
<tr>
<td>alloc</td>
<td>20</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>060</td>
</tr>
<tr>
<td>alloc</td>
<td>13</td>
<td>1</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>080</td>
</tr>
<tr>
<td>alloc</td>
<td>1</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>090</td>
</tr>
<tr>
<td>alloc</td>
<td>2</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>100</td>
</tr>
<tr>
<td>alloc</td>
<td>3</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>180</td>
</tr>
<tr>
<td>alloc</td>
<td>4</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>200</td>
</tr>
<tr>
<td>alloc</td>
<td>5</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>280</td>
</tr>
<tr>
<td>alloc</td>
<td>6</td>
<td>1</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>300</td>
</tr>
<tr>
<td>alloc</td>
<td>65</td>
<td>2</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>380</td>
</tr>
<tr>
<td>alloc</td>
<td>65</td>
<td>3</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>400</td>
</tr>
<tr>
<td>alloc</td>
<td>65</td>
<td>4</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>480</td>
</tr>
<tr>
<td>alloc</td>
<td>65</td>
<td>5</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>500</td>
</tr>
<tr>
<td>alloc</td>
<td>128</td>
<td>6</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>580</td>
</tr>
<tr>
<td>alloc</td>
<td>128</td>
<td>7</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>600</td>
</tr>
<tr>
<td>alloc</td>
<td>128</td>
<td>8</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>680</td>
</tr>
<tr>
<td>alloc</td>
<td>128</td>
<td>9</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>780</td>
</tr>
<tr>
<td>alloc</td>
<td>128</td>
<td>10</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>98</td>
</tr>
<tr>
<td>alloc</td>
<td>3</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>74</td>
</tr>
<tr>
<td>free</td>
<td>13</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>60</td>
</tr>
<tr>
<td>alloc</td>
<td>7</td>
<td>0</td>
<td>0000 0000 0000 0000 0000 0000 0000 0000 007F 7F</td>
<td>07</td>
</tr>
</tbody>
</table>

b. Simulation waveforms:

The simulation waveforms are shown on page 77-80.
5.3.2 Deallocation test

a. Test patterns:

