HARDWARE INFORMATION

SPECIAL REGISTERS

There are nine special registers, as follows

FLAGS	A single word containing all of the one-bit flags
PDBR	Page Directory Base Register
INTVEC	The address of the interrupt vector
CGBR	Call Gate Base Register
CGLEN	Number of call gates
DEBUG	If the value of PC ever = this value, a debug interrupt is signalled
TIMER	Reduced by 1 after each instruction, causes timer interrupt when zero
SYSSP	System stack pointer. If in system mode, equivalent to SP
SYSFP	System frame pointer, not so useful.

The assembler understands the names of these registers (put a $ sign in front), they stand for the numbers 0 to 9 in instruction operands.

There are two instructions that directly access the special registers:

GETSR	loads a special register value into a normal register
SETSR	stores a normal register value into a special register

Example: how to set the TIMER register to 100:

```
LOAD    R1, 100
SETSR   R1, $TIMER
```

The value stored in $PDBR is always treated as a physical memory address.
The values stored in $INTVEC, $CGBR, $DEBUG, $SYSSP, and $SYSFP are treated as virtual addresses when virtual memory is turned on.

FLAGS

There are seven one-bit CPU flags, as follows

R	Indicates that the CPU is running, not halted
Z	Zero. Set by some instructions to indicate a zero (or equal) result.
N	Negative. Set by some instructions to indicate a negative result.
ERR	Error. Used only by the PERI instruction. Zero means success.
SYS	Set when CPU is in system mode, Zero when in user mode.
IP	Interrupt in progress. Set to 1 to ignore interrupts.
VM	Virtual Memory. If zero, all memory accesses use physical addresses, if set, page tables must be correctly set up, all memory addresses are translated.

The final three, SYS, IP, and VM, may only be modified when the CPU is in system mode.
At start-up, SYS=1, IP=1, VM=0.

The assembler understands the names of these flags (put a $ sign in front), they stand for the numbers 0 to 6 in instruction operands.

There are two instructions that directly access the special registers:

GETFL	loads the value of a single flag into a register
SETSR	sets a single flag equal to a register value (0 for off, non-0 for on)

The COMP and COMPZ instructions set or clear both Z and N, depending on the result.
The JCOND instruction jumps if the flags have a particular combination of values.

All the flag values may be read at once, using the GETSR instruction on the $FLAGS special register. The flags occupy the least significant bits of the value, in the order shown above. R is the least significant bit, VM is bit 6 (equivalent value 64).

All the flag values may be set at once using the SETSR instruction on the $FLAGS special register.

Example: Turn the SYS flag off, and the VM flag on, leaving other flags untouched:

```
GETSR   R1, $FLAGS
CBIT    R1, $SYS
SBIT    R1, $VM
SETSR   R1, $FLAGS
```

The special instruction FLAGSJ sets all the flags at once, and causes an unconditional jump by setting the PC. The only real point of this weird instruction is that it lets you turn on virtual memory without crashing the system. As soon as the VM flag is turned on, virtual-to-physical address translation begins for all memory accesses, so in the example above, if the program counter = 101 for the first instruction the GETSR is fetched from physical location 101, the CBIT is fetched from physical location 102, the SBIT is fetched from physical location 103, then suddenly physical addresses are not used any more, and the next instruction is fetched from *virtual* address 104. Unless virtual address 104 maps to physical address 104 (which would not make much sense), everything fails. This sequence:

```
GETSR   R1, $FLAGS
CBIT    R1, $SYS
SBIT    R1, $VM
FLAGSJ  R1, xxx
```

is safe. Of course ‘xxx’ should be replaced by the correct virtual address for program continuation.

INTERRUPTS

There are interrupts that represent a fatal problem (such as a user mode program attempting a privileged operation) and there are interrupts that represent some useful notification (such as keyboard input ready, or countdown timer reached zero). If interrupts are being processed (that is, the IP flag is 0, and the INTVEC special register contains the address of a proper interrupt vector), then all interrupts are trappable, regardless of how fatal they are.

If interrupts are being ignored (IP flag is 1), then fatal interrupts still stop a program, but notification interrupts are just ignored.

If interrupts are being accepted (IP=0) and a particular interrupt arises, but the interrupt vector is invalid, a second interrupt, INTRFAULT, is signalled. This may also be trapped, but

given that it is caused by the failure to correctly process another interrupt, it will probably turn out to be fatal.

Beware of this. Problems with regular programs (system or user mode) cause interrupts, and that is fine. The interrupt gives the system a chance to correct whatever condition caused it. BUT interrupt handling functions have no backup. If an interrupt handler causes a non-trivial interrupt, even a page fault, it will normally be fatal.

The INTRFAULT interrupt is the last chance to avoid a big crash. If you have a handling function for INTRFAULT stored in the interrupt vector, it will be called if a fatal interrupt occurs during interrupt processing, but it will not be able to return to processing the original interrupt after fixing the situation.

There are 14 interrupts defined, each with a name known to the assembler. Their names all begin with "IV$". An interrupt vector is really an array, and must be at least 14 words long. To be used, its address must be stored in the special register INTVEC. Each entry in the array is either zero (the corresponding interrupt will not be handled) or the address of an almost perfectly normal function that will be called automatically whenever the relevant interrupt occurs. The only special requirement is that interrupt handling functions must use IRET in all places instead of RET.

The defined interrupts are:

IV$NONE	= 0:	(not a real interrupt code)
IV$MEMORY	= 1:	Physical memory access failed
IV$PAGEFAULT	= 2:	Page fault
IV$UNIMPOP	= 3:	Unimplemented operation code (i.e. instruction opcode wrong)
IV$HALT	= 4:	HALT instruction executed
IV$DIVZERO	= 5:	Division by zero
IV$UNWROP	= 6:	Unwritable instruction operand (e.g. INC 72)
IV$TIMER	= 7:	Countdown timer reached zero
IV$PRIVOP	= 8:	Privileged operation attempted by user mode program
IV$KEYBD	= 9:	at least one keyboard character typed and ready
IV$BADCALL	= 10:	Bad SYSCALL index (i.e. <0 or >=$CGLEN)
IV$PAGEPRIV	= 11:	User mode access to system mode page
IV$DEBUG	= 12:	PC=$DEBUG trap
IV$INTRFAULT	= 13:	Failure to process interrupt.

The IV$ values are the positions in the interrupt vector where the handler function's address should be stored.

Example: How to set up an interrupt handler that automatically prints a dot whenever a keyboard key is pressed, and a star whenever another 5000 instructions have been executed...

```
LOAD    R1, TIMHANDLER
STORE   R1, [IVEC+IV$TIMER]
LOAD    R1, KBHANDLER
STORE   R1, [IVEC+IV$KEYBD]
LOAD    R1, IVEC
SETSR   R1, $INTVEC
LOAD    R1, 0
```

```
        SETFL       R1,   $IP
        LOAD        R1,   5000
        SETSR       R1,   $TIMER
        ......

  TIMHANDLER:
        LOAD        R1,   '*'
        CALL        PRINTCHARACTER      // which you would have to write somewhere
        LOAD        R1,   5000
        SETSR       R1,   $TIMER
        IRET

   KBHANDLER:
        LOAD        R1,   '.'
        CALL        PRINTCHARACTER
NOTE this interrupt will be repeatedly signalled until the character is consumed.
        IRET

   IVEC:
        .SPACE      16
```

ACTIONS AUTOMATICALLY PERFORMED WHEN AN INTERRUPT OCCURS, IF IP FLAG IS 0.

oldflags = FLAGS register
flag SYS turned on. (i.e. now using system SP and system stack)
flag IP turned on.
PUSH R0
PUSH R1
...
...
PUSH R11
PUSH R12
PUSH SP
PUSH FP
PUSH PC
PUSH additional interrupt information if available
PUSH interrupt-causing address
PUSH interrupt code (i.e. position in interrupt vector)
PUSH oldflags
PC = memory[$INTVEC + interrupt code]

These are exactly the same as the SYSCALL actions, except for the three values pushed after the 16 registers. These are information that may be needed to correctly handle the interrupt.

Note that if the interrupt handler behaves like a normal function, and performs "PUSH FP" and "LOAD FP, SP" as its first actions, then those three pieces of information will be available at [FP+2], [FP+3], and [FP+4].

The first parameter is always the interrupt code, the IV$ value for the interrupt.

For the following interrupts:

PAGEFAULT, PAGEPRIV,
the second parameter is the virtual address that caused the problem.

For this interrupt:

MEMORY,
the second parameter is the physical address that caused the problem.

For the following interrupts:

UNIMPOP, HALT, DIVZERO, UNWROP, PRIVOP, BADCALL, DEBUG,
the second parameter is the address of the instruction that caused the problem (i.e. PC value).

For this interrupt:

BADCALL,
the third parameter is the operand of the SYSCALL instruction that caused the problem.

For this interrupt:

INTRFAULT,
which is only caused by a fatal error during interrupt processing, the second parameter is left unchanged from the original interrupt's setting, and the third parameter is set to the interrupt code for the original interrupt.

Realise that is each process has its own system stack, then each process must also have its own value for the system stack pointer, which must be saved and restored when processes are switched.

INPUT AND OUTPUT OPERATIONS

All interactions with any hardware outside of the CPU are controlled by the PERI instruction. There are four general groups of IO operations supported:

Disc Operations: These allow direct access to the emulated disc drives, permitting whole blocks (128 words, which is the same size as 512 bytes) to be transferred between memory and a specified location on the disc. These operations are necessary for file-system implementation.

Magnetic Tape Operations: These provide a realistic way of accessing files in the real (i.e. outside the emulator, probably unix) file system. Without these it would be very difficult and time consuming to get useful test data into your own file system implementations.

Terminal Operations: These allow characters to be read from the controlling keyboard or written to appear on the monitor.

Time Operations: There is only one. It reads the emulated hardware clock and tells you the date and time.

All IO operations are controlled in the same way. A small lump of memory is filled with information describing the operation to be performed, and with space to receive the results. The PERI instruction sends these few words to the appropriate piece of hardware. When the operation is complete, data returned by the hardware, if any, is stored back into the small lump of memory, a success-or-error code (zero or positive for success, negative for failure) is put into the

instruction's main register, and execution continues. The ERR flag is also cleared for success and set for failure.

Example: Finding the total size of disc drive number one.

The SIZEDISC IO operation requires a three-word control structure. All IO control structures must have the required operation code, in this case $SIZEDISC, stored in the first word. The SIZEDISC operation also requires the second word do contain the disc drive number. The third word is used to deliver the answer:

```
        LOAD    R2,     control
        LOAD    R1,     $SIZEDISC
        STORE   R1,     [R2]
        LOAD    R1,     1
        STORE   R1,     [R2+1]
        LOAD    R1,     0
        STORE   R1,     [R2+2]
        PERI    R3,     control
        JCOND   ERR,    failed
        LOAD    R1,     [R2+2]
        BREAK
        ...etc...
control: .SPACE 3
```

If the operation is not successful, the ERR flag will be set, and the program will jump to the "failed:" label to deal with the situation, and R3 will contain a negative number as an error code. If the operation is successful, then by the time the BREAK instruction is reached, R1 and R3 will both contain the total number of blocks in disc number 1.

Of course, control structures may be set up in advance, like this:

```
        PERI    R3,     control
        JCOND   ERR,    failed
        LOAD    R1,     [control+2]
        BREAK
        ...etc...
control: .DATA  $SIZEDISC
        .DATA   1
        .DATA   0
```

This style requires fewer instructions, but is slightly less flexible.

PERI is a privileged operation, and can not be executed in user mode.

DISC OPERATIONS

Disc drives are set up at system initialisation. The `system.setup` file describes the disc drives that are needed. An example line from `system.setup` is "`disc maindrive 6000`". If this is the first "`disc`" command encountered, it means that disc drive number 1 should be at least 6000 blocks long, and will actually be kept in the real file `maindrive.disc`. If such a file does not exist, it is created. If the file does exist, it is used as-is. The size of `maindrive.disc` will of course be 6000*512 bytes.

$READDISC

Requires 5 word control structure, as follows
0: the value $READDISC
1: disc drive number
2: the number of consecutive blocks to be read
3: (disc address) the number of the first block to be read
4: (memory address) the address into which the data should be stored.
make sure that there are at least (number of blocks * 128) words of space there.

Error codes:
-2: memory problem, either reading the control structure or writing the data.
-3: invalid disc drive number
-4: invalid block number

Successful result (returned in register):
number of blocks transferred from disc to memory.

$WRITEDISC

Requires 5 word control structure, as follows
0: the value $WRITEDISC
1: disc drive number
2: the number of consecutive blocks to be written
3: (disc address) the number of the first block to be written
4: (memory address) the address where the data to be written may be found.

Error codes:
-2: memory problem, either reading the control structure or reading the data.
-3: invalid disc drive number
-4: invalid block number

Successful result (returned in register):
number of blocks transferred from memory to disc.

$SIZEDISC

Requires 3 word control structure, as follows
0: the value $SIZEDISC
1: disc drive number
2: output only: used to receive the answer

Error codes:
-2: memory problem reading the control structure.
-3: invalid disc drive number

Successful result (returned in register):
number of blocks in the disc

Also sets third word of control structure to contain number of blocks.

MAGNETIC TAPE OPERATIONS

Real files in the outside operating system are made available in the guise of magnetic tapes. There is only one magnetic tape drive in the system. To access a real file, a program must first load that file onto the tape drive. It may then either read the file sequentially in units of 128 word blocks, or it may write to the file sequentially in units of 128 word blocks. There is no ability to move directly to any particular block. Finally, the tape drive must be unloaded. Files/tapes are automatically rewound to the beginning when they are loaded.

$MTLOAD

Requires 3 word control structure, as follows
0: the value $MTLOAD
1: 0 for read access, or 1 for write access.
if zero, the file must already exist;
if one, a new file will be created regardless of whether it already existed or not.
2: a pointer to a string containing the real file name.

Error codes:
-2: memory problem, either reading the control structure or reading the filename.
-5: could not open the real file

Successful result (returned in register):
1

$MTUNLOAD

Requires 1 word control structure, as follows
0: the value $MTUNLOAD

Error codes:
-3: invalid request, no tape is loaded

Successful result (returned in register):
1

$MTREAD

Requires 3 word control structure, as follows
0: the value $MTREAD
1: the number of consecutive blocks to read
2: (memory address) the address into which the data should be stored.
make sure that there are at least (number of blocks * 128) words of space there.

Error codes:
-2: memory problem, either reading the control structure or writing the data.
-3: invalid request, no tape is loaded
-5: error in accessing the real file

Successful result (returned in register):
number of blocks transferred from tape/file to memory.

Note:
It is not an error to attempt to read more blocks than the file contains

$MTWRITE

Requires 3 word control structure, as follows
0: the value $MTWRITE
1: the number of consecutive blocks to written
2: (memory address) the address where the data to be written may be found.

Error codes:
-2: memory problem, either reading the control structure or writing the data.
-3: invalid request, no tape is loaded
-5: error in writing to the real file

Successful result (returned in register):
number of blocks transferred from memory to tape/file.

Example: Reading the first 512 characters from a real unix file and displaying them.

```
LOAD   R1, control
LOAD   R2, $MTLOAD
STORE  R2, [R1]
LOAD   R2, 0                  // read only
STORE  R2, [R1+1]
LOAD   R2, filename
STORE  R2, [R1+2]
PERI   R3, control            // have the tape loaded
JCOND  ERR, failed

LOAD   R2, $MTREAD
STORE  R2, [R1]
LOAD   R2, 1                  // number of blocks
STORE  R2, [R1+1]
LOAD   R2, space              // where to put those characters
STORE  R2, [R1+2]
PERI   R3, control            // read from the tape
JCOND  ERR, failed
```

```
        LOAD    R2, $TERMOUTC
        STORE   R2, [R1]
        LOAD    R2, 512                 // number of characters
        STORE   R2, [R1+1]
        LOAD    R2, space               // where those characters are
        STORE   R2, [R1+2]
        PERI    R3, control             // print
        HALT

filename:
        .STRING "tests/file.txt"
control:
        .SPACE  3
space:
        .SPACE  128
```

TERMINAL OPERATIONS

There are two essential operations: read a bunch of characters from the keyboard and write a bunch of characters to the screen. The read function is compatible with interrupt-driven user input: when a program is running (not just single stepping) and interrupts are enabled, every time a keyboard key is pressed its ascii code is added to the end of the hardware keyboard buffer and a $KEYBD interrupt is signalled. The $TERMINC operation takes characters from the beginning of the hardware keyboard buffer.

Character codes are available as soon as the key is pressed, the system does not wait until a whole line is available. This means that any special behaviour associated with particular keys (such as ENTER or BACKSPACE) must be programmed. The one exception is control-c; that will always interrupt a running program and return to single stepping mode.

$TERMINC

Requires 3 word control structure, as follows
0: the value $TERMINC
1: the maximum number of characters to be read
2: (memory address) the address into which the characters should be stored.
make sure that there are at least (maximum number / 4) words of space there.

Error codes:
-2: memory problem, either reading the control structure or writing the data.

Successful result (returned in register):
number of characters actually read.

Notes:
It is not an error to attempt to read when the keyboard buffer is empty.

If no characters are already in the keyboard buffer, it will not wait for input.

The characters received are packed four per word to make a proper string, and that string will be zero terminated. Strings are organised so that the first character goes in the least-significant bits of the first word. This means that if just a single character is read, the first word of the result will simply be its ascii code.

Any characters left unread in the buffer will be received by the next TERMINC.

$TERMOUTC

Requires 3 word control structure, as follows

0: the value $TERMOUTC
1: the number of characters to be printed
2: (memory address) the address at which the characters may be found.

Error codes:

-2: memory problem, either reading the control structure or reading the string.

Successful result (returned in register):

the number of characters actually printed

Notes:

The characters to be printed should be in the form of a proper string (packed four per word) starting at the given memory location. The string does not *need* to be zero-terminated.

If the number of characters is specified to be zero, the string will be assumed to be zero-terminated, and an unlimited number of characters will be printed.

If the number of characters is specified to be non-zero, that number of characters will be printed, even if they include some zeros.

If the number of characters is specified to be one, then the memory location may just contain the character's ascii code; no extra formatting is required to make it into a string.

$TERMINW and
$TERMOUTW

These operations perform exactly as TERMINC and TERMOUTC with the following exceptions:

The data is not formatted as a string. Reading or printing N characters requires exactly N words of memory, containing one ascii code each.

The input operation will *not* zero-terminate the array of characters.

TIME OPERATIONS

$SECONDS

Requires 1 word control structure, as follows

0: the value $SECONDS

Error codes:
none

Successful result (returned in register):
The number of seconds elapsed since midnight (0000 hours) on 1st January 2000.

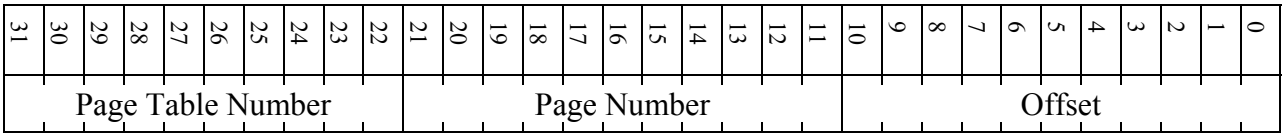
VIRTUAL MEMORY

Because the emulator uses 32 bit words instead of 8 bit bytes, the Intel scheme of splitting a virtual address into a 10 bit page table number, a 10 bit page number, and a 12 bit offset can not be used exactly.

A 12 bit offset means that there would be 4096 memory locations in a page, and that would mean that a page table could hold the addresses of 4096 pages instead of 1024, so we would not need so many of them.

In the emulator a page of memory consists of 2048 32-bit locations requiring only an 11 bit offset. That means that a page table can hold the addresses of 2048 pages, so 11 bits are required for page numbers. That leaves only 10 bits for the page table number, meaning that page directories only fill half a page.

A Virtual Address



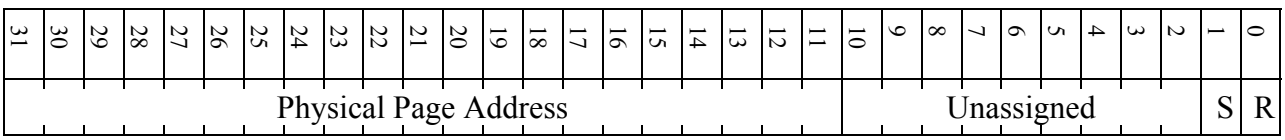
There are two advantages to this changed layout: pages are smaller, so more of them are available without using up so much real memory, and page directories only fill half a page, so it is quite possible that you can store everything you need to know about a process in one single page.

Only the most significant 22 bits of the value stored in the Page Directory Base Register are looked at during virtual address translation. Page tables must occupy complete half-pages; their addresses must be multiples of 1024 (i.e. in binary they must end in 10 zeros).

Only the most significant 21 bits of the values stored in the Page Directories are looked at during virtual address translation. Page tables must occupy whole pages; their addresses must be multiples of 2048 (i.e. in binary they must end in 11 zeros).

The entries in page tables include two page status bits in the least significant bits. They are the Resident or Valid bit (in bit 0) and the System bit (in bit 1). The meaning of a page table entry depends upon the value of the Resident bit.

A Page Table Entry



If the Resident bit is Zero, any access to this virtual page will immediately cause a PAGEFAULT interrupt, and the other 31 bits will not even be seen. They may be used for any purpose whatsoever.

If the System bit is 1, any attempted access to this virtual page while in User mode will result in a PAGEPRIV interrupt, and the access will not occur.

In all cases, bits 2 to 10 have no assigned meaning, and may be used for any purpose whatsoever.

MEMORY ACCESS ALGORITHM

```
let A be the address in memory referenced by an instruction.
if $VM flag is OFF:
    Use physical memory at address A
otherwise, if $VM flag is ON:
    // A is a virtual address and will be translated.
    let DIR be (A>>22)&0x3FF          // most significant 10 bits
    let PG be (A>>11)&0x7FF           // next 11 bits
    let OFFS be A&0x7FF               // least significant 11 bits
    let POS be DIR + contents of $PDBR register
    read PTADDR from physical memory address POS
    if PTADDR is Zero:
        PAGEFAULT, translation abandoned
    PTADDR &= 0xFFFFF800              // zero out least significant 11 bits
    read PGADDR from physical memory address (PG + PTADDR)
    let R be PGADDR & 1               // least significant bit
    if R is Zero:
        PAGEFAULT, translation abandoned
    let S be (PGADDR & 2)>>1          // second least significant bit
    if S is One and $SYS flag is Zero:
        PAGEPRIV, translation abandoned
    PGADDR &= 0xFFFFF800              // zero out least significant 11 bits
    let PHYS be PGADDR + OFFS
    Use physical memory at address PHYS
```

PRIVILEGED OPERATIONS

If any of the following instructions are executed when the $SYS flag is off (zero), a PRIVOP interrupt will be triggered and the operation will not be performed.

SETSR, PERI, IRET, PHLOAD, PHSTORE, FLAGSJ

If a HALT instruction is executed when the $SYS flag is off (zero), a HALT interrupt will be triggered and the processor will *not* be halted.

If a SETFL instruction is executed when the $SYS flag is off (zero), and it attempts to modify either the R, SYS, VM, or IP flag, a PRIVOP interrupt will be triggered and the operation will not be performed.