Documentation for the EMACSIM Computer System

K. Mandelberg
E. Leon

Department of Mathematics and Computer Science
Emory University
Atlanta, Georgia 30322

ABSTRACT

The Emory MACintosh SIMulator (EMACSIM) is a computer system that was designed as an educational tool for a beginning graduate course in operating systems and machine architecture at Emory University. EMACSIM is based on CHIP† (Cornell Hypothetical Instructional Processor) and GDB, the GNU debugger. The EMACSIM architecture is currently based on the MC68000†† and CHIP, but the goal of EMACSIM is to do a full simulation of the Macintosh II††† . The EMACSIM architecture includes dynamic memory mapping suitable for implementing virtual memory, eight interrupt priority levels, memory-mapped input/output and two modes of processor operation. The central processor of EMACSIM is compatible with the MC68000. Therefore, any code written for the MC68000 can be executed on EMACSIM. The EMACSIM simulator also supports input/output devices such as terminals, floppies, disks and printers. All interactions with EMACSIM take place through GDB, and its X11 window interface. Users can debug their programs at the source code level and the reader should refer to the GDB Manual. Currently GCC, the GNU compiler is used as a cross-compiler on the Sun workstations to generate 68000 code.

This manual is a revised version of the CHIP manual.

---

† Developed by Özalp Babaoğlu at Cornell University.

†† MC68000 is a Trademark of Motorola.

††† Macintosh and A/UX are Trademarks of Apple Computer.

Chapter One
Principles of Operation

1. Introduction

EMACSIM is designed to simulate a Macintosh II at a level of detail appropriate for an operating systems course. The EMACSIM architecture is originally based on the MC68000 microprocessor rather than the MC68020 which the Mac II uses because the 68000 is simpler and has a smaller instruction set. The design of EMACSIM was derived from an existing simulator, CHIP, which is based on the architecture of a PDP-11. The new machine supports dynamic memory mapping, eight interrupt priorities, memory-mapped I/O devices and two modes of operation: SUPERVISOR and USER.

The EMACSIM system has been implemented on a Mac II under A/UX. The I/O devices are simulated in software except for the floppy drive which is used by EMACSIM as a swap device.

This document, together with a processor handbook for a MC68000 , forms a complete description of the EMACSIM machine.

Throughout this manual the following notational conventions are used:

- Words being defined are *italicized*.
- Field F of register R is denoted by R.F.
- Bits of storage units are numbered right-to-left starting with 0.
- The i-th bit of a storage unit named N is denoted by N[i].
- The contents of a storage unit named N is denoted by (N).
- Memory addresses and operation codes are given in hex.

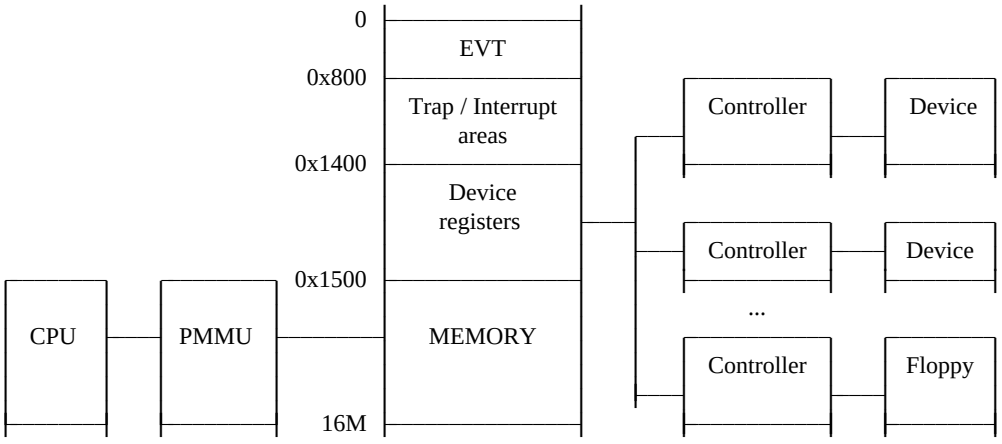


Figure 1. *Major components of EMACSIM*

2. Major Components

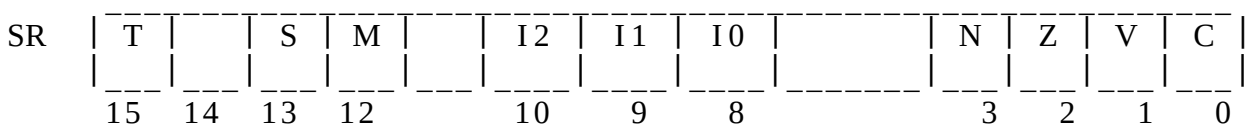
As shown in Figure 1, the CPU and I/O devices are connected through main memory. Device controllers, which act as a communication channel between devices and memory, transfer data directly to or from memory without CPU intervention. The CPU is interrupted by a device only upon the completion of an I/O operation.

3. Machine Registers

This section describes the registers internal to the CPU. Device registers, which are used to control I/O devices, are discussed in §8. Each CPU register can hold a 32-bit quantity called a *long word*.

The 68000 has 19 registers which form the *Processor State*: D0-D7, A0-A7, PC, SR and CRP. D0-D7 are general data registers, A0-A7 are address registers. A7 is the *Stack Pointer or SP*. If the processor is in supervisor mode, A7 represents the *Supervisor Stack Pointer or SSP* , otherwise A7 represents the *User Stack Pointer or USP*. The *Program Counter or PC* only uses the low order 24 bits.

The *Status Register or SR* is shown below:



FIELD	FUNCTION
T	if 1, the processor is in TRACE mode
S	if 1, the processor is in SUPERVISOR mode; USER mode otherwise
I2-I1-I0	represents a 3 bit binary number n. Interrupts of priority <= n are disabled
M	if 1, memory Mapping is enabled
N	set to 1 if last instruction executed yielded a Negative number; 0 otherwise
Z	set to 1 if last instruction executed yielded a Zero result; 0 otherwise
V	set to 1 if last instruction executed caused arithmetic overflow; 0 otherwise
C	set to 1 if last instruction executed caused a Carry; 0 otherwise

Table 1. *Status Register fields*

The setting of the condition codes N, Z, V and C are identical to the respective codes in the MC68000 and the reader is referred to the MC68000 handout for a detailed description.

TDCK (Time of Day ClocK) is a long-word register containing the elapsed time, in microseconds, since power-up.

IT (Interval Timer) is also a long-word register. It is decremented by one each microsecond. When the value reaches zero, a Clock interrupt occurs.

The MC68020 uses a co-processor to support virtual memory. The co-processor has its own registers and instructions. CRP is one of those registers. (see §6 for details).

4. Modes of Operation

The EMACSIM processor can operate in two modes: USER and SUPERVISOR. The mode of operation is determined by the setting of the SR.S bit. To change the processor mode, a new processor state must be loaded. This can be done by executing the MOVE.w <ea>,SR or as the result of a trap or interrupt.

In USER mode, execution of Privileged Instructions is illegal and causes a program trap. All instructions are executable in SUPERVISOR mode.

4.1. USER Mode

All USER mode instructions of the MC68000 are supported. Note that the floating point coprocessor instructions are not implemented in EMACSIM. Several assembly routines are provided to facilitate the work of the students and these can be found in ~cs580000/share/hoca/util. Please link these routines to your directory instead of copying them.

In the following descriptions, the ''Usage'' field indicates how the utility functions can be invoked through C program statements to be compiled with the **gcc** compiler (see Chapter 2 for details).

4.2. USER Utilities

STCK STore time of day ClocK
Operation: addr ← (SP)
(addr) ← contents of TDCK
Description: The contents of TDCK are stored in the long word pointed to by the argument on the top of the stack.
Usage: **long** time;

void
STCK(&time);

SYS Cause a system call trap
Operation: ↓(SP) ← system call number
trap &3
Description: Executing SYS causes a system call trap. The system call number is pushed on the stack. Then the trap &3 instruction is executed and exception processing will take place.
Usage: SYS0(); SYS1(); SYS2(); ... SYS17();

4.3. SUPERVISOR Utilities

In SUPERVISOR mode, privileged instructions as well as USER mode instructions are executable. The privileged instructions are described here.

LDST LoaD processor STate

Operation: addr ← (SP)
processor state registers ← (addr+i), i=0,4,...,72

Description: The processor state registers are loaded from the state stored in the 76 consecutive bytes pointed to by the argument on the top of the stack. Loading takes place in the following order: D0-D7, A0-A7, SP, PC, SR and CRP.

Usage: **state_t** newstate;

```
void
LDST(&newstate);
```

STST STore processor STate

Operation: addr ← (SP)
(addr+i) ← current processor state, i=0,4,...,72

Description: The current processor state is stored in the 76 consecutive bytes pointed to by the argument on the top of stack. Storing takes place in the following order: D0-D7, A0-A7, SP, PC, SR and CRP.

Usage: **state_t** savearea;

```
void
STST(&savearea);
```

LDIT LoaD Interval Timer

Operation: addr ← (SP)
contents of IT ← (addr)

Description: Register IT is loaded with the contents of the long word pointed to by the argument on the top of stack.

Usage: **long** interval;

```
void
LDIT(&interval);
```

HALT Halt

Description: Stop the CPU by executing a trap instruction that has no exception processing routine.

Usage:
```
void
HALT();
```

5. Memory

The EMACSIM memory is byte addressable. The MC68000 can access up to 16M of memory, but EMACSIM only uses 128K. A *word* in EMACSIM memory is two bytes long and is divided into a *high byte* and a *low byte*. The low bytes of words are stored at odd-numbered locations and the high bytes at even-numbered locations. An even address can denote a byte, word or long word, depending upon the context.

Certain memory locations are reserved for special purposes. The first 1024 bytes of memory are used for the *Exception Vector Table or EVT* and other areas of memory are used for Trap / Interrupt Areas and for memory mapped I/O.

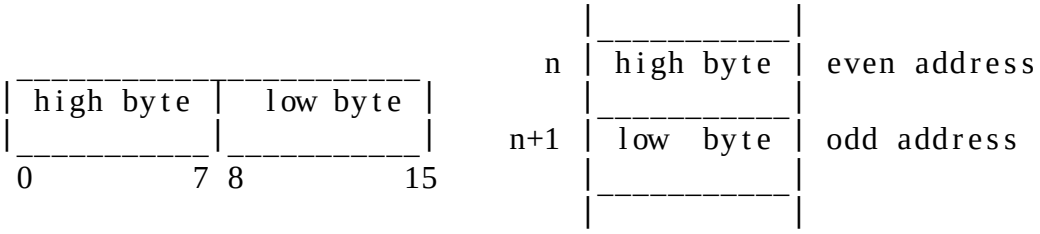


Figure 2. *A EMACSIM word and its representation in memory*

Instructions in memory must be word aligned. An attempt to execute an instruction starting at an odd address causes a Program Trap. Addresses of operands must also be word aligned, except for byte operands in byte instructions.

6. The Interrupt/Trap System

The sequence of instructions being executed by the CPU can be altered by two types of events. *Traps* are generated by the processor as the result of instruction execution. Examples include Illegal Operation Code, Zero Divide and Page Fault. *Interrupts* are due to sources external to the processor, such as I/O devices.

A trap or interrupt indicates a malfunction or a need to service a device or request. The CPU stops normal execution and services the exception. The series of actions taken by the CPU is called *Exception Processing*. The first 1K of memory is reserved for *Exception Processing*. This area of memory forms a table called the *Exception Vector Table or EVT*. This table contains 256 entries, each of them is 4 bytes long and can hold the address of a service routine. When an exception occurs, the CPU identifies the exception and this determines the *Exception Vector Number*. Then the CPU stores the *SR* in an internal register, sets the *SUPERVISOR* bit, clears the *TRACE* bit and sets the interrupt level to that of the exception. The address of the exception vector is found by multiplying the *Exception Vector* by 4. The *PC* and *SP* are pushed on the *SSP*. The *PC* is then loaded with the contents at the address of the exception vector. If there is no valid address there the CPU is halted.

We provide assembly routines that should be used as the exception service routines. These routines can be found in ~cs580000/share/hoca/util. The purpose of these routines is to save the current processor state and to load a new processor state. It is the student's responsability to load the address of these assembly routines at the correct location in the *EVT*. Logically the trap and interrupt areas are divided in two sections, the *Old Area* which holds the current processor state and the *New Area* which holds the new processor state. The new processor state has among other things the address of a trap or interrupt handler which the students will write.

6.1. Traps

Traps can be grouped into three categories:

Program
Memory Management (MM)
System Call (SYS)

MM traps are generated by the memory management system. SYS traps are generated by software. They occur as a consequence of executing a trap instruction. *Program* traps are caused by encountering abnormal conditions during instruction execution.

MM and Program traps always occur *during* execution of an instruction, as opposed to SYS traps, which occur as a *result* of executing a trap instruction.

Trap instructions can provide a way for a user to communicate with and request services from an operating system. For example, in order to write to a disk device a user may execute a particular trap instruction.

Since the MM and Program traps may occur anywhere during instruction execution, there must be a mechanism for restoring the state of the processor and memory to the values they had before the instruction was executed after the trap is handled. Fortunately, EMACSIM does this automatically, ensuring that when the trap is signaled, the state of the machine is as it was before the offending instruction was executed.

For each of these three categories of traps, there is a *trap area* in a location in memory. These areas are not hardware dependant and are selected by the assembly routines. The areas are 76 bytes long and logically subdivided into two consecutive regions of 19 long words each. The first of these regions is the *Old* area mentioned above and the second is the *New* area.

EVT Addresses for Traps

Different Program Traps have different addresses in the *EVT*. For example, Zero Divide is a program trap and its address in the *EVT* is 0x14.

Memory Managment traps are bus errors and the *EVT* address is 0x8.

Trap Category	Trap Area Address (convention)
PROG	0x800
MM	0x898
SYS	0x930

Fig 5. *Trap Vector Locations*

System traps are executed by calling a SYS function which pushes the system call number on the *SP* and executes a trap &3 assembly instruction. Therefore the *EVT* address for System traps is 0x8c = 0x80 + 3*4.

Note that all the program traps use the same *Trap Vector Location*, i.e the same *Old/New* areas, even though each program trap has a different assembly routine. When the new processor state is loaded and the handler is running it must know which kind of trap occured. The assembly routine will put the *Exception Vector Number* in bytes 70 and 71 of the *Old* area.

6.2. Interrupts

An interrupt is signaled with an *interrupt request*. There are eight interrupt priority levels (numbered 0 through 7). Each device is associated with one of these levels at system configuration time. Zero or more devices can be associated with the same priority level. Among the unmasked interrupt requests, the one with the highest priority is serviced first.

For each priority level, the assembly routines that we provide define an *interrupt area* starting at a predefined location in memory. These areas are similar to the trap areas presented in the previous section.

In EMACSIM the five distinct sources of interrupts are:

Clock
Floppy
Disk
Printer
Terminal

EVT Addresses for Devices

Each device has a unique device number which is defined in the const.h file and which we will call devnum. The *EVT* address is computed as 0x100 + devnum * 4. Note that 0x100 is the start of the User Interrupt Vector area in the *EVT*. By convention the *EVT* address for the clock is 0x140 = 0x100 + 16 * 4.

Priority	Interrupt Type	Interrupt Area Address (convention)
0	Terminal	0x9c8
1	Printer	0xa60
2	Disk	0xaf8
3	Floppy	0xb90
5	Clock	0xc28

Fig 6. *Interrupt Vector Locations*

7. Input/Output Devices

The following devices are supported by EMACSIM:

Terminals (at most 5)
Printers (at most 2)
Disks (at most 4)
Floppies (at most 4)

Associated with each device are several *Device Registers*. These registers reside in memory and can be accessed as ordinary memory words. The starting address of the device registers for each device is shown in Table 7.

In order to request an operation from a device, its Operation register is loaded with a code for the desired operation. This triggers the device to perform the specified operation. In carrying out the operation, the device can read and/or write to memory without interrupting the CPU. When the device finishes performing an operation, it places status information into the Status register and makes an interrupt request. Interpretation of the various completion codes are given in Table 8.

If the Operation register is loaded while an operation is already in progress, the corresponding device behaves as if the first operation was never requested and starts the

Device	Starting Address	Range of i
Terminal(i)	0x1400 + 0x10*i	$0 \le i \le 4$
Printer(i)	0x1450 + 0x10*i	$0 \le i \le 1$
Disk(i)	0x1470 + 0x10*i	$0 \le i \le 3$
Floppy(i)	0x14b0 + 0x10*i	$0 \le i \le 3$

Table 7. *Device register locations*

Condition	Relevant Devices	Code
Successful Completion	All	0
Hardware Failure	All	1
Invalid Operation	All	2
Invalid Buffer	All	3
Invalid Length	Printer, Terminal	4
Invalid Track No.	Disk	5
Invalid Sector No.	Disk, Floppy	6
End of Input	Terminal	7
Non-Existent Device	All	8
Device Not Ready	Printer,Terminal	9

Table 8. *Status register codes*

new request as described above. Other registers of the device can be read or loaded without affecting the device's current operation.

If a request is made to a device that does not exist in the current configuration, the request fails with the ''Non-Existent Device'' status code.

7.1. Disk

Each disk device has *UPTRACK*+1† tracks (numbered 0 to *UPTRACK*) (99) and each track has *UPSECTOR*+1 (7) sectors (numbered 0 to *UPSECTOR*) of 512 bytes. Disks can be read/written on a per sector basis. Each Disk has four registers. They are shown in Table 9 in order of increasing address.

The Status register is set upon completion of a Disk operation; the other registers are set by the user. Operations performed by the Disk have the codes as indicated in

Operation
Disk Address
Buffer Address
Status

Table 9. *Disk device registers positions*

† System configuration parameters such as the number of devices of each type and device capacities are defined by constants within the simulator and have the same names as given here. To change any one of them, it suffices to edit the configuration file and recompile the system.

Table 10.

A *Disk Seek* moves the disk head to the track number specified in the Disk Address register. A *Disk Read* operation reads a sector of the current track and stores it in the 512-byte buffer whose first location address is given in the Buffer Address register. Similarly, a *Disk Write* operation writes the contents of the 512-byte buffer addressed by the Buffer Address register onto a sector of the current track. For a Disk Read or Write, the sector number is always taken from the Disk Address register. The Buffer Address register always contains a *physical* address.

7.2. Floppy

EMACSIM uses the real floppy of the Macintosh II. The floppies have 100 tracks and each track has 8 sectors of 512 bytes. The total capacity of a single sided floppy is 400K. The floppy has the same registers as the disk, is used in the same way and the same codes are loaded in its status register.

7.3. Printer

Four device registers are associated with printers:

The printer performs only one operation—*Print Line* (operation code 1)—which prints the string of bytes (up to *UPAMOUNT* (128) bytes in length) pointed to by the Buffer Address register. A ‘New Line’ character is appended to the string. An attempt to write more than *UPAMOUNT* (128) bytes in a *single* operation results in the ‘‘Invalid

Operation	Code
Disk Read	0
Disk Write	1
Disk Seek	2

Table 10. *Disk device operation codes*

Operation
Length
Buffer Address
Status

Table 11. *Printer device registers*

Length'' completion code to be returned without performing any I/O. All output written to printer number *i* is collected in the file named *printer*i in the current directory. That is, output for printer number 0 will be in the file *printer0*, printer number 1 in the file *printer1*, etc. If the appropriate file cannot be created in the current directory (due to lack of permission), the operation terminates with the ''Device Not Ready'' status code.

The Length register contains the number of bytes, starting at the Buffer Address, to be printed. The device completion Status codes are as given in Table 8.

7.4. Terminal

Terminals can be written to or read from one line at a time. Each terminal has the same registers as a Printer. Terminal Status register codes are as given in the Table 8. A *Terminal Write* operation (code 1) is performed in exactly the same way as a Printer Print operation. All output written to terminal number *i* is collected in the file named *termout*i in the current directory. As with the printers, if the appropriate file cannot be created in the current directory, the operation terminates with the ''Device Not Ready'' status code.

A *Terminal Read* operation (code 0) reads a string of bytes (from the file named *termin*i in the current directory) up to a 'New Line' character (octal 012) or *UPAMOUNT* (128) bytes, whichever occurs first, and stores them in the buffer pointed to by the Buffer Address register. The 'New Line' character is not considered to be part of the string. The actual number of bytes read is stored in the Length register and the ''Successful Completion'' code is returned. Subsequent reads from the same terminal will continue to read from the point where the last read operation left off. The last read operation which exhausts the available data associated with the terminal returns the ''End of Input'' rather than the ''Successful Completion'' status code (as usual, the Length register will contain the count of characters read). Subsequent read operations will continue to return the ''End of Input'' status. If on a read operation from terminal *i*, the device is not ready for input (the file *termin*i does not exist in the current directory), the status register is loaded with the ''Device Not Ready'' code.

8. The Memory Management System

An EMACSIM address can be interpreted either as a physical address or as a virtual address. An address is interpreted as a physical address whenever the memory mapping mechanism is off (i.e. SR.M=0) and interpreted as a virtual address otherwise (i.e. SR.M=1). In EMACSIM, the virtual space is 16M with addresses ranging from 0 to 0xffffff. The virtual address space is divided into 512-byte units called *pages*. The physical address space is 128K and it is divided into 512-byte units called *page frames*.

When memory mapping is in effect, a virtual address is automatically mapped into a physical address. The CPU Root Pointer, CRP, contains the physical address of a *Segment Table*, which is composed of thirty two entries. Each entry is eight bytes long and is called a *Segment Descriptor* (SD).

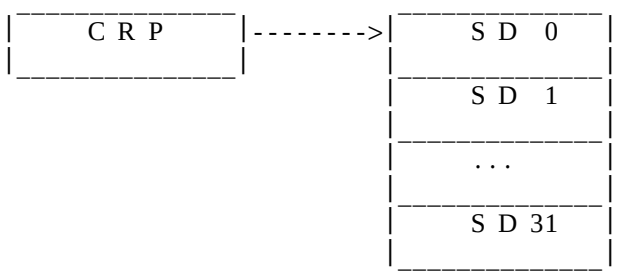
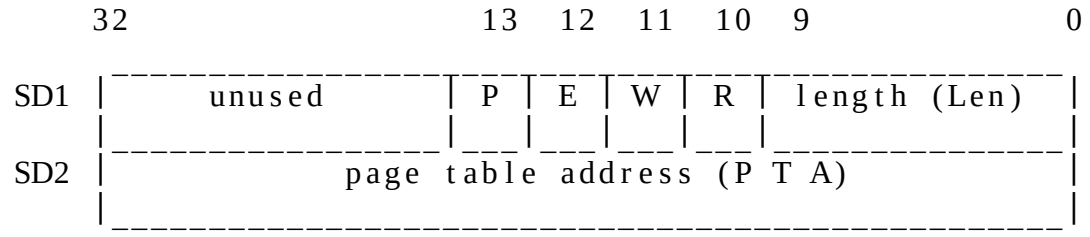


Figure 3. *Segment table*

A segment descriptor has the following format:



FIELD	MEANING
Len	Maximum valid page number within segment (one less than actual segment size)
E,W,R	Execute, Write and Read access rights for the segment, respectively; if 1, the corresponding access right is permitted
P	Presence bit; if 0, a missing segment trap is generated
PTA	physical address of the corresponding page table

Table 2. *Segment descriptor fields*

A *page table* can have up to 1024 entries. One less than its actual size is given by the SD1.Len field as defined above. Each entry in a page table is 4 bytes long and is called a *page descriptor* (*PD*).

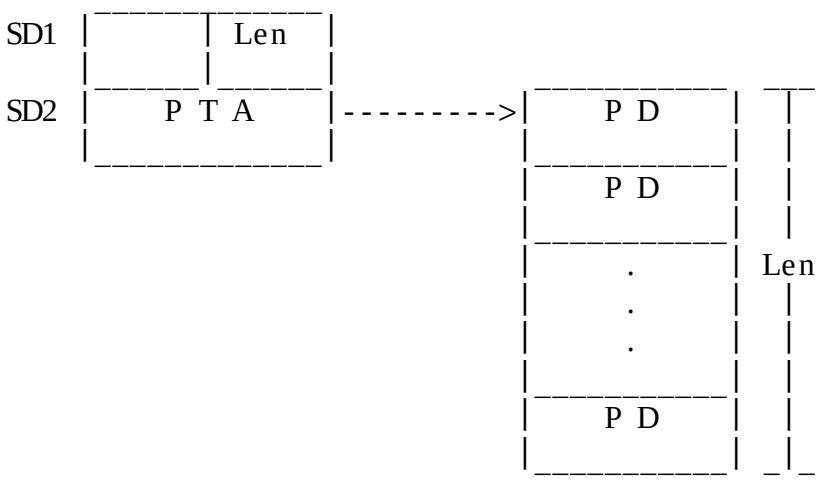
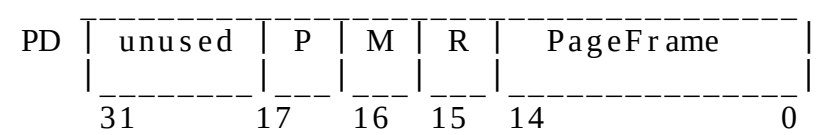


Figure 4. *Page table*

The format of a page descriptor is given below:



FIELD	MEANING
P	Presence bit; if 0, a missing page trap is generated
M	Modified bit; set to 1 by the processor whenever the page is modified
R	Reference bit; set to 1 whenever the page is read, written or executed
PageFrame	if the Presence bit is set, contains the number of the page frame to which this page is mapped; otherwise undefined

Table 3. *Page descriptor fields*

A page in EMACSIM must be aligned at an address that is a multiple of 512 bytes. Consequently, the real address of any page is a 24-bit number with the least significant nine bits equal zero. Therefore, the fifteen bits of PD.PageFrame are sufficient to uniquely identify a page frame.

8.1. Address Mapping

A *virtual address* in EMACSIM is composed of three fields:

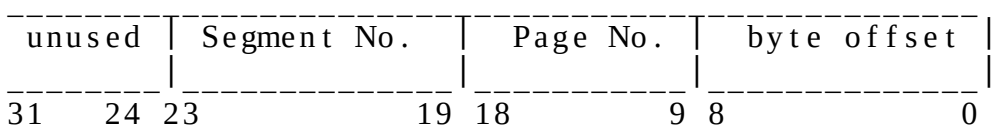


Figure 5. *Virtual address format*

Given a virtual address *va*, the corresponding physical address is calculated as follows:

1. The contents of CRP (CPU Root Pointer) is added to *va*.Seg, resulting in the physical address of a segment descriptor (*sd*) in the segment table.

   $$sd := (CRP + va.Seg)$$

2. If the page table for the segment *sd* is marked as present, i.e. *sd*.P=1, and if *va* is a valid page number, i.e. *va*.Page ≤ *sd*.Len, and the current access mode does not violate the access rights of the segment, then *sd*.PTA is added to *va*.Page, resulting in the physical address of a page descriptor (*pd*). Otherwise, a Memory Management Trap occurs.

   $$pd := (sd.PTA + va.Page)$$

3. If the page is marked as present, i.e. *pd*.P=1, the physical address is calculated by adding *pd*.PageFrame*512 to *va*.offset. If the page is not present, a Memory Management Trap occurs.

Step	Condition	Condition Name
2	*sd*.P=0	Segment Missing
2	*va*.Page>*sd*.Len	Invalid Page number
2	A read, write or execute access is attempted when the corresponding *sd*.R, *sd*.W or *sd*.E is equal to 0.	Access Protection Violation
3	*pd*.P=0	Page Missing

Table 4. *Sources for Memory Management traps during memory mapping*

pa := *pd*.PageFrame*512+*va*.offset

The Memory Management Traps that might occur during the mapping are summarized in Table 4 in the order in which the conditions are tested.

The mapping of a virtual address *va* into a physical address *pa* can be expressed as:

pa :=((CRP+*va*.Seg).PTA+*va*.Page).PageFrame*512+*va*.offset

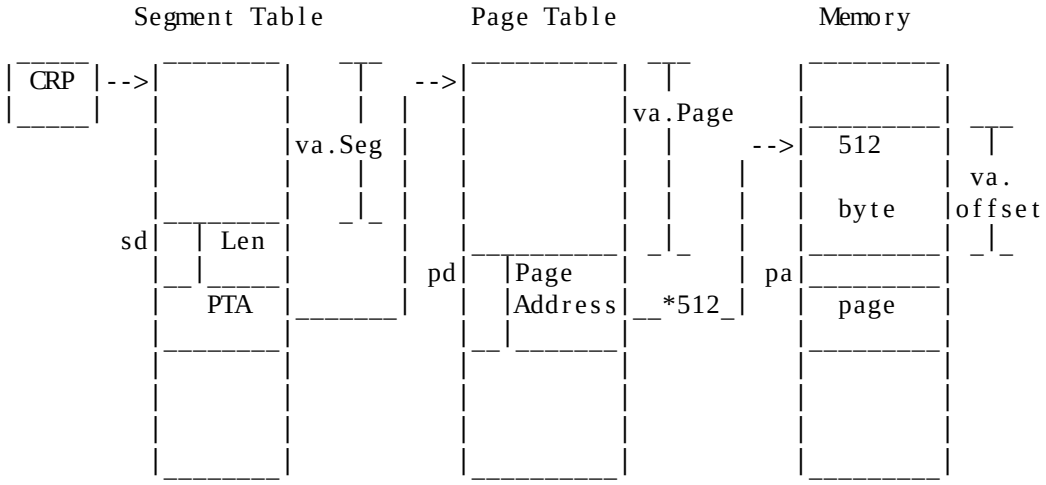


Figure 6. *Memory mapping*