<table>
<thead>
<tr>
<th>operation</th>
<th>ADDR_</th>
<th>vector</th>
<th>#</th>
<th>current bit-vector</th>
<th>availability</th>
<th>ADD. OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td>reset</td>
<td>reset</td>
<td>0000</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>alloc</td>
<td>-</td>
<td>128</td>
<td>0</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>alloc</td>
<td>-</td>
<td>128</td>
<td>2</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>alloc</td>
<td>-</td>
<td>128</td>
<td>3</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>alloc</td>
<td>-</td>
<td>128</td>
<td>4</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>alloc</td>
<td>-</td>
<td>128</td>
<td>5</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>alloc</td>
<td>-</td>
<td>128</td>
<td>6</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>alloc</td>
<td>-</td>
<td>128</td>
<td>7</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>alloc</td>
<td>-</td>
<td>128</td>
<td>8</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>alloc</td>
<td>-</td>
<td>128</td>
<td>9</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>alloc</td>
<td>-</td>
<td>128</td>
<td>10</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>alloc</td>
<td>-</td>
<td>128</td>
<td>11</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>alloc</td>
<td>-</td>
<td>128</td>
<td>12</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>alloc</td>
<td>-</td>
<td>128</td>
<td>13</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>alloc</td>
<td>-</td>
<td>128</td>
<td>14</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>alloc</td>
<td>-</td>
<td>128</td>
<td>15</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>free</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>free</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>free</td>
<td>10</td>
<td>5</td>
<td>0</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>free</td>
<td>30</td>
<td>7</td>
<td>0</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>free</td>
<td>40</td>
<td>21</td>
<td>0</td>
<td>FFFF FFFF FFFF FEE000000</td>
<td>FF80</td>
<td>FFEE0</td>
</tr>
<tr>
<td>free</td>
<td>58</td>
<td>4</td>
<td>0</td>
<td>FFFF FFFF FFOE000000</td>
<td>FF80</td>
<td>FFEE0</td>
</tr>
<tr>
<td>free</td>
<td>68</td>
<td>8</td>
<td>0</td>
<td>FFFF FFFF FFOE000000</td>
<td>FF80</td>
<td>FFEE0</td>
</tr>
<tr>
<td>free</td>
<td>7A</td>
<td>2</td>
<td>0</td>
<td>F3FF FFE000000</td>
<td>FF80</td>
<td>FFEE0</td>
</tr>
<tr>
<td>free</td>
<td>1D0</td>
<td>15</td>
<td>3</td>
<td>FFFF FFFF 8000 FFFF</td>
<td>FFFF</td>
<td>FFFF</td>
</tr>
<tr>
<td>free</td>
<td>A2</td>
<td>2</td>
<td>1</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>free</td>
<td>108</td>
<td>7</td>
<td>2</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>free</td>
<td>210</td>
<td>15</td>
<td>4</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>free</td>
<td>3A0</td>
<td>29</td>
<td>7</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>free</td>
<td>2C0</td>
<td>31</td>
<td>5</td>
<td>FFFF FFFF 8000 0000</td>
<td>FFFF</td>
<td>FFFF</td>
</tr>
<tr>
<td>free</td>
<td>404</td>
<td>3</td>
<td>8</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>free</td>
<td>308</td>
<td>7</td>
<td>6</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>free</td>
<td>490</td>
<td>11</td>
<td>9</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>free</td>
<td>7A0</td>
<td>10</td>
<td>15</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>free</td>
<td>560</td>
<td>30</td>
<td>10</td>
<td>FFFF FFFF C000 0000</td>
<td>FFFF</td>
<td>FFFF</td>
</tr>
<tr>
<td>free</td>
<td>760</td>
<td>8</td>
<td>14</td>
<td>FFFF FF00 FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>free</td>
<td>5F0</td>
<td>15</td>
<td>11</td>
<td>8000 FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>free</td>
<td>681</td>
<td>1</td>
<td>13</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
<tr>
<td>free</td>
<td>603</td>
<td>1</td>
<td>12</td>
<td>FFFF FFFF FFFF FFFF</td>
<td>FF</td>
<td>FF00</td>
</tr>
</tbody>
</table>

b. Simulation waveforms:

The simulation waveforms are shown on page 81-84.
6.3.3 Mixed Test

a. Test patterns:

<table>
<thead>
<tr>
<th>operation</th>
<th>ADDR_</th>
<th>vector</th>
<th>availa-</th>
<th>ADDI</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td>reset</td>
<td>reset</td>
<td>0000</td>
<td>FF</td>
</tr>
<tr>
<td>alloc</td>
<td>—</td>
<td>10</td>
<td>0000</td>
<td>00</td>
</tr>
<tr>
<td>alloc</td>
<td>—</td>
<td>0</td>
<td>0000</td>
<td>00</td>
</tr>
<tr>
<td>alloc</td>
<td>—</td>
<td>12</td>
<td>0000</td>
<td>00</td>
</tr>
<tr>
<td>alloc</td>
<td>—</td>
<td>27</td>
<td>0000</td>
<td>00</td>
</tr>
<tr>
<td>free</td>
<td>10</td>
<td>9</td>
<td>0000</td>
<td>00</td>
</tr>
<tr>
<td>alloc</td>
<td>—</td>
<td>30</td>
<td>0000</td>
<td>00</td>
</tr>
<tr>
<td>free</td>
<td>50</td>
<td>5</td>
<td>0000</td>
<td>00</td>
</tr>
<tr>
<td>alloc</td>
<td>—</td>
<td>0</td>
<td>3FE0</td>
<td>00</td>
</tr>
<tr>
<td>alloc</td>
<td>—</td>
<td>3</td>
<td>3FE0</td>
<td>00</td>
</tr>
<tr>
<td>alloc</td>
<td>—</td>
<td>0</td>
<td>3FE0</td>
<td>00</td>
</tr>
<tr>
<td>free</td>
<td>20</td>
<td>15</td>
<td>3FE0</td>
<td>00</td>
</tr>
<tr>
<td>alloc</td>
<td>—</td>
<td>0</td>
<td>3FE0</td>
<td>00</td>
</tr>
</tbody>
</table>

b. Simulation waveforms:

The simulation waveforms are shown on page 85-86.

5.4 DISCUSSION OF AC TIMING

Five typical examples of AC timing are extracted from the previous simulation waveforms to illustrate the system timing. They are as listed below:

a. Allocation – with changing of selected vector address (SEO internally)

The waveform is shown on page 87. This timing happens when the current selected bit-vector does not have enough free mapping bits for allocation. Another appropriate bit-vector is selected and used as the new current bit-vector for the CBT. Is is a very tight timing. We can look at the two data paths below:

Rising edge of CLK1 $\rightarrow$ PRE $\rightarrow$ POST $\rightarrow$ SEO $\rightarrow$ SEL $\rightarrow$ FLIP ready

Total = 41.1 ns
Rising edge of CLK1 $\rightarrow$ PRE $\rightarrow$ POST $\rightarrow$ SEO $\rightarrow$ SEL $\rightarrow$ OUT_ADDR ready
Total = 44.0 ns

As both of the data paths must be ready before the rising of CLK2, the timing tolerance is only 6~8.9ns. This operation will be the bottleneck at our speed, and more effort should be expended to improve this part.

b. Allocation — without changing of selected vector address.

The waveform is shown on page 88. This timing happens when current bit-vector has enough free bits for allocation. The timing has a large tolerance as the FLIP data is ready 60.9ns before the CLK2.

c. Allocation — with user allocating 128 mapping bits at the same time.

The FLIP_ALL pin is enabled in this case. The waveform is shown on page 89. The FLIP data is ready long before CLK2. However, the timing tolerance between OUT_ADDR and CLK2 is very small, it is only about 4ns. This part should need further attention in the future.

d. Deallocation followed by allocation cycle.

The waveform is shown on page 90. The free cycle has much timing tolerance. The FLIP data is ready long before CLK2. However, the allocation-cycle timing is very tight. It takes longer for CBT to be stable, as the CBT internal data path is changed when it is changed from deallocation mode to allocation mode. There is only about 6ns timing tolerance between the OUT_ADDR and the CLK2.

e. Nop-deallocation-nop-allocation cycles

The waveform is shown on page 91. Compared with the allocation cycle, the free cycle does not need to generate the starting address from the PRE and POST values. The address is already ready as it is input by the user. The data path from CLK1 to FLIP ready only takes $7.5+3.2+6.3 = 17$ ns. On the other hand, the allocation cycle has to first compare the POST and values of the 16 availability vectors to get the vector address. The vector address then selects the corresponding bit-vector and outputs it to the CBT.

From rising edge of CLK1 $\rightarrow$ PRE $\rightarrow$ POST $\rightarrow$ SEO $\rightarrow$ SEL
(vector address) = 22.7ns.
and from SEL $\rightarrow$ IN (data from selected bit-vector) = 3.7ns
After the IN value becomes valid, the CBT will calculate the starting address of the allocated block and at the same time enable the appropriate flip-bit. From \( \text{IN} \rightarrow \text{FLIP ready} = 19.6\text{ns} \).

Totally, from \( \text{CLK1} \rightarrow \text{FLIP ready} = 22.7 + 3.7 + 19.6 = 46\text{ns} \). The timing tolerance is only 4ns. The timing tolerance for OUT_ADDR is further less. It is only about 2ns.

5.5 DISCUSSION

The system can work fine with a 75ns clock cycle. However, there is not enough timing tolerance in some allocation cycles. We can improve that by:

a. Optimizing some logic circuits. For example, some circuit libraries show a similar time delay on an 2-input or-gate and an 4-input OR-gate. In this case, we can use one 4-input or-gate to replace two 2-input or-gate in some circumstances to reduce delay time.

b. Change the system architecture.

This project is not finalized yet. Many improvements can be made. It will be an interesting project topic for other ambitious students.

The circuit diagrams and the simulation waveforms are shown on the following page 59-91.

The Viewlogic simulation files are stored under directory: "project/memory_allocator" in the Tatung workstation